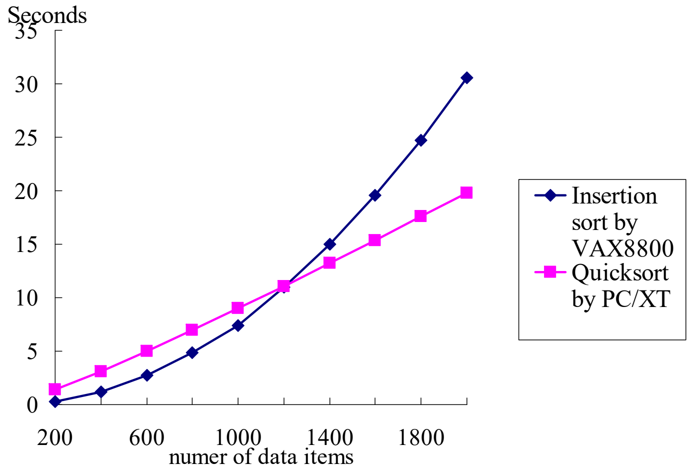
Chapter 1

Introduction

Why to study algorithms?

- Sorting problem:
  To sort a set of elements into increasing or decreasing order.
  11, 7, 14, 1, 5, 9, 10
  ↓sort
  1, 5, 7, 9, 10, 11, 14
- Insertion sort
- Quick sort

- Comparison of two algorithms implemented on two computers



Analysis of algorithms

- Measure the goodness of algorithms
  - efficiency
  - asymptotic notations: e.g. $O(n^2)$
  - worst case
  - average case
  - amortized
- Measure the difficulty of problems
  - NP-complete
  - undecidable
  - lower bound
- Is the algorithm optimal?

0/1 Knapsack problem

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Value	10	5	1	9	3	4	11	17
Weight	7	3	3	10	1	9	22	15

M(weight limit)=14

best solution: P_1, P_2, P_3, P_5(optimal)

This problem is NP-complete.

Traveling salesperson problem

- Given: A set of n planar points
  Find: A closed tour which includes all points exactly once such that its total length is minimized.
- This problem is NP-complete.

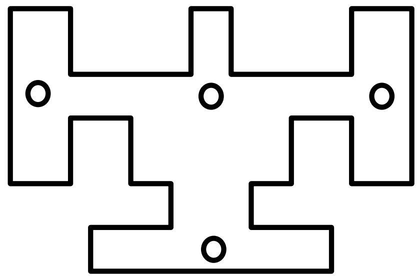
Partition problem

- Given: A set of positive integers S
  Find: S_1 and S_2 such that $S_1 \cap S_2 = \varnothing$, $S_1 \cup S_2 = S$, $\sum_{i \in S_1} i = \sum_{i \in S_2} i$

(partition into S_1 and S_2 such that the sum of S_1 is equal to that of S_2)

- e.g. S={1, 7, 10, 4, 6, 8, 3, 13}
  - S_1={1, 10, 4, 8, 3}
  - S_2={7, 6, 13}
- This problem is NP-complete.

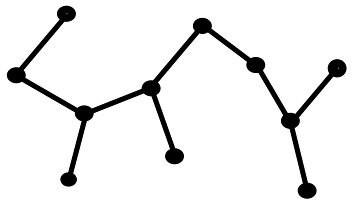
Art gallery problem



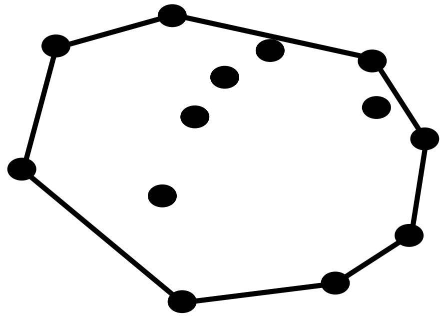
- Given: an art gallery
  Determine: min # of guards and their placements such that the entire art gallery can be monitored.
- NP-complete

Minimum spanning tree

- graph: greedy method
- geometry(on a plane): divide-and-conquer
- # of possible spanning trees for n points: n^{n-2}
- $n=10 \rightarrow 10^8$, $n=100 \rightarrow 10^{196}$

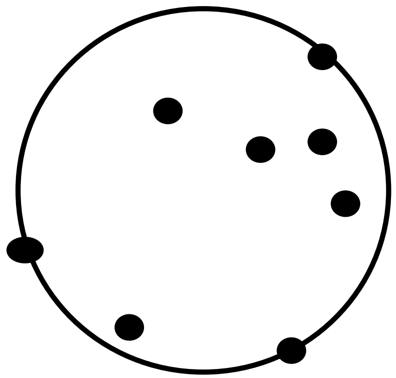


Convex hull



- Given a set of planar points, find a smallest convex polygon which contains all points.
- It is not obvious to find a convex hull by examining all possible solutions
- divide-and-conquer

One-center problem



- Given a set of planar points, find a smallest circle which contains all points.
- Prune-and-search

Chapter 2

The Complexity of Algorithms and the Lower Bounds of Problems

The goodness of an algorithm

- Time complexity (more important)
- Space complexity
- For a parallel algorithm :
  - time-processor product
- For a VLSI circuit :
  - area-time (AT, AT^2)

Measure the goodness of an algorithm

- Time complexity of an algorithm
  - efficient (algorithm)
  - worst-case
  - average-case
  - amortized

Measure the difficulty of a problem

- NP-complete ?
- Undecidable ?
- Is the algorithm best ?
  - optimal (algorithm)

- We can use the number of comparisons to measure a sorting algorithm.

Asymptotic notations

- Def: $f(n) = O(g(n))$ "at most"
  $\exists$ $c, n_0 \ni |f(n)| \le c|g(n)| \quad \forall\, n \ge n_0$
- e.g. $f(n) = 3n^2 + 2$
  $g(n) = n^2$
  $\Rightarrow n_0=2, c=4$
  $\therefore f(n) = O(n^2)$

- e.g. $f(n) = n^3 + n = O(n^3)$
- e. g. $f(n) = 3n^2 + 2 = O(n^3)$ or $O(n^{100})$

- Def : $f(n) = \Omega(g(n))$ "at least", "lower bound"
  $\exists$ c, and n_0, $\ni |f(n)| \ge c|g(n)| \ \forall\, n \ge n_0$
  e. g. $f(n) = 3n^2 + 2 = \Omega(n^2)$ or $\Omega(n)$

- Def : $f(n) = \Theta(g(n))$
  $\exists$ c_1, c_2, and n_0, $\ni c_1|g(n)| \le |f(n)| \le c_2|g(n)| \ \forall\, n \ge n_0$
  e. g. $f(n) = 3n^2 + 2 = \Theta(n^2)$

- Def : $f(n) \sim o(g(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \to 1$$

e.g. $f(n) = 3n^2+n = o(3n^2)$

Problem size

	10	10^2	10^3	10^4
$\log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n\log_2 n$	$0.33\text{x}10^2$	$0.7\text{x}10^3$	10^4	$1.3\text{x}10^5$
n^2	10^2	10^4	10^6	10^8
2^n	1024	$1.3\text{x}10^{30}$	$>10^{100}$	$>10^{100}$
n!	$3\text{x}10^6$	$>10^{100}$	$>10^{100}$	$>10^{100}$

Time Complexity Functions

Common computing time functions

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$
  - Exponential algorithm: $O(2^n)$
  - Polynomial algorithm: e.g. $O(n^2)$, $O(n\log n)$

- Algorithm A : $O(n^3)$, Algorithm B : $O(n)$
  - Should Algorithm B run faster than A?
    NO !
  - It is true only when n is large enough!

Analysis of algorithms

- Best case: easiest
- Worst case
- Average case: hardest

Straight insertion sort

input: 7,5,1,4,3

7,5,1,4,3

5,7,1,4,3

1,5,7,4,3

1,4,5,7,3

1,3,4,5,7

Algorithm 2.1 Straight Insertion Sort

Input: $x_1,x_2,\ldots,x_n$

Output: The sorted sequence of $x_1,x_2,\ldots,x_n$

For j := 2 to n do
Begin
    i := j-1
    x := x_j
    While x<x_i and i > 0 do
    Begin
        x_{i+1} := x_i
        i := i-1
    End
    x_{i+1} := x
End

Inversion table

- $(a_1,a_2,\ldots,a_n)$: a permutation of $\{1,2,\ldots,n\}$
- $(d_1,d_2,\ldots,d_n)$: the inversion table of $(a_1,a_2,\ldots a_n)$
- d_j: the number of elements to the left of j that are greater than j
- e.g. permutation (7 5 1 4 3 2 6)
inversion table 2 4 3 2 1 1 0
- e.g. permutation (7 6 5 4 3 2 1)
inversion table 6 5 4 3 2 1 0

Analysis of # of movements

- M: # of data movements in straight insertion sort

1 5 7 4 3

temporary

e.g. d_3=2

- $$M = \sum_{i=1}^{n-1} (2 + d_i)$$

Analysis by inversion table

- best case: already sorted

  d_i = 0 for $1 \le i \le n$

  $\Rightarrow$M = 2(n – 1) = O(n)

- worst case: reversely sorted

  d_1 = n – 1

  d_2 = n – 2

  :

  d_i = n – i

  d_n = 0

$$M = \sum_{i=1}^{n-1} (2 + d_i) = 2(n-1) + \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

- average case:

  x_j is being inserted into the sorted sequence

  x_1 x_2 ... x_{j-1}

- the probability that x_j is the largest: 1/j
  - takes 2 data movements
- the probability that x_j is the second largest : 1/j
  - takes 3 data movements

  :

- # of movements for inserting x_j:

1 4 7 5

$$\frac{2}{j} + \frac{3}{j} + \cdots + \frac{j+1}{j} = \frac{j+3}{2}$$

$$M = \sum_{j=2}^{n} \frac{j+3}{2} = \frac{(n+8)(n-1)}{4} = \mathrm{O}(n^2)$$

Analysis of # of exchanges

- Method 1 (straightforward)
- x_j is being inserted into the sorted sequence

  x_1 x_2 .... x_{j-1}

- If x_j is the *k*th ($1 \le k \le j$) largest, it takes (*k*–1) exchanges.
- e.g. 1 5 7↔4

  1 5↔4 7

  1 4 5 7

- # of exchanges required for x_j to be inserted:

$$\frac{0}{j} + \frac{1}{j} + \cdots + \frac{j-1}{j} = \frac{j-1}{2}$$

- # of exchanges for sorting:

$$\sum_{j=2}^{n} \frac{j-1}{2}$$
$$= \sum_{j=2}^{n} \frac{j}{2} - \sum_{j=2}^{n} \frac{1}{2}$$
$$= \frac{1}{2} \cdot \frac{(n-1)(n+2)}{2} - \frac{n-1}{2}$$
$$= \frac{n(n-1)}{4}$$

Method 2: with inversion table and generating function

$I_n(k)$: # of permutations in *n* nmbers which have exactly *k* inversions

n\k	0	1	2	3	4	5	6
1	1	0	0	0	0	0	0
2	1	1	0	0	0	0	0
3	1	2	2	1	0	0	0
4	1	3	5	6	5	3	1

- Assume we have $I_3(k)$, $0 \le k \le 3$. We will calculate $I_4(k)$.

(1) $a_1\ a_2\ a_3\ a_4$ ↑ largest $G_3(Z)$

(2) $a_1\ a_2\ a_3\ a_4$ ↑ second largest $ZG_3(Z)$

(3) $a_1\ a_2\ a_3\ a_4$ ↑ third largest $Z^2G_3(Z)$

(4) $a_1\ a_2\ a_3\ a_4$ ↑ smallest $Z^3G_3(Z)$

case	$I_4(0)$	$I_4(1)$	$I_4(2)$	$I_4(3)$	$I_4(4)$	$I_4(5)$	$I_4(6)$
1	$I_3(0)$	$I_3(1)$	$I_3(2)$	$I_3(3)$			
2		$I_3(0)$	$I_3(1)$	$I_3(2)$	$I_3(3)$		
3			$I_3(0)$	$I_3(1)$	$I_3(2)$	$I_3(3)$	
4				$I_3(0)$	$I_3(1)$	$I_3(2)$	$I_3(3)$

case	$I_4(0)$	$I_4(1)$	$I_4(2)$	$I_4(3)$	$I_4(4)$	$I_4(5)$	$I_4(6)$
1	1	2	2	1			
2		1	2	2	1		
3			1	2	2	1	
4				1	2	2	1
total	1	3	5	6	5	3	1

- generating function for $I_n(k)$

$$G_n(Z) = \sum_{k=0}^{m} I_n(k) Z^k$$

- for n = 4

$$G_4(Z) = (1 + 3Z + 5Z^2 + 6Z^3 + 5Z^4 + 3Z^5 + Z^6)$$
$$= (1 + Z + Z^2 + Z^3) G_3(Z)$$

- in general,

$$G_n(Z) = (1 + Z + Z^2 + \cdots + Z^{n-1}) G_{n-1}(Z)$$

$P_n(k)$: probability that a given permutation of *n* numbers has *k* inversions

- generating function for $P_n(k)$:

$$g_n(Z) = \sum_{k=0}^{m} P_n(k) Z^k = \sum_{k=0}^{m} \frac{In(k)}{n!} Z^k$$
$$= \frac{1}{n!} G_n(Z)$$
$$= \frac{1 + Z + Z^2 + \cdots + Z^{n-1}}{n} \cdot \frac{1 + Z + Z^2 + Z^{n-2}}{n-1} \cdots \frac{1+Z}{2} \cdot 1$$

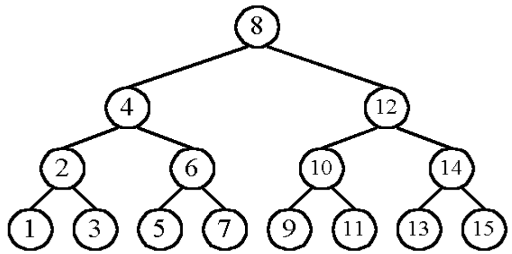
$$\sum_{k=0}^{m} k P_n(k) = g_n'(1)$$
$$= \frac{1 + 2 + \cdots + (n-1)}{n} + \frac{1 + 2 + \cdots + (n-2)}{n-1} + \cdots + \frac{1}{2} + 0$$
$$= \frac{n-1}{2} + \frac{n-2}{2} + \cdots + \frac{1}{2} + 0$$
$$= \frac{1}{4} n(n-1)$$

Binary search

- sorted sequence : (search 9)

	1	4	5	7	9	10	12	15
step 1				↑				
step 2						↑		
step 3					↑			

- best case: 1 step = O(1)
- worst case: $(\lfloor \log_2 n \rfloor + 1)$ steps = O(log n)
- average case: O(log n) steps



n cases for successful search
n+1 cases for unsuccessful search

Average # of comparisons done in the binary tree:

$$A(n) = \frac{1}{2n+1}\left(\underline{\sum_{i=1}^{k} i 2^{i-1}} + \underline{k(n+1)}\right), \text{ where } k = \lfloor \log n \rfloor + 1$$

Assume $n=2^k$

$\sum_{i=1}^{k} i\,2^{i-1} = 2^k(k-1)+1$

proved by induction on k

$$A(n) = \frac{1}{2n+1}((k-1)2^k + 1 + k(2^k+1))$$

$\approx k$ as n is very large

$= \log n$

$= O(\log n)$

Straight selection sort

-
  7 5 1 4 3
  1 5 7 4 3
  1 3 7 4 5
  1 3 4 7 5
  1 3 4 5 7
- Only consider # of changes in the flag which is used for selecting the smallest number in each iteration.
  - best case: O(1)
  - worst case: $O(n^2)$
  - average case: O(n log n)

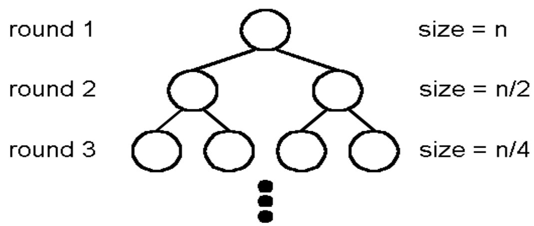
Quicksort

11	5	24	2	31	7	8	26	10	15
		↑						↑	
11	5	10	2	31	7	8	26	24	15
				↑		↑			
11	5	10	2	8	7	31	26	24	15
△					△				
7	5	10	2	8	11	31	26	24	15
|←		<11		→|		|←	>11		→|

- Recursively apply the same procedure.

Best case of quicksort

- Best case: O(nlogn)
- A list is split into two sublists with almost equal size.



- *log n* rounds are needed
- In each round, *n* comparisons (ignoring the element used to split) are required.

Worst case of quicksort

- Worst case: $O(n^2)$
- In each round, the number used to split is either the smallest or the largest.
- $$n+(n-1)+\cdots+1=\frac{n(n+1)}{2}=O(n^2)$$

Average case of quicksort

- Average case: $O(n \log n)$

s	n-s

include the splitter

$$T(n) = \underset{1\le s\le n}{\mathrm{Avg}}(T(s)+T(n-s))+cn, \qquad \text{c is a constant}$$

$$= \frac{1}{n}\sum_{s=1}^{n}(T(s)+T(n-s))+cn$$

$$= \frac{1}{n}(T(1)+T(n-1)+T(2)+T(n-2)+\cdots+T(n)+T(0))+cn,\ T(0)=0$$

$$= \frac{1}{n}(2T(1)+2T(2)+\cdots+2T(n-1)+T(n))+cn$$

$$(n-1)T(n) = 2T(1)+2T(2)+\cdots+2T(n-1) + cn^2 \cdots\cdots(1)$$
$$(n-2)T(n-1)=2T(1)+2T(2)+\cdots+2T(n-2)+c(n-1)^2\cdots(2)$$

(1) − (2)

$$(n-1)T(n) -(n-2)T(n-1) = 2T(n-1)+c(2n-1)$$
$$(n-1)T(n) -nT(n-1) = c(2n-1)$$

$$\frac{T(n)}{n}=\frac{T(n-1)}{n-1}+c\left(\frac{1}{n}+\frac{1}{n-1}\right)$$
$$=c\left(\frac{1}{n}+\frac{1}{n-1}\right)+c\left(\frac{1}{n-1}+\frac{1}{n-2}\right)+\cdots+c\left(\frac{1}{2}+1\right)+T(1),\ T(1)=0$$
$$=c\left(\frac{1}{n}+\frac{1}{n-1}+\ldots+\frac{1}{2}\right)+c\left(\frac{1}{n-1}+\frac{1}{n-2}+\ldots+1\right)$$

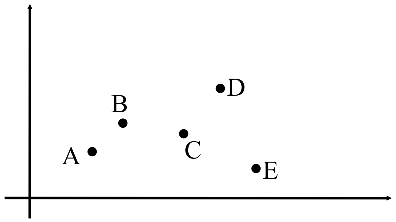
Harmonic number [Knuth 1986]

$$H_n = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$
$$=\ln n + \gamma + \frac{1}{2n}-\frac{1}{12n^2}+\frac{1}{120n^4}-\varepsilon, \text{ where } 0<\varepsilon<\frac{1}{252n^6}$$
$$\gamma = 0.5772156649\cdots.$$
$$H_n = O(\log n)$$

$$\frac{T(n)}{n} = c(H_n-1) + cH_{n-1}$$
$$= c\left(2H_n-\frac{1}{n}-1\right)$$
$$\Rightarrow T(n) = 2\,c\,n\,H_n - c(n+1)$$
$$=O(n \log n)$$

2-D ranking finding

- **Def**: Let A = (x_1,y_1), B = (x_2,y_2). B dominates A iff $x_2 > x_1$ and $y_2 > y_1$
- **Def**: Given a set S of n points, the rank of a point x is the number of points dominated by x.



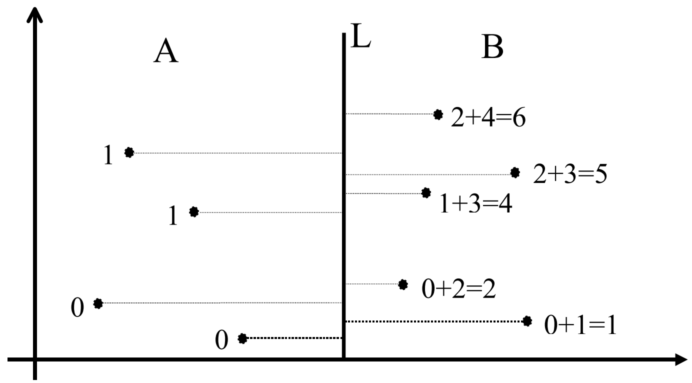
rank(A)= 0 rank(B) = 1 rank(C) = 1

rank(D) = 3 rank(E) = 0

- **Straightforward algorithm:**

compare all pairs of points : $O(n^2)$

- More efficient algorithm (divide-and-conquer)



Divide-and-conquer 2-D ranking finding

Step 1: Split the points along the median line L into A and B.

Step 2: Find ranks of points in A and ranks of points in B, recursively.

Step 3: Sort points in A and B according to their y-values. Update the ranks of points in B.

- time complexity : step 1 : O(n) (finding median)
  step 3 : O(n log n) (sorting)
- total time complexity :

$$T(n) \le 2T(\tfrac{n}{2}) + c_1 n \log n + c_2 n, \quad c_1, c_2 \text{ are constants}$$
$$\le 2T(\tfrac{n}{2}) + c\, n \log n, \quad \text{let } c = c_1 + c_2$$
$$\le 4T(\tfrac{n}{4}) + c\, n \log \tfrac{n}{2} + c\, n \log n$$
$$\le nT(1) + c(n \log n + n \log \tfrac{n}{2} + n \log \tfrac{n}{4} + \cdots + n \log 2)$$
$$= nT(1) + \frac{cn \log n(\log n + \log 2)}{2}$$
$$= O(n \log^2 n)$$

Lower bound

- **Def** : A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.
- ☆ **worst case lower bound**

  ☆ **average case lower bound**

- The lower bound for a problem is not unique.
  - e.g. $\Omega(1)$, $\Omega(n)$, $\Omega(n \log n)$ are all lower bounds for sorting.
  - ($\Omega(1)$, $\Omega(n)$ are trivial)

- At present, if the highest lower bound of a problem is $\Omega(n \log n)$ and the time complexity of the best algorithm is $O(n^2)$.
  - We may try to find a higher lower bound.
  - We may try to find a better algorithm.
  - Both of the lower bound and the algorithm may be improved.
- If the present lower bound is $\Omega(n \log n)$ and there is an algorithm with time complexity $O(n \log n)$, then the algorithm is optimal.

The worst case lower bound of sorting

6 permutations for 3 data elements

a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Straight insertion sort:

- input data: (2, 3, 1)

  (1) a_1:a_2

  (2) a_2:a_3, $a_2 \leftrightarrow a_3$

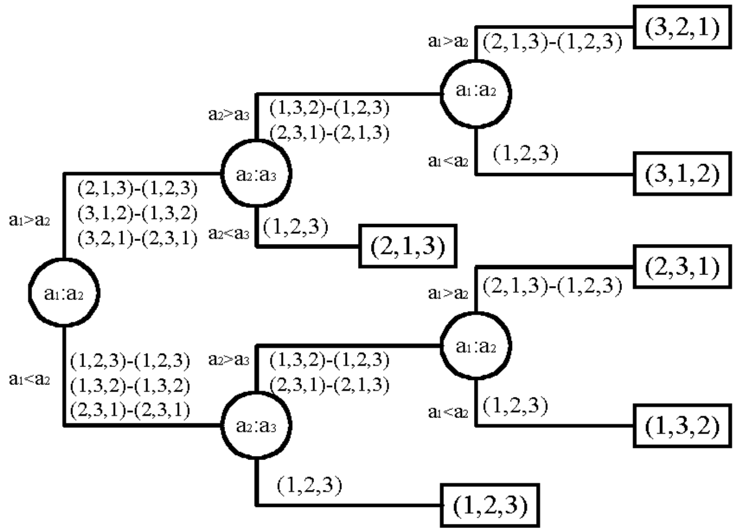
  (3) a_1:a_2, $a_1 \leftrightarrow a_2$

- input data: (2, 1, 3)

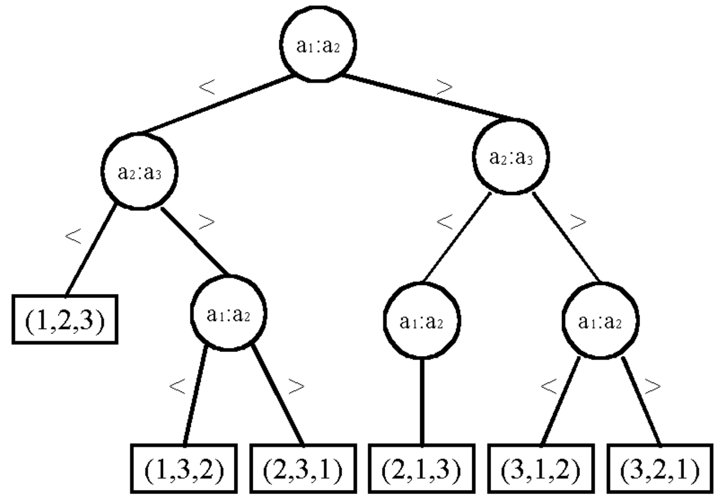
  (1)a_1:a_2, $a_1 \leftrightarrow a_2$

  (2)a_2:a_3

Decision tree for straight insertion sort



Decision tree for bubble sort



Lower bound of sorting

- Every sorting algorithm (based on comparisons) corresponds to a decision tree.
- To find the lower bound, we have to find the depth of a binary tree with the smallest depth.
- n! distinct permutations

  n! leaf nodes in the binary decision tree.
- balanced tree has the smallest depth:

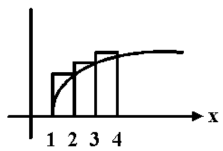
  $\lceil \log(n!) \rceil = \Omega(n \log n)$

  lower bound for sorting: $\Omega(n \log n)$

(See the next page.)

Method 1:

$$
\begin{aligned}
\log(n!) &= \log(n(n-1)\cdots 1) \\
&= \log 2 + \log 3 + \cdots + \log n \\
&> \int_1^n \log x\,dx \\
&= \log e \int_1^n \ln x\,dx \\
&= \log e[x \ln x - x]_1^n \\
&= \log e(n \ln n - n + 1) \\
&= n \log n - n \log e + 1.44 \\
&\geq n \log n - 1.44n \\
&= \Omega(n \log n)
\end{aligned}
$$



Method 2:

- Stirling approximation:

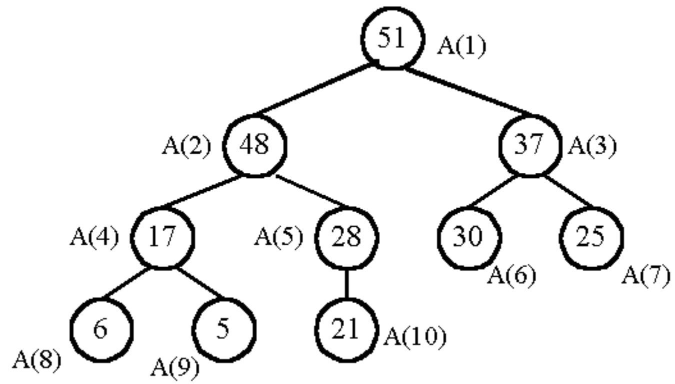
$$n! \approx S_n = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$\log n! \approx \log\sqrt{2\pi} + \frac{1}{2}\log n + n\log\frac{n}{e} \approx n\log n = \Omega(n\log n)$$

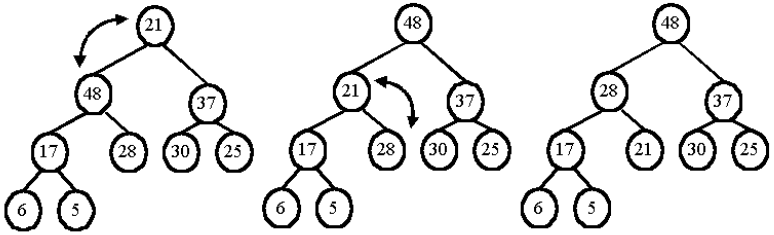
n	n!	S_n
1	1	0.922
2	2	1.919
3	6	5.825
4	24	23.447
5	120	118.02
6	720	707.39
10	3,628,800	3,598,600
20	$2.433\text{x}10^{18}$	$2.423\text{x}10^{18}$
100	$9.333\text{x}10^{157}$	$9.328\text{x}10^{157}$

Heapsort—An optimal sorting algorithm

- A heap : parent ≥ son



- output the maximum and restore:

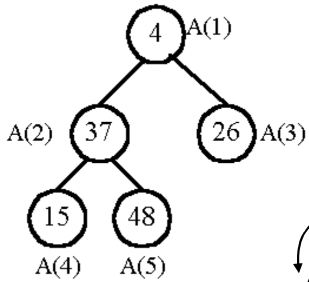


(a) (b) (c)

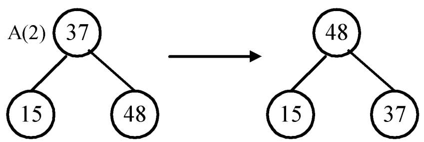
- Heapsort:
  - Phase 1: Construction
  - Phase 2: Output

Phase 1: construction

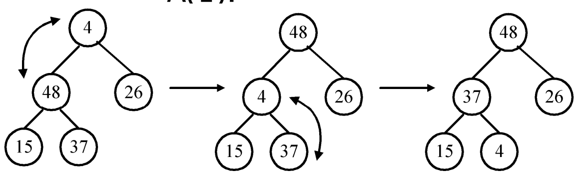
- input data: 4, 37, 26, 15, 48



- restore the subtree rooted at A(2):

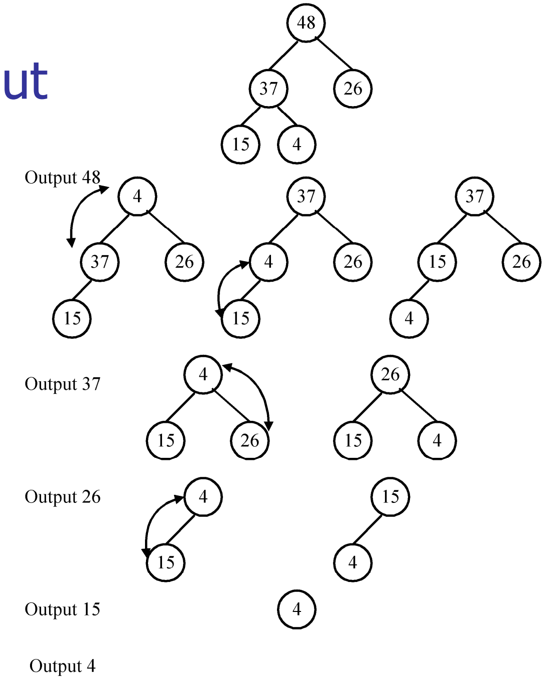


- restore the tree rooted at A(1):



A Heap

Phase 2: output



Implementation

- using a linear array
  not a binary tree.
  - The sons of A(h) are A(2h) and A(2h+1).
- time complexity: O(n log n)

Time complexity
Phase 1: construction

$d = \lfloor \log n \rfloor$: depth

# of comparisons is at most:

$$\sum_{L=0}^{d-1} 2(d-L)2^{L}$$

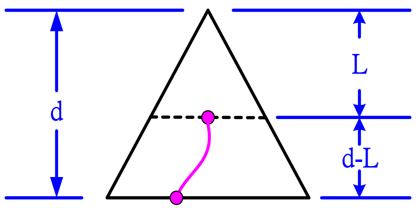
$$=2d\sum_{L=0}^{d-1} 2^{L} - 4\sum_{L=0}^{d-1} L2^{L-1}$$

$$\left(\sum_{L=0}^{k} L2^{L-1} = 2^{k}(k-1)+1\right)$$

$$=2d(2^{d}-1) - 4(2^{d-1}(d-1-1)+1)$$

$$\vdots$$

$$= cn - 2\lfloor \log n \rfloor - 4, \quad 2 \le c \le 4$$



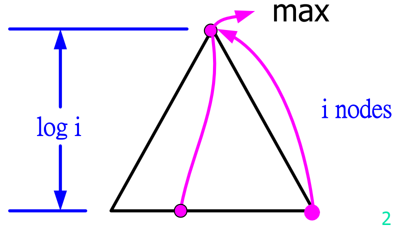
Time complexity
Phase 2: output

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor$$

$$= \quad \vdots$$

$$=2n\lfloor \log n \rfloor - 4cn + 4, \quad 2 \le c \le 4$$

$$=O(n \log n)$$



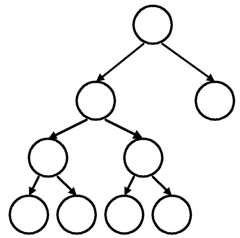
Average case lower bound of sorting

- By binary decision tree
- The average time complexity of a sorting algorithm:

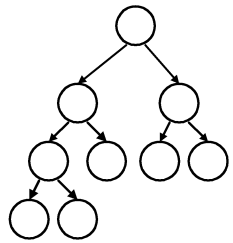
  $$\frac{\text{the external path length of the binary tree}}{n!}$$

- <u>The external path length is minimized if the tree is balanced.</u>

  (all leaf nodes on level d or level d–1)



unbalanced
external path length
= 4·3 + 1 = 13



balanced
external path length
= 2·3+3·2 = 12

Compute the min external path length

1. Depth of balanced binary tree with c leaf nodes:
   $d = \lceil \log c \rceil$
   Leaf nodes can appear only on level d or d–1.
2. x_1 leaf nodes on level d–1
   x_2 leaf nodes on level d
   - $x_1 + x_2 = c$
   - $x_1 + \frac{x_2}{2} = 2^{d-1}$

   $\Rightarrow x_1 = 2^d - c$
   $x_2 = 2(c - 2^{d-1})$

3. External path length:
   $M = x_1(d-1) + x_2 d$
   $= (2^d - c)(d-1) + 2(c - 2^{d-1})d$
   $= c + cd - 2^d, \qquad \log c \le d < \log c + 1$
   $\ge c + c \log c - 2 * 2^{\log c}$
   $= c \log c - c$
4. $c = n!$
   $M = n! \log n! - n!$
   $M/n! = \log n! - 1$
   $= \Omega(n \log n)$

<u>Average case lower bound of sorting: $\Omega(n \log n)$</u>

Quicksort & Heapsort

- Quicksort is optimal in the average case.
  (O(n log n) in average)
- (i)worst case time complexity of heapsort is
  O(n log n)
  (ii)average case lower bound: Ω(n log n)
  - average case time complexity of heapsort is O(n log n)
  - Heapsort is optimal in the average case.

Improving a lower bound through oracles

- Problem P: merge two sorted sequences A and B with lengths m and n.
- Conventional 2-way merging:
  2 3 5 6
  1 4 7 8
- Complexity: at most m+n-1 comparisons

- (1) Binary decision tree:
  There are $\binom{m+n}{n}$ ways !
  $\binom{m+n}{n}$ leaf nodes in the decision tree.
  $\Rightarrow$ The lower bound for merging:
  $\lceil \log \binom{m+n}{n} \rceil \le m + n - 1$
  (conventional merging)

- When m = n

$$\log\binom{m+n}{n} = \log\frac{(2m)!}{(m!)^2} = \log((2m)!) - 2\log m!$$

Using Stirling approximation

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$\log\binom{m+n}{n} \approx (\log\sqrt{2\pi} + \log\sqrt{2m} + 2m\log\frac{2m}{e}) -$$

$$-2\,(\log\sqrt{2\pi} + \log\sqrt{m} + m\log\frac{m}{e})$$

$$\approx 2m - \frac{1}{2}\log m + O(1) \; < \; 2m-1$$

- Optimal algorithm: conventional merging needs 2m-1 comparisons

(2) Oracle:

- The oracle tries its best to cause the algorithm to work as hard as it might. (to give a very hard data set)
- Two sorted sequences:
  - A: $a_1 < a_2 < \ldots < a_m$
  - B: $b_1 < b_2 < \ldots < b_m$
- The very hard case:
  - $a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$

- We must compare:

  $a_1 : b_1$
  $b_1 : a_2$
  $a_2 : b_2$
  $\vdots$
  $b_{m-1} : a_{m-1}$
  $a_m : b_m$

- Otherwise, we may get a wrong result for some input data.

  e.g. If b_1 and a_2 are not compared, we can not distinguish

  $a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$ and

  $a_1 < a_2 < b_1 < b_2 < \ldots < a_m < b_m$

- Thus, at least $2m-1$ comparisons are required.
- The conventional merging algorithm is optimal for $m = n$.

Finding lower bound by problem transformation

- Problem A reduces to problem B ($A \propto B$)
  - iff A can be solved by using any algorithm which solves B.
  - If $A \propto B$, B is more difficult.

$$\begin{array}{ccc} \text{instance of A} & \xrightarrow[T(tr_1)]{\text{transformation}} & \text{instance of B} \\ T(A)\downarrow & & \downarrow T(B) \ \text{solver of B} \\ \text{answer of A} & \xleftarrow[T(tr_2)]{\text{transformation}} & \text{answer of B} \end{array}$$

- Note: $T(tr_1) + T(tr_2) < T(B)$

  $T(A) \le T(tr_1) + T(tr_2) + T(B) \sim O(T(B))$

The lower bound of the convex hull problem

- sorting $\propto$ convex hull

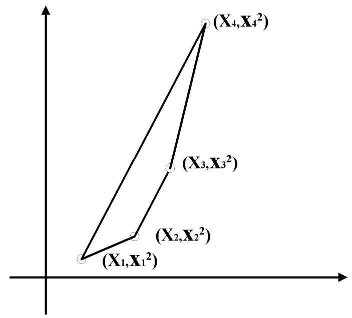
  A B

- an instance of A: $(x_1, x_2, \ldots, x_n)$

  $\downarrow$transformation

  an instance of B: $\{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$

  assume: $x_1 < x_2 < \ldots < x_n$



- If the convex hull problem can be solved, we can also solve the sorting problem.
  - The lower bound of sorting: $\Omega(n \log n)$
- The lower bound of the convex hull problem: $\Omega(n \log n)$

The lower bound of the Euclidean minimal spanning tree (MST) problem

- sorting $\propto$ Euclidean MST

  A B
- an instance of A: $(x_1, x_2, \ldots, x_n)$

  $\downarrow$transformation

  an instance of B: $\{(x_1, 0), (x_2, 0), \ldots, (x_n, 0)\}$
  - Assume $x_1 < x_2 < x_3 < \ldots < x_n$
  - $\Leftrightarrow$there is an edge between $(x_i, 0)$ and $(x_{i+1}, 0)$ in the MST, where $1 \leq i \leq n-1$

- If the Euclidean MST problem can be solved, we can also solve the sorting problem.
  - The lower bound of sorting: $\Omega(n \log n)$
- The lower bound of the Euclidean MST problem: $\Omega(n \log n)$

Chapter 3

The Greedy Method

A simple example

- Problem: Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- Algorithm:

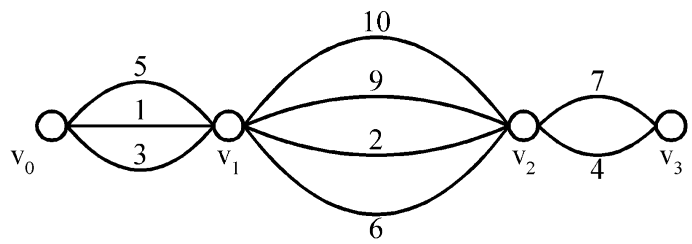
```
FOR i = 1 to k
    pick out the largest number and
    delete this number from the input.
ENDFOR
```


The greedy method

- Suppose that a problem can be solved by a sequence of decisions. The greedy method has that each decision is locally optimal. These locally optimal solutions will finally add up to a globally optimal solution.
- <戰國策.秦策>范睢對秦昭襄王說：「王不如遠交而近攻，得寸，王之寸；得尺，亦王之尺也。」
- Only a few optimization problems can be solved by the greedy method.

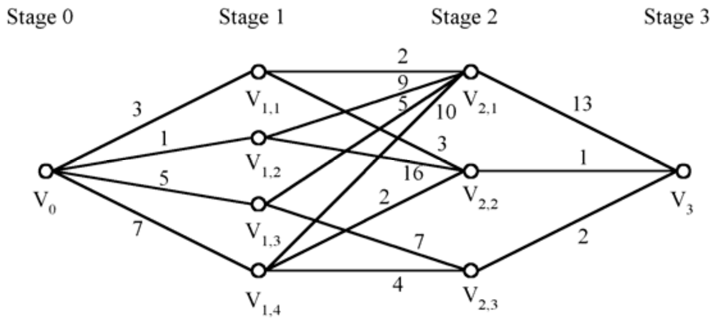
Shortest paths on a special graph

- Problem: Find a shortest path from v_0 to v_3.
- The greedy method can solve this problem.
- The shortest path: 1 + 2 + 4 = 7.



Shortest paths on a multi-stage graph

- Problem: Find a shortest path from v_0 to v_3 in the multi-stage graph.



- Greedy method: $v_0v_{1,2}v_{2,1}v_3 = 23$
- Optimal: $v_0v_{1,1}v_{2,2}v_3 = 7$
- The greedy method does not work.

Solution of the above problem

- $d_{min}(i,j)$: minimum distance between i and j.

$$d_{min}(v_0,v_3)=\min\begin{cases} 3+d_{min}(v_{1,1},v_3) \\ 1+d_{min}(v_{1,2},v_3) \\ 5+d_{min}(v_{1,3},v_3) \\ 7+d_{min}(v_{1,4},v_3) \end{cases}$$

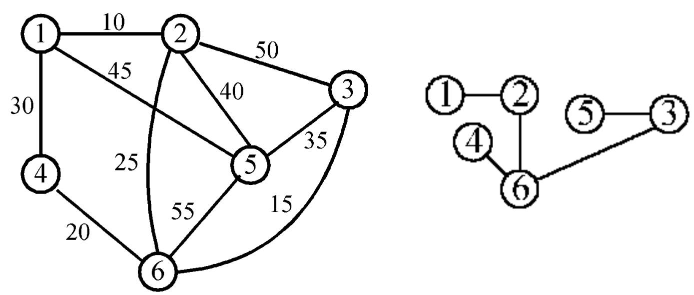
- This problem can be solved by the dynamic programming method.

Minimum spanning trees (MST)

- It may be defined on Euclidean space points or on a graph.
- G = (V, E): weighted connected undirected graph
- Spanning tree : S = (V, T), T $\subseteq$ E, undirected tree
- Minimum spanning tree(MST) : a spanning tree with the smallest total weight.

An example of MST

- A graph and one of its minimum costs spanning tree



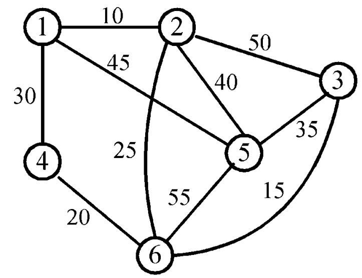
Kruskal's algorithm for finding MST

Step 1: Sort all edges into nondecreasing order.

Step 2: Add the next smallest weight edge to the forest if it will not cause a cycle.

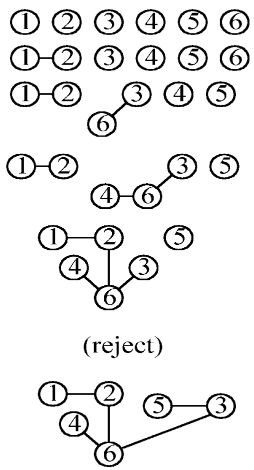
Step 3: Stop if n-1 edges. Otherwise, go to Step2.

An example of Kruskal's algorithm



Edge	Cost
(1,2)	10
(3,6)	15
(4,6)	20
(2,6)	25
(1,4)	30
(3,5)	35

Spanning Forest



The details for constructing MST

- How do we check if a cycle is formed when a new edge is added?
  - By the SET and UNION method.
- Each tree in the spanning forest is represented by a SET.
  - If (u, v) $\in$ E and u, v are in the same set, then the addition of (u, v) will form a cycle.
  - If (u, v) $\in$ E and u$\in S_1$, v$\in S_2$, then perform UNION of S_1 and S_2 .

Time complexity

- Time complexity: O(|E| log|E|)
  - Step 1: O(|E| log|E|)
  - Step 2 & Step 3: $O(|E|\,\alpha(|E|,|V|))$
    Where α is the inverse of Ackermann's function.

Ackermann's function

- $$A(1, j) = 2^j \quad \text{for } j \geq 1$$
  $$A(i,1) = A(i-1,2) \quad \text{for } i \geq 2$$
  $$A(i, j) = A(i-1, A(i, j-1)) \quad \text{for } i, j \geq 2$$

$\Rightarrow$ A(p, q+1) > A(p, q), A(p+1, q) > A(p, q)

$$A(3,4) = \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right\} \text{ 65536 two's}$$

Inverse of Ackermann's function

- $\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log_2 n\}$

  Practically, $A(3,4) > \log_2 n$

  $\Rightarrow \alpha(m, n) \leq 3$

  $\Rightarrow \alpha(m, n)$ is almost a constant.

Prim's algorithm for finding MST

Step 1: $x \in V$, Let $A = \{x\}$, $B = V - \{x\}$.

Step 2: Select $(u, v) \in E$, $u \in A$, $v \in B$ such that (u, v) has the smallest weight between A and B.

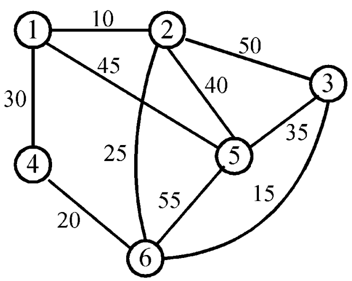
Step 3: Put (u, v) in the tree. $A = A \cup \{v\}$, $B = B - \{v\}$

Step 4: If $B = \varnothing$, stop; otherwise, go to Step 2.

- Time complexity : $O(n^2)$, $n = |V|$.

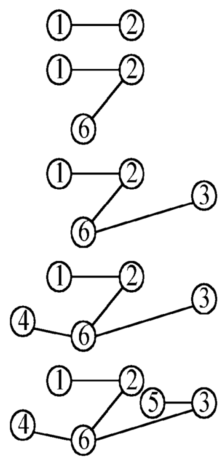
(see the example on the next page)

An example for Prim's algorithm



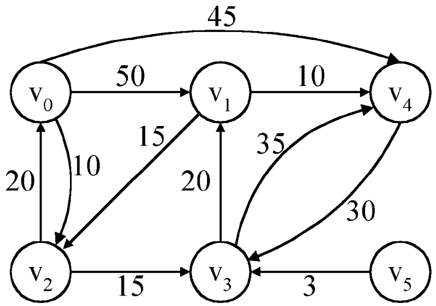
Edge	Cost
(1,2)	10
(2,6)	25
(3,6)	15
(6,4)	20
(3,5)	35

Spanning tree



The single-source shortest path problem

- shortest paths from v_0 to all destinations



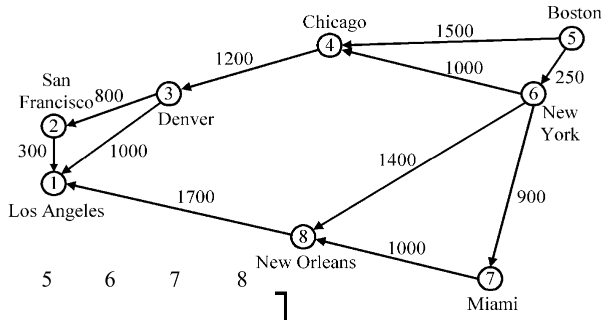
(a)

	Path	Length
1)	v_0v_2	10
2)	$v_0v_2v_3$	25
3)	$v_0v_2v_3v_1$	45
4)	v_0v_4	45

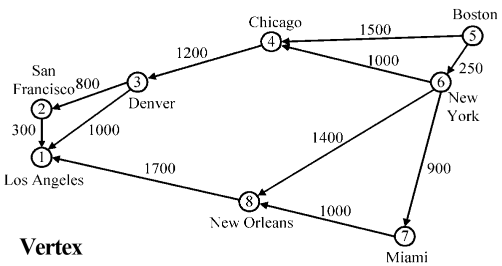
(b)

Dijkstra's algorithm

In the cost adjacency matrix, all entries not shown are $+\infty$.



$$
\begin{array}{c}
 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8
\end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\left[\begin{array}{cccccccc}
0 & & & & & & & \\
300 & 0 & & & & & & \\
1000 & 800 & 0 & & & & & \\
 & & 1200 & 0 & & & & \\
 & & & 1500 & 0 & 250 & & \\
 & & & 1000 & & 0 & 900 & 1400 \\
 & & & & & & 0 & 1000 \\
1700 & & & & & & & 0
\end{array}\right]
\end{array}
$$



Iteration	S	Vertex Selected	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Initial		----								
1	5	6	$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
2	5,6	7	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	5,6,7	4	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
4	5,6,7,4	8	$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
5	5,6,7,4,8	3	3350	$+\infty$	2450	1250	0	250	1150	1650
6	5,6,7,4,8,3	2	3350	3250	2450	1250	0	250	1150	1650
	5,6,7,4,8,3,2		3350	3250	2450	1250	0	250	1150	1650

- Time complexity : $O(n^2)$, n = |V|.

The longest path problem

- Can we use Dijkstra's algorithm to find the longest path from a starting vertex to an ending vertex in an acyclic directed graph?
- There are 3 possible ways to apply Dijkstra's algorithm:
  - Directly use "max" operations instead of "min" operations.
  - Convert all positive weights to be negative. Then find the shortest path.
  - Give a very large positive number M. If the weight of an edge is w, now M-w is used to replace w. Then find the shortest path.
- All these 3 possible ways would not work!

CPM for the longest path problem

- The longest path(critical path) problem can be solved by the critical path method(CPM) :

Step 1:Find a topological ordering.

Step 2: Find the critical path.

(see [Horiwitz 1995].)

  - [[Horowitz 1995] E. Howowitz, S. Sahni and D. Metha, *Fundamentals of Data Structures in C++*, Computer Science Press, New York, 1995

The 2-way merging problem

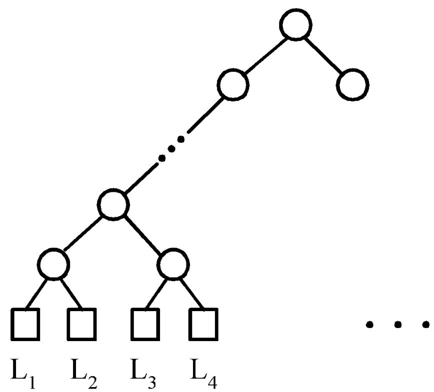
- # of comparisons required for the linear 2-way merge algorithm is $m_1 + m_2 - 1$ where m_1 and m_2 are the lengths of the two sorted lists respectively.
  - 2-way merging example

    2 3 5 6

    1 4 7 8
- The problem: There are n sorted lists, each of length m_i. What is the optimal sequence of merging process to merge these n lists into one sorted list ?

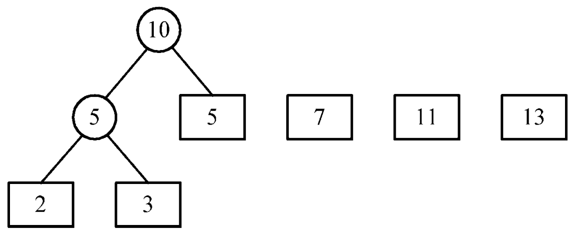
Extended binary trees

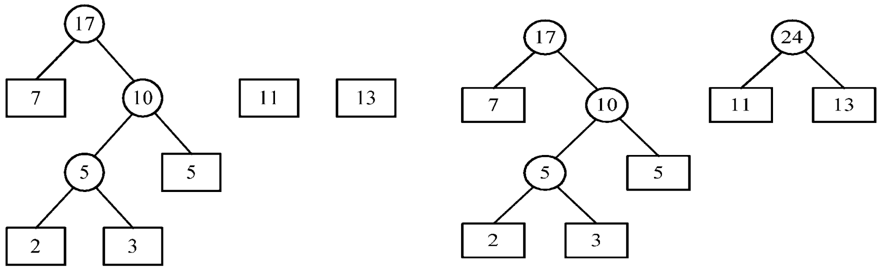
- An extended binary tree representing a 2-way merge

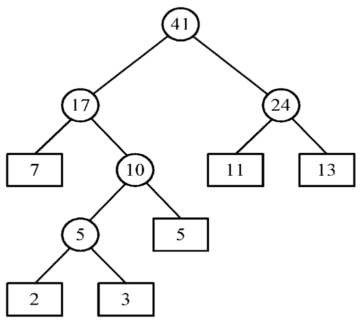


An example of 2-way merging

- Example: 6 sorted lists with lengths 2, 3, 5, 7, 11 and 13.







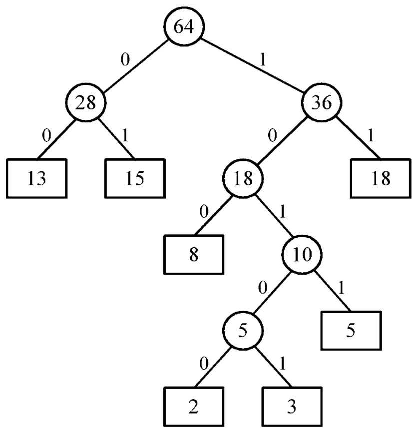
- Time complexity for generating an optimal extended binary tree:O(n log n)

Huffman codes

- In telecommunication, how do we represent a set of messages, each with an access frequency, by a sequence of 0's and 1's?
- To minimize the transmission and decoding costs, we may use short strings to represent more frequently used messages.
- This problem can by solved by using an extended binary tree which is used in the 2-way merging problem.

An example of Huffman algorithm

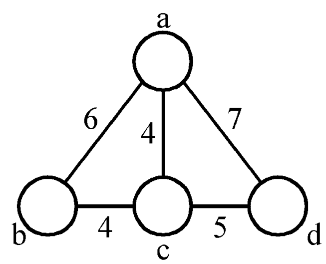
- Symbols: A, B, C, D, E, F, G
  freq. : 2, 3, 5, 8, 13, 15, 18
- Huffman codes:
  A: 10100 B: 10101 C: 1011
  D: 100 E: 00 F: 01
  G: 11



A Huffman code Tree

The minimal cycle basis problem

- 3 cycles:
  A_1 = {ab, bc, ca}
  A_2 = {ac, cd, da}
  A_3 = {ab, bc, cd, da}
  where $A_3 = A_1 \oplus A_2$
  $(A \oplus B = (A \cup B)-(A \cap B))$
  $A_2 = A_1 \oplus A_3$
  $A_1 = A_2 \oplus A_3$
  Cycle basis : {A_1, A_2} or {A_1, A_3} or {A_2, A_3}



- **Def** : A cycle basis of a graph is a set of cycles such that every cycle in the graph can be generated by applying ⊕ on some cycles of this basis.
- Minimal cycle basis : smallest total weight of all edges in this cycle.
- e.g. $\{A_1, A_2\}$

- Algorithm for finding a minimal cycle basis:

  Step 1: Determine the size of the minimal cycle basis, demoted as k.

  Step 2: Find all of the cycles. Sort all cycles(by weight).

  Step 3: Add cycles to the cycle basis one by one. Check if the added cycle is a linear combination of some cycles already existing in the basis. If it is, delete this cycle.

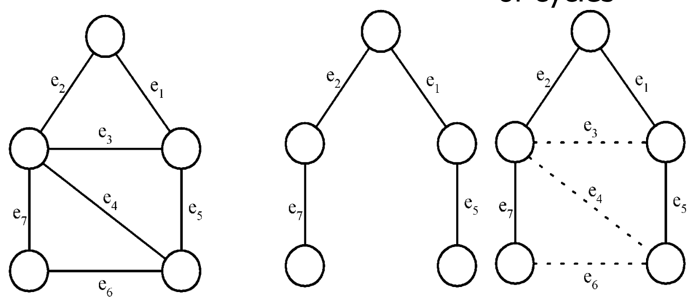
  Step 4: Stop if the cycle basis has k cycles.

Detailed steps for the minimal cycle basis problem

- Step 1 :

  A cycle basis corresponds to the fundamental set of cycles with respect to a spanning tree.

a graph　　a spanning tree　　a fundamental set of cycles



\# of cycles in a cycle basis :

= k

= |E| - (|V|- 1)

= |E| - |V| + 1

- Step 2:

  How to find all cycles in a graph?

  [Reingold, Nievergelt and Deo 1977]

  How many cycles in a graph in the worst case?

  In a complete digraph of n vertices and n(n-1) edges:

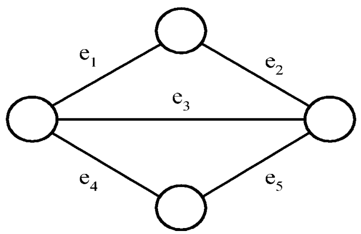
$$\sum_{i=2}^{n} C_i^n (i-1)! > (n-1)!$$

- Step 3:

  How to check if a cycle is a linear combination of some cycles?

  Using Gaussian elimination.

Gaussian elimination



- 2 cycles C_1 and C_2 are represented by a 0/1 matrix

$$\begin{array}{c} \\ C_1 \\ C_2 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \end{bmatrix} \end{array}$$

- Add C_3

$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \\ 1 & 1 & & 1 & 1 \end{bmatrix} \end{array}$$

- $\oplus$ on rows 1 and 3

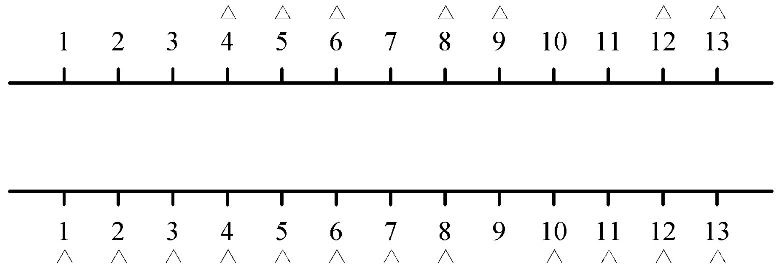
$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \\ & & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$\oplus$ on rows 2 and 3 : empty

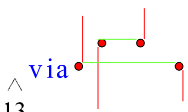
$\because C_3 = C_1 \oplus C_2$

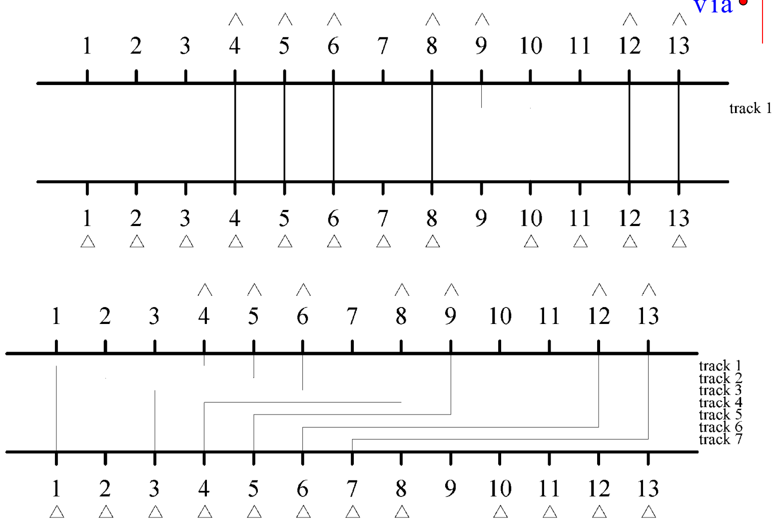
The 2-terminal one to any special channel routing problem

- Def: Given two sets of terminals on the upper and lower rows, respectively, we have to connect each upper terminal to the lower row in a one to one fashion. This connection requires that # of tracks used is minimized.



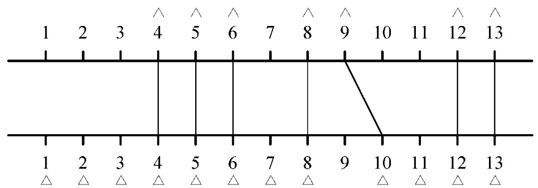
2 feasible solutions



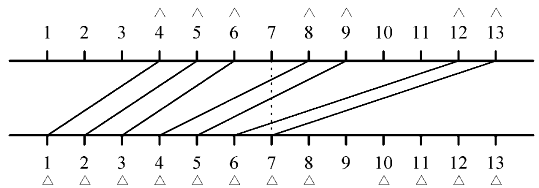


Redrawing solutions

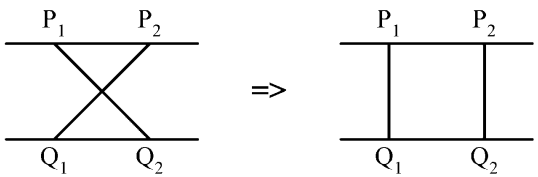
(a) Optimal solution



(b) Another solution



- At each point, the local density of the solution is # of lines the vertical line intersects.
- The problem: to minimize the density. The density is a lower bound of # of tracks.
- Upper row terminals: P_1 ,P_2 ,…, P_n from left to right
- Lower row terminals: Q_1 ,Q_2 ,…, Q_m from left to right m > n.
- It would never have a crossing connection:



- Suppose that we have a method to determine the minimum density, d, of a problem instance.
- The greedy algorithm:

Step 1 : P_1 is connected Q_1.

Step 2 : After P_i is connected to Q_j, we check whether P_{i+1} can be connected to Q_{j+1}. If the density is increased to d+1, try to connect P_{i+1} to Q_{j+2}.

Step 3 : Repeat Step2 until all P_i's are connected.

The knapsack problem

- n objects, each with a weight $w_i > 0$
  a profit $p_i > 0$
  capacity of knapsack: M

Maximize $\sum_{1\le i\le n} p_i x_i$

Subject to $\sum_{1\le i\le n} w_i x_i \le M$

$0 \le x_i \le 1,\ 1 \le i \le n$

The knapsack algorithm

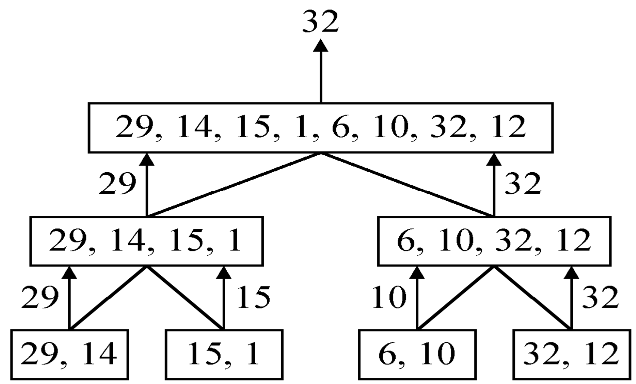
- The greedy algorithm:
  Step 1: Sort p_i/w_i into nonincreasing order.
  Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.
- e. g.
  n = 3, M = 20, $(p_1, p_2, p_3) = (25, 24, 15)$
  $(w_1, w_2, w_3) = (18, 15, 10)$
  Sol: $p_1/w_1 = 25/18 = 1.39$
  $p_2/w_2 = 24/15 = 1.6$
  $p_3/w_3 = 15/10 = 1.5$
  Optimal solution: $x_1 = 0$, $x_2 = 1$, $x_3 = 1/2$

Chapter 4

The Divide-and-Conquer Strategy

A simple example

- finding the maximum of a set S of n numbers



Time complexity

- Time complexity:

$$T(n)=\begin{cases} 2T(n/2)+1 & , n>2 \\ 1 & , n\le 2 \end{cases}$$

- Calculation of T(n):

Assume $n = 2^k$,

$$\begin{aligned} T(n) &= 2T(n/2)+1 \\ &= 2(2T(n/4)+1)+1 \\ &= 4T(n/4)+2+1 \\ &\quad\vdots \\ &=2^{k-1}T(2)+2^{k-2}+\ldots+4+2+1 \\ &=2^{k-1}+2^{k-2}+\ldots+4+2+1 \\ &=2^k-1 = n-1 \end{aligned}$$

諫逐客書—李斯

- 文選自 〈史記. 李斯列傳〉，是李斯上呈秦王政的一篇奏疏。
- 「惠王用張儀之計，拔三川之地，西并巴、蜀，北收上郡，南取漢中，包九夷，制鄢(ㄧㄢ)、郢(ㄧㄥˇ)，東據成皋之險，割膏腴之壤，遂散六國之從(縱)，使之西面事秦，功施(ㄧˋ)到今。」
- 註：秦滅六國順序：韓、趙、魏、楚、燕、齊

A general divide-and-conquer algorithm

Step 1: If the problem size is small, solve this problem directly; otherwise, split the original problem into 2 sub-problems with equal sizes.

Step 2: Recursively solve these 2 sub-problems by applying this algorithm.

Step 3: Merge the solutions of the 2 sub-problems into a solution of the original problem.

Time complexity of the general algorithm

- Time complexity:

$$T(n)=\begin{cases} 2T(n/2)+S(n)+M(n) & , n \ge c \\ b & , n < c \end{cases}$$

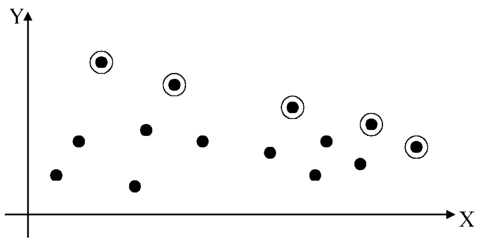
where S(n) : time for splitting
M(n) : time for merging
b : a constant
c : a constant

- e.g. Binary search
- e.g. quick sort
- e.g. merge sort e.g. 2 6 5 3 7 4 8 1

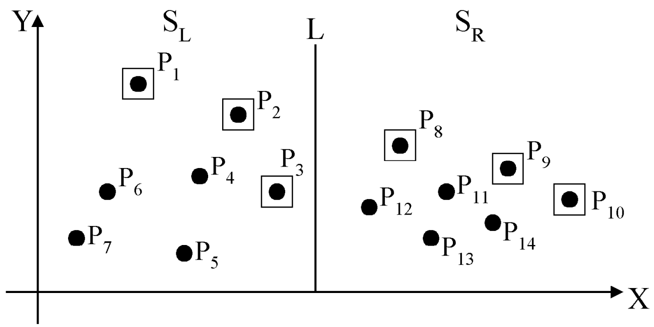
2-D maxima finding problem

- **Def** : A point (x_1, y_1) dominates (x_2, y_2) if $x_1 > x_2$ and $y_1 > y_2$. A point is called a maximum if no other point dominates it
- Straightforward method : Compare every pair of points.

Time complexity: $O(n^2)$



Divide-and-conquer for maxima finding



The maximal points of S_L and S_R

The algorithm:

- Input: A set S of n planar points.
- Output: The maximal points of S.

Step 1: If S contains only one point, return it as the maximum. Otherwise, find a line L perpendicular to the X-axis which separates S into S_L and S_R, with equal sizes.

Step 2: Recursively find the maximal points of S_L and S_R.

Step 3: Find the largest y-value of S_R, denoted as y_R. Discard each of the maximal points of S_L if its y-value is less than or equal to y_R.

- Time complexity: T(n)

  Step 1: O(n)

  Step 2: 2T(n/2)

  Step 3: O(n)

$$T(n)=\begin{cases} 2T(n/2)+O(n)+O(n) & , n>1 \\ 1 & , n=1 \end{cases}$$

Assume $n = 2^k$

$T(n) = O(n \log n)$

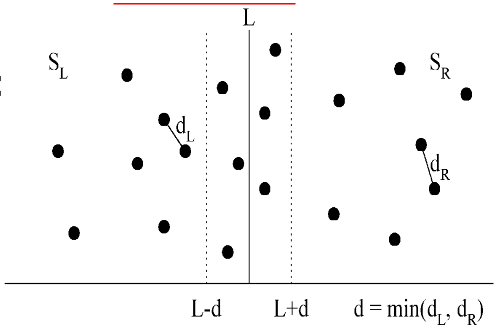
The closest pair problem

- Given a set S of n points, find a pair of points which are closest together.
- 1-D version :

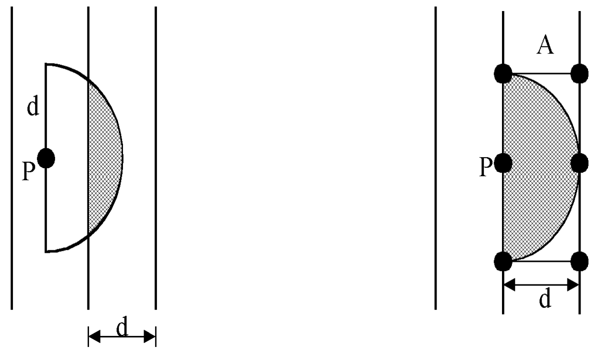
  Solved by sorting

  Time complexity :

  O(n log n)
- 2-D version



- at most 6 points in area A:



The algorithm:

- Input: A set S of n planar points.
- Output: The distance between two closest points.

Step 1: Sort points in S according to their y-values.

Step 2: If S contains only one point, return infinity as its distance.

Step 3: Find a median line L perpendicular to the X-axis to divide S into S_L and S_R, with equal sizes.

Step 4: Recursively apply Steps 2 and 3 to solve the closest pair problems of S_L and S_R. Let $d_L(d_R)$ denote the distance between the closest pair in S_L (S_R). Let d = min(d_L, d_R).

Step 5: For a point P in the half-slab bounded by L-d and L, let its y-value be denoted as y_P. For each such P, find all points in the half-slab bounded by L and L+d whose y-value fall within y_P+d and y_P-d. If the distance d′ between P and a point in the other half-slab is less than d, let d=d′. The final value of d is the answer.

- Time complexity: O(n log n)

  Step 1: O(n log n)

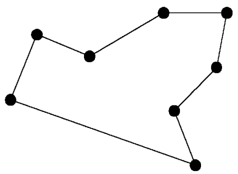
  Steps 2~5:

$$T(n)=\begin{cases} 2T(n/2)+O(n)+O(n) & , n>1 \\ 1 & , n=1 \end{cases}$$

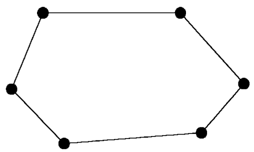
$\Rightarrow$T(n) = O(n log n)

The convex hull problem

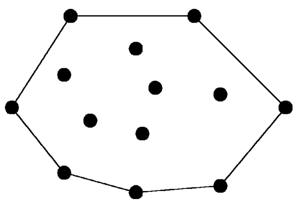
concave polygon:



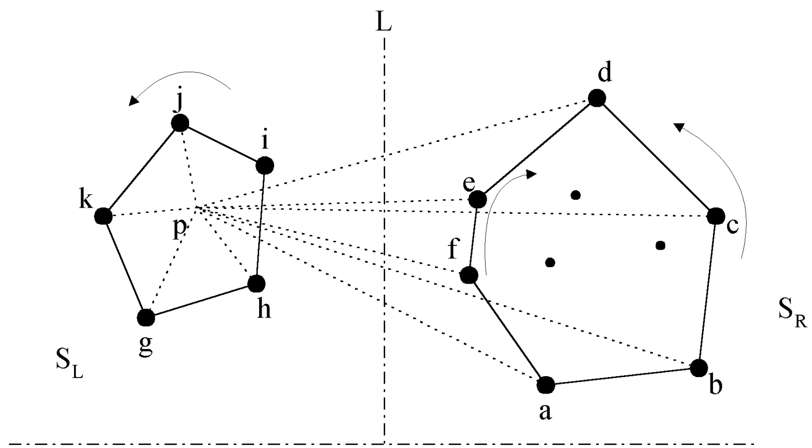
convex polygon:



- The convex hull of a set of planar points is the smallest convex polygon containing all of the points.



- The divide-and-conquer strategy to solve the problem:

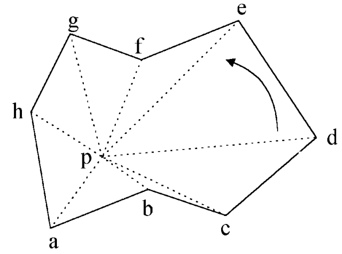


- The merging procedure:

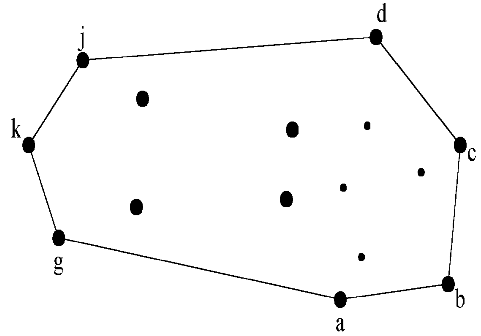
1. Select an interior point p.
2. There are 3 sequences of points which have increasing polar angles with respect to p.
   (1) g, h, i, j, k
   (2) a, b, c, d
   (3) f, e
3. Merge these 3 sequences into 1 sequence:
   g, h, a, b, f, c, e, d, i, j, k.
4. Apply Graham scan to examine the points one by one and eliminate the points which cause reflexive angles.

(See the example on the next page.)

- e.g. points b and f need to be deleted.



Final result:



Divide-and-conquer for convex hull

- Input : A set S of planar points
- Output : A convex hull for S

Step 1: If S contains no more than five points, use exhaustive searching to find the convex hull and return.

Step 2: Find a median line perpendicular to the X-axis which divides S into S_L and S_R, with equal sizes.

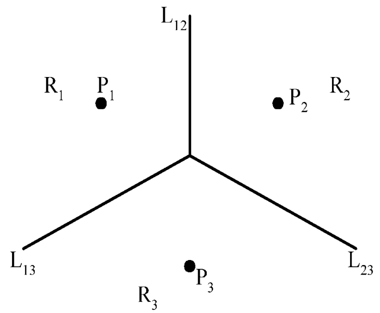
Step 3: Recursively construct convex hulls for S_L and S_R, denoted as Hull(S_L) and Hull(S_R), respectively.

- Step 4: Apply the merging procedure to merge Hull(S_L) and Hull(S_R) together to form a convex hull.
- Time complexity:

$$T(n) = 2T(n/2) + O(n)$$
$$= O(n \log n)$$

The Voronoi diagram problem

- e.g. The Voronoi diagram for three points

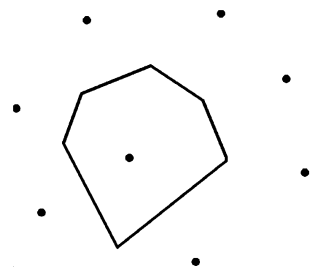


Each L_{ij} is the perpendicular bisector of line segment $\overline{P_iP_j}$. The intersection of three L_{ij}'s is the circumcenter (外心) of triangle $P_1P_2P_3$.

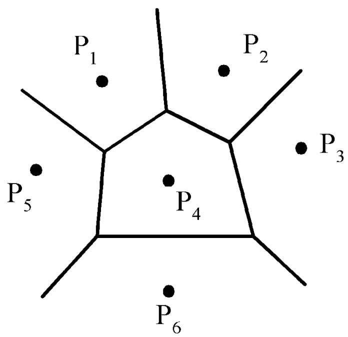
Definition of Voronoi diagrams

- **Def** : Given two points P_i, $P_j \in S$, let $H(P_i,P_j)$ denote the half plane containing P_i. The Voronoi polygon associated with P_i is defined as

$$V(i) = \bigcap_{i \neq j} H(P_i, P_j)$$

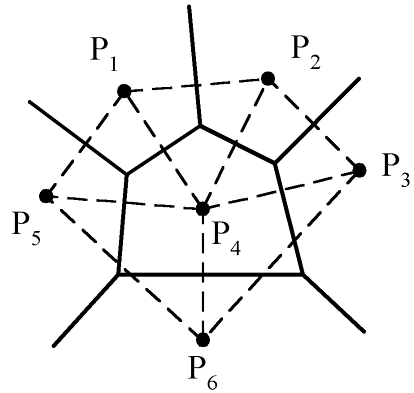


- Given a set of n points, the Voronoi diagram consists of all the Voronoi polygons of these points.

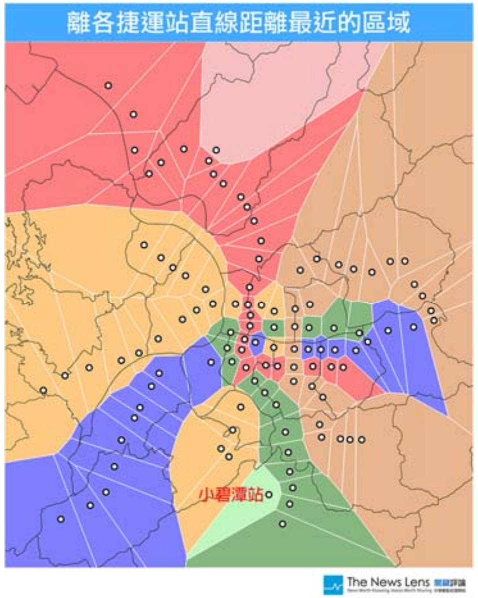


- The vertices of the Voronoi diagram are called Voronoi points and its segments are called Voronoi edges.

Delaunay triangulation



臺北捷運站涵蓋區域圖



Example for constructing Voronoi diagrams

- Divide the points into two parts.

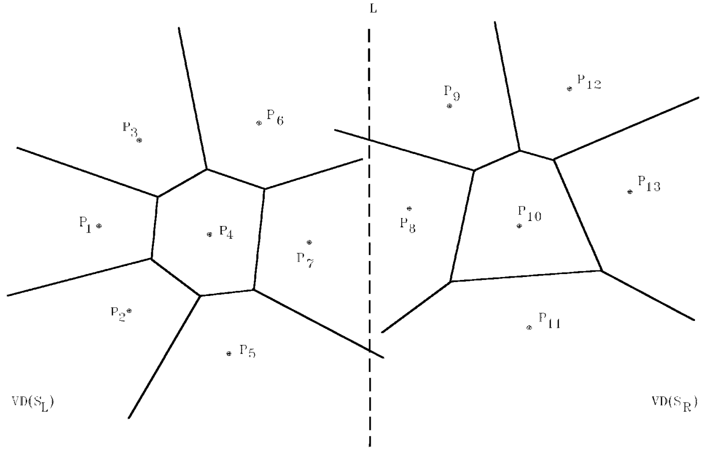


Fig. 5-17: Two Voronoi Diagrams After Step 2

Merging two Voronoi diagrams

- Merging along the piecewise linear hyperplane

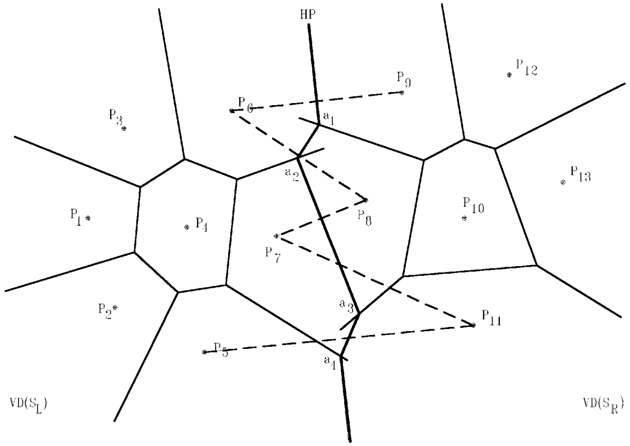


Fig. 5-18: The Piecewise Linear Hyperplane for the set of Points Shown in Fig. 5-17.

The final Voronoi diagram

- After merging

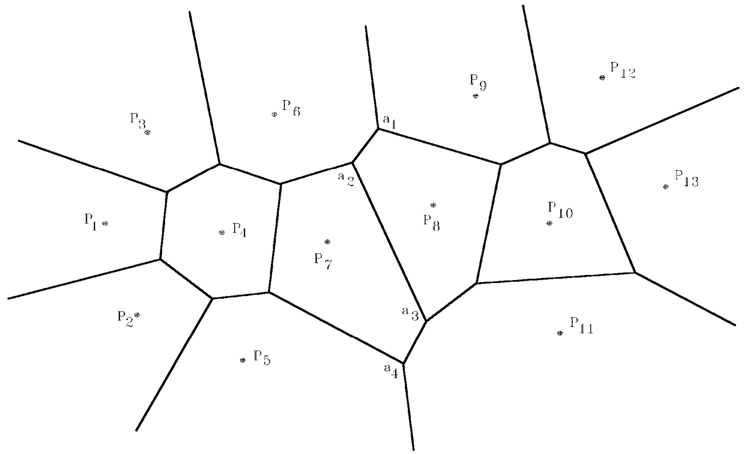


Fig. 5-19: The Voronoi Diagram of the Points in Fig. 5-17.

Divide-and-conquer for Voronoi diagram

- Input: A set S of n planar points.
- Output: The Voronoi diagram of S.

Step 1: If S contains only one point, return.

Step 2: Find a median line L perpendicular to the X-axis which divides S into S_L and S_R, with equal sizes.

Step 3: Construct Voronoi diagrams of S_L and S_R recursively. Denote these Voronoi diagrams by VD(S_L) and VD(S_R).

Step 4: Construct a dividing piece-wise linear hyperplane HP which is the locus of points simultaneously closest to a point in S_L and a point in S_R. Discard all segments of VD(S_L) which lie to the right of HP and all segments of VD(S_R) that lie to the left of HP. The resulting graph is the Voronoi diagram of S.

(See details on the next page.)

Mergeing Two Voronoi Diagrams into One Voronoi Diagram

- Input: (a) S_L and S_R where S_L and S_R are divided by a perpendicular line L.
  (b) VD(S_L) and VD(S_R).
- Output: VD(S) where S = $S_L \cap S_R$

Step 1: Find the convex hulls of S_L and S_R, denoted as Hull(S_L) and Hull(S_R), respectively. (A special algorithm for finding a convex hull in this case will by given later.)

Step 2: Find segments $\overline{P_aP_b}$ and $\overline{P_cP_d}$ which join HULL(S_L) and HULL(S_R) into a convex hull (P_a and P_c belong to S_L and P_b and P_d belong to S_R) Assume that $\overline{P_aP_b}$ lies above $\overline{P_cP_d}$. Let x = a, y = b, SG= $\overline{P_xP_y}$ and HP = $\varnothing$.

Step 3: Find the perpendicular bisector of SG. Denote it by BS. Let HP = HP$\cup${BS}. If SG = $\overline{P_cP_d}$, go to Step 5; otherwise, go to Step 4.

Step 4: The ray from VD(S_L) and VD(S_R) which BS first intersects with must be a perpendicular bisector of either $\overline{P_xP_z}$ or $\overline{P_yP_z}$ for some z. If this ray is the perpendicular bisector of $\overline{P_yP_z}$, then let SG = $\overline{P_xP_z}$; otherwise, let SG = $\overline{P_zP_y}$. Go to Step 3.

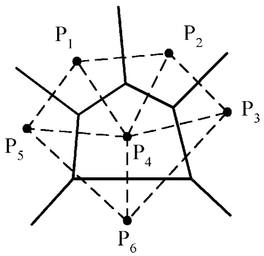
Step 5: Discard the edges of VD(S_L) which extend to the right of HP and discard the edges of VD(S_R) which extend to the left of HP. The resulting graph is the Voronoi diagram of S = $S_L \cup S_R$.

Properties of Voronoi Diagrams

- **Def :** Given a point P and a set S of points, the distance between P and S is the distance between P and P_i which is the nearest neighbor of P in S.
- The HP obtained from the above algorithm is the locus of points which keep equal distances to S_L and S_R .
- The HP is monotonic in y.

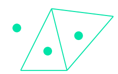
of Voronoi edges

- # of edges of a Voronoi diagram $\leq$ 3n - 6, where n is # of points.
- Reasoning:
  i. # of edges of a planar graph with n vertices $\leq$ 3n - 6.
  ii. A Delaunay triangulation is a planar graph.
  iii. Edges in Delaunay triangulation $\xleftrightarrow{1-1}$ edges in Voronoi diagram.



of Voronoi vertices

- # of Voronoi vertices $\leq$ 2n - 4.
- Reasoning:
  i. Let F, E and V denote # of face, edges and vertices in a planar graph.
     Euler's relation: F = E - V + 2.
  ii. In a Delaunay triangulation,
     triangle $\xleftrightarrow{1-1}$ Voronoi vertex
     V = n, E $\leq$ 3n – 6
     $\Rightarrow$ F = E - V + 2 $\leq$ 3n - 6 - n + 2 = 2n - 4.



Construct a convex hull from a Voronoi diagram

- After a Voronoi diagram is constructed, a convex hull can by found in O(n) time.

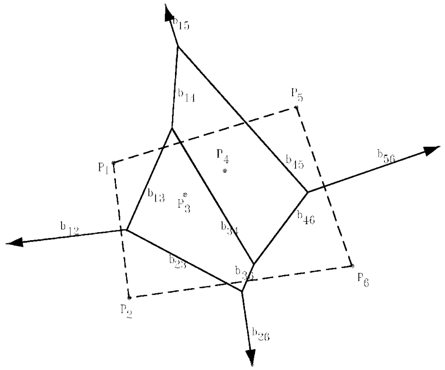


Fig. 5-25: Constructing a Convex Hull from a Voronoi Diagram

Construct a convex hull from a Voronoi diagram

Step 1: Find an infinite ray by examining all Voronoi edges.

Step 2: Let P_i be the point to the left of the infinite ray. P_i is a convex hull vertex. Examine the Voronoi polygon of P_i to find the next infinite ray.

Step 3: Repeat Step 2 until we return to the starting ray.

Time complexity

- Time complexity for merging 2 Voronoi diagrams:

  Total: O(n)
  - Step 1: O(n)
  - Step 2: O(n)
  - Step 3 ~ Step 5: O(n)

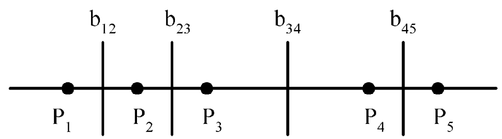
  (at most 3n - 6 edges in VD(S_L) and VD(S_R) and at most n segments in HP)
- Time complexity for constructing a Voronoi diagram: O(n log n)

  because T(n) = 2T(n/2) + O(n)=O(n log n)

Lower bound

- The lower bound of the Voronoi diagram problem is Ω(n log n).

  sorting $\propto$ Voronoi diagram problem



The Voronoi diagram for a set of points on a straight line

Applications of Voronoi diagrams

- The Euclidean nearest neighbor searching problem.
- The Euclidean all nearest neighbor problem.

Fast Fourier transform (FFT)

- Fourier transform

$$b(f) = \int_{-\infty}^{\infty} a(t)e^{i2\pi ft}\,dt, \text{ where } i = \sqrt{-1}$$

- Inverse Fourier transform

$$a(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} b(f)e^{-i2\pi ft}\,dt$$

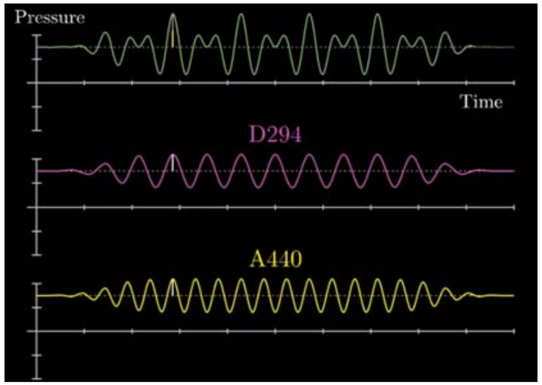
- Discrete Fourier transform(DFT)

Given $a_0, a_1, \ldots, a_{n-1}$, compute

$$\begin{aligned} b_j &= \sum_{k=0}^{n-1} a_k e^{i2\pi jk/n}, 0 \le j \le n-1 \\ &= \sum_{k=0}^{n-1} a_k \omega^{kj}, \text{ where } \omega = e^{i2\pi/n} \end{aligned}$$

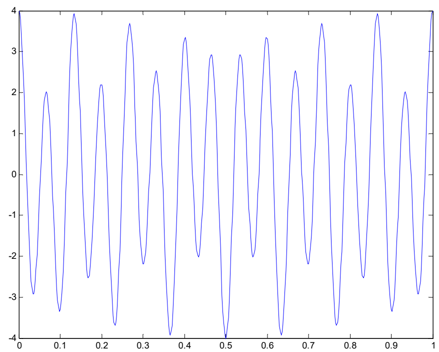
DFT and waveform(1)

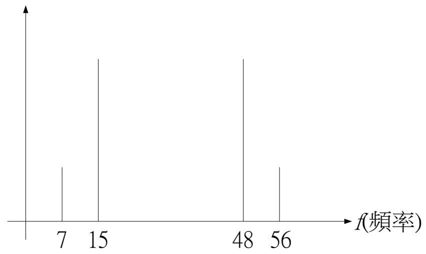
- Any periodic waveform can be decomposed into the linear sum of sinusoid functions (sine or cosine).



DFT and waveform(2)

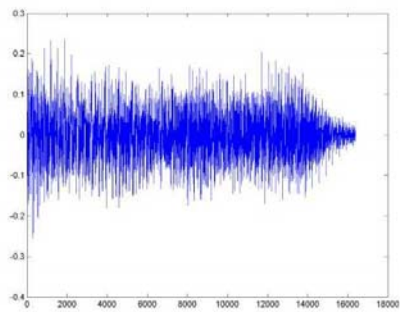
- Any periodic waveform can be decomposed into the linear sum of sinusoid functions (sine or cosine).



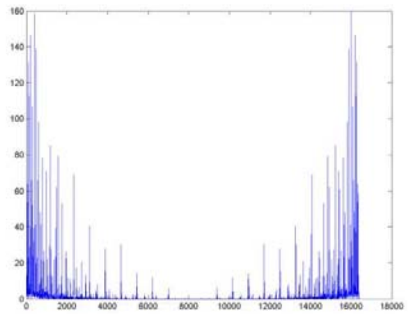


$$f(t) = \cos(2\pi(7)t) + 3\cos(2\pi(15)t) + 3\cos(2\pi(48)t) + \cos(2\pi(56)t)$$

DFT and waveform (3)



The waveform of a music signal of 1 second



The frequency spectrum of the music signal with DFT

An application of the FFT — polynomial multiplication

- Polynomial multiplication:

$$f(x)=\sum_{j=0}^{n-1} a_j x^j, \quad g(x)=\sum_{k=0}^{n-1} c_k x^k \qquad h(x)=f(x)\bullet g(x)$$

- The straightforward product requires O(n^2) time.
- DFT notations:

$$f(x)=a_0+a_1x+a_2x^2+\ldots+a_{n-1}x^{n-1}$$

$$\text{Let } b_j=f\left(w^j\right),\ 0\le j\le n-1,\ w^n=1$$

$$\{b_0,b_1,\cdots,b_{n-1}\} \text{ is the DFT of } \{a_0,a_1,\cdots,a_{n-1}\}.$$

$$h(x)=b_0+b_1x+b_2x^2+\ldots+b_{n-1}x^{n-1}$$

$$a_k=\tfrac{1}{n}h\left(w^{-k}\right),\quad 0\le j\le n-1$$

$$\{a_0,a_1,\cdots,a_{n-1}\} \text{ is the inverse DFT of } \{b_0,b_1,\cdots,b_{n-1}\}.$$

Fast polynomial multiplication

Step 1: Let N be the smallest integer that N=2^q and N≥2n-1.

Step 2: Compute FFT of $\underbrace{\{a_0,a_1,\cdots,a_{n-1},0,0,\cdots,0\}}_{N}.$

Step 3: Compute FFT of $\underbrace{\{c_0,c_1,\cdots,c_{n-1},0,0,\cdots,0\}}_{N}.$

Step 4: Compute $f\left(w^j\right)\bullet g(w^j),\ \ 0\le j\le N-1,\ w=e^{2\pi i/N}$

Step 5: Let $h(w^j)=f\left(w^j\right)\bullet g(w^j)$

Compute inverse DFT of $\{h(w^0),h(w^1),\cdots,h(w^{N-1})\}$.

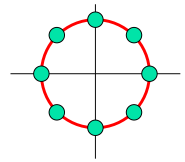
The resulting sequence of numbers are the coefficients of $h(x)$.

- Time complexity: O(NlogN)=O(nlogn), N<4n.

FFT algorithm

- Inverse DFT:

$$a_k=\frac{1}{n}\sum_{j=0}^{n-1} b_j\omega^{-jk},\ 0\le k\le n-1$$



- $$e^{i\theta}=\cos\theta+i\sin\theta$$

$$\omega^n=(e^{i2\pi/n})^n=e^{i2\pi}=\cos 2\pi+i\sin 2\pi=1$$

$$\omega^{n/2}=(e^{i2\pi/n})^{n/2}=e^{i\pi}=\cos\pi+i\sin\pi=-1$$

- DFT can be computed in O(n^2) time by a straightforward method.
- DFT can be solved by the divide-and-conquer strategy (FFT) in O($n$log$n$) time.

FFT algorithm when n=4

- n=4, $w=e^{i2\pi/4}$, w^4=1, w^2=-1
  $b_0=a_0+a_1+a_2+a_3$
  $b_1=a_0+a_1w+a_2w^2+a_3w^3$
  $b_2=a_0+a_1w^2+a_2w^4+a_3w^6$
  $b_3=a_0+a_1w^3+a_2w^6+a_3w^9$

$$b_j=\sum_{k=0}^{n-1}a_k e^{i2\pi jk/n}$$
$$=\sum_{k=0}^{n-1}a_k\omega^{kj}$$

- another form:
  $b_0=(a_0+a_2)+(a_1+a_3)$
  $b_2=(a_0+a_2w^4)+(a_1w^2+a_3w^6)=(a_0+a_2)-(a_1+a_3)$
- When we calculate b_0, we shall calculate (a_0+a_2) and (a_1+a_3). Later, b_2 van be easily calculated.
- Similarly,
  $b_1=(a_0+ a_2w^2)+(a_1w+a_3w^3)=(a_0-a_2)+w(a_1-a_3)$
  $b_3=(a_0+a_2w^6)+(a_1w^3+a_3w^9)=(a_0-a_2)-w(a_1-a_3)$.

FFT algorithm when n=8

$$b_j=\sum_{k=0}^{n-1}a_k e^{i2\pi jk/n}=\sum_{k=0}^{n-1}a_k\omega^{kj}$$

- n=8, $w=e^{i2\pi/8}$, w^8=1, w^4=-1

$b_0=a_0+a_1+a_2+a_3+a_4+a_5+a_6+a_7$
$b_1=a_0+a_1w+a_2w^2+a_3w^3+a_4w^4+a_5w^5+a_6w^6+a_7w^7$
$b_2=a_0+a_1w^2+a_2w^4+a_3w^6+a_4w^8+a_5w^{10}+a_6w^{12}+a_7w^{14}$
$b_3=a_0+a_1w^3+a_2w^6+a_3w^9+a_4w^{12}+a_5w^{15}+a_6w^{18}+a_7w^{21}$
$b_4=a_0+a_1w^4+a_2w^8+a_3w^{12}+a_4w^{16}+a_5w^{20}+a_6w^{24}+a_7w^{28}$
$b_5=a_0+a_1w^5+a_2w^{10}+a_3w^{15}+a_4w^{20}+a_5w^{25}+a_6w^{30}+a_7w^{35}$
$b_6=a_0+a_1w^6+a_2w^{12}+a_3w^{18}+a_4w^{24}+a_5w^{30}+a_6w^{36}+a_7w^{42}$
$b_7=a_0+a_1w^7+a_2w^{14}+a_3w^{21}+a_4w^{28}+a_5w^{35}+a_6w^{42}+a_7w^{49}$

- After reordering, we have

$b_0=(a_0+a_2+a_4+a_6)+(a_1+a_3+a_5+a_7)$
$b_1=(a_0+a_2w^2+a_4w^4+a_6w^6)+ w(a_1+a_3w^2+a_5w^4+a_7w^6)$
$b_2=(a_0+a_2w^4+a_4w^8+a_6w^{12})+ w^2(a_1+a_3w^4+a_5w^8+a_7w^{12})$
$b_3=(a_0+a_2w^6+a_4w^{12}+a_6w^{18})+ w^3(a_1+a_3w^6+a_5w^{12}+a_7w^{18})$
$b_4=(a_0+a_2+a_4+a_6)-(a_1+a_3+a_5+a_7)$
$b_5=(a_0+a_2w^2+a_4w^4+a_6w^6)-w(a_1+a_3w^2+a_5w^4+a_7w^6)$
$b_6=(a_0+a_2w^4+a_4w^8+a_6w^{12})-w^2(a_1+a_3w^4+a_5w^8+a_7w^{12})$
$b_7=(a_0+a_2w^6+a_4w^{12}+a_6w^{18})-w^3(a_1+a_3w^6+a_5w^{12}+a_7w^{18})$

- Rewrite as

$b_0=c_0+d_0$	$b_4=c_0-d_0=c_0+w^4d_0$
$b_1=c_1+wd_1$	$b_5=c_1-wd_1=c_1+w^5d_1$
$b_2=c_2+w^2d_2$	$b_6=c_2-w^2d_2=c_2+w^6d_2$
$b_3=c_3+w^3d_3$	$b_7=c_3-w^3d_3=c_3+w^7d_3$

- $c_0=a_0+a_2+a_4+a_6$
  $c_1=a_0+a_2w^2+a_4w^4+a_6w^6$
  $c_2=a_0+a_2w^4+a_4w^8+a_6w^{12}$
  $c_3=a_0+a_2w^6+a_4w^{12}+a_6w^{18}$
- Let $x=w^2=e^{i2\pi/4}$
  $c_0=a_0+a_2+a_4+a_6$
  $c_1=a_0+a_2x+a_4x^2+a_6x^3$
  $c_2=a_0+a_2x^2+a_4x^4+a_6x^6$
  $c_3=a_0+a_2x^3+a_4x^6+a_6x^9$
- Thus, $\{c_0,c_1,c_2,c_3\}$ is FFT of $\{a_0,a_2,a_4,a_6\}$.
  Similarly, $\{d_0,d_1,d_2,d_3\}$ is FFT of $\{a_1,a_3,a_5,a_7\}$.

General FFT

- In general, let $w=e^{2\pi/n}$ (assume n is even.)

$$w^n=1,\ w^{n/2}=-1$$

$$\begin{aligned}
b_j &= a_0+a_1w^j+a_2w^{2j}+\ldots+a_{n-1}w^{(n-1)j},\\
&= \{a_0+a_2w^{2j}+a_4w^{4j}+\ldots+a_{n-2}w^{(n-2)j}\}+\\
&\quad w^j\{a_1+a_3w^{2j}+a_5w^{4j}+\ldots+a_{n-1}w^{(n-2)j}\}\\
&= c_j+w^jd_j\\
b_{j+n/2} &= a_0+a_1w^{j+n/2}+a_2w^{2j+n}+a_3w^{3j+3n/2}+\ldots\\
&\quad +a_{n-1}w^{(n-1)j+n(n-1)/2}\\
&= a_0-a_1w^j+a_2w^{2j}-a_3w^{3j}+\ldots+a_{n-2}w^{(n-2)j}-a_{n-1}w^{(n-1)j}\\
&= c_j-w^jd_j\\
&= c_j+w^{j+n/2}d_j
\end{aligned}$$

Divide-and-conquer (FFT)

- Input: $a_0, a_1, \ldots, a_{n-1}$, $n = 2^k$
- Output: b_j, $j=0, 1, 2, \ldots, n-1$

  where $b_j = \sum_{0\le k\le n-1} a_k w^{kj}$, where $w = e^{i2\pi/n}$

Step 1: If n=2, compute

$b_0 = a_0 + a_1$,

$b_1 = a_0 - a_1$, and return.

Step 2: Recursively find the Fourier transform of $\{a_0, a_2, a_4, \ldots, a_{n-2}\}$ and $\{a_1, a_3, a_5, \ldots, a_{n-1}\}$, whose results are denoted as $\{c_0, c_1, c_2, \ldots, c_{n/2-1}\}$ and $\{d_0, d_1, d_2, \ldots, d_{n/2-1}\}$.

Step 3: Compute b_j:

$b_j = c_j + w^jd_j$ for $0 \le j \le n/2 - 1$

$b_{j+n/2} = c_j - w^jd_j$ for $0 \le j \le n/2 - 1$.

- Time complexity:

$T(n) = 2T(n/2) + O(n)$

$= O(n \log n)$

Matrix multiplication

- Let A, B and C be $n \times n$ matrices

$C = AB$

$C(i, j) = \sum_{1\le k\le n} A(i, k)B(k, j)$

- The straightforward method to perform a matrix multiplication requires $O(n^3)$ time.

Divide-and-conquer approach

- C = AB

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$
$C_{12} = A_{11}B_{12} + A_{12} B_{22}$
$C_{21} = A_{21} B_{11} + A_{22} B_{21}$
$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

- Time complexity:

$$T(n) = \begin{cases} b & , n \le 2 \\ 8T(n/2)+cn^2 & , n > 2 \end{cases}$$

(# of additions : n^2)

We get $T(n) = O(n^3)$

Strassen's matrix multiplicaiton

- $P = (A_{11} + A_{22})(B_{11} + B_{22})$
  $Q = (A_{21} + A_{22})B_{11}$
  $R = A_{11}(B_{12} - B_{22})$
  $S = A_{22}(B_{21} - B_{11})$
  $T = (A_{11} + A_{12})B_{22}$
  $U = (A_{21} - A_{11})(B_{11} + B_{12})$
  $V = (A_{12} - A_{22})(B_{21} + B_{22})$.
- $C_{11} = P + S - T + V$
  $C_{12} = R + T$
  $C_{21} = Q + S$
  $C_{22} = P + R - Q + U$

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$
$C_{12} = A_{11}B_{12} + A_{12} B_{22}$
$C_{21} = A_{21} B_{11} + A_{22} B_{21}$
$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

Time complexity

- 7 multiplications and 18 additions or subtractions
- Time complexity:

$$T(n) = \begin{cases} b & , n \le 2 \\ 7T(n/2)+an^2 & , n > 2 \end{cases}$$

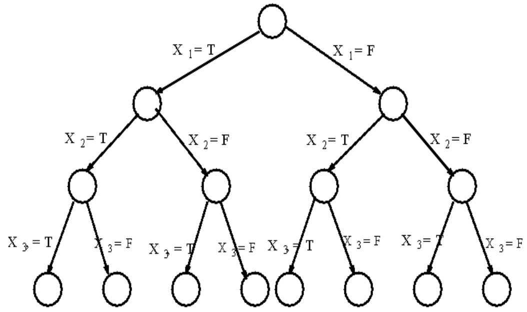
$$\begin{aligned}
T(n) &= an^2 + 7T(n/2) \\
&= an^2 + 7(a(\tfrac{n}{2})^2 + 7T(n/4)) \\
&= an^2 + \tfrac{7}{4}an^2 + 7^2T(n/4) \\
&= \cdots \\
&\ \ \vdots \\
&= an^2(1 + \tfrac{7}{4} + (\tfrac{7}{4})^2 + \cdots + (\tfrac{7}{4})^{k-1}) + 7^kT(1) \\
&\le cn^2(\tfrac{7}{4})^{\log_2 n} + 7^{\log_2 n}, \quad c \text{ is a constant} \\
&= cn^2(\tfrac{7}{4})^{\log_2 n} + n^{\log_2 7} = cn^{\log_2 4 - \log_2 7 + \log_2 4} + n^{\log_2 7} \\
&= O(n^{\log_2 7}) \cong O(n^{2.81})
\end{aligned}$$

Chapter 5

Tree Searching Strategies

Satisfiability problem

x_1	x_2	x_3
F	F	F
F	F	T
F	T	F
F	T	T
T	F	F
T	F	T
T	T	F
T	T	T

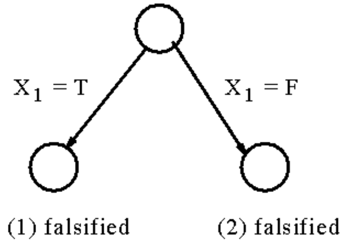


Tree representation of 8 assignments.

If there are n variables $x_1, x_2, \ldots, x_n$, then there are 2^n possible assignments.

- An instance:

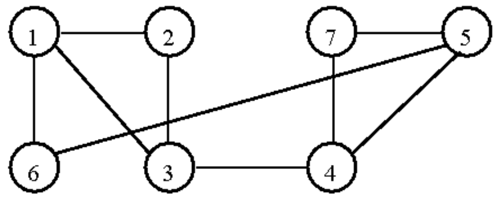
  $-x_1$..............(1)
  x_1..............(2)
  $x_2 \vee x_5$.........(3)
  x_3..............(4)
  $-x_2$..............(5)



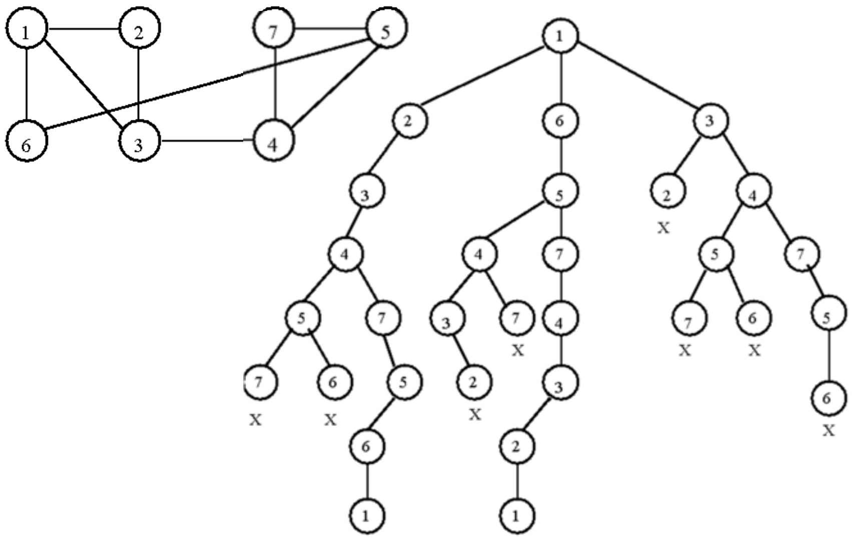
A partial tree to determine the satisfiability problem.

- We may not need to examine all possible assignments.

Hamiltonian circuit problem



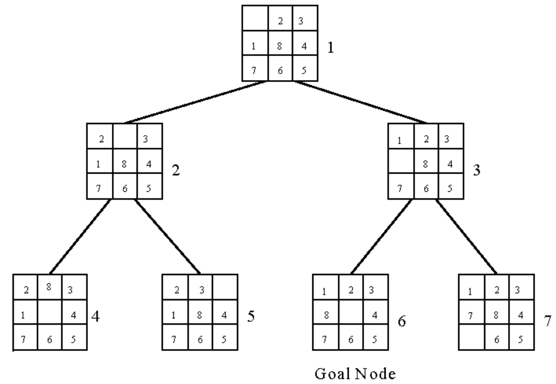
A graph containing a Hamiltonian circuit.



The tree representation of whether there exists a Hamiltonian circuit.

Breadth-first search (BFS)

- 8-puzzle problem



- The breadth-first search uses a queue to hold all expanded nodes.

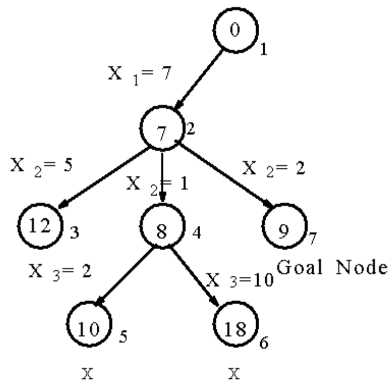
Depth-first search (DFS)

- e.g. sum of subset problem

  S={7, 5, 1, 2, 10}

  $\exists$ S' $\subseteq$ S $\ni$ sum of S' = 9 ?

- A stack can be used to guide the depth-first search.



A sum of subset problem solved by depth-first search.

Hill climbing

- A variant of depth-first search

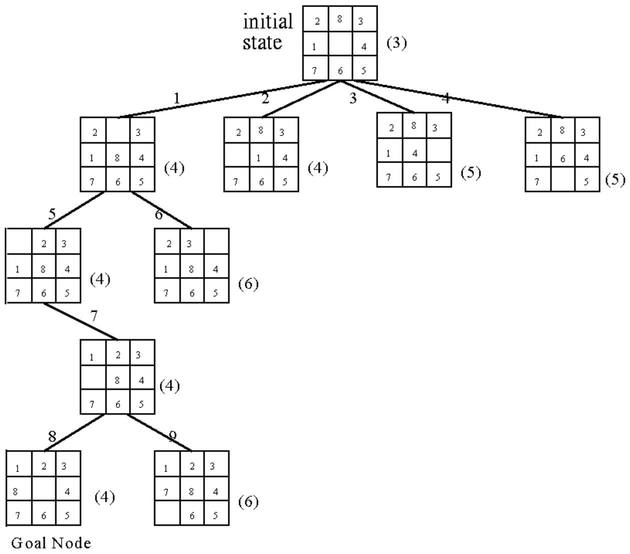
  The method selects the locally optimal node to expand.

- e.g. 8-puzzle problem

  evaluation function f(n) = d(n) + w(n)

  where d(n) is the depth of node n

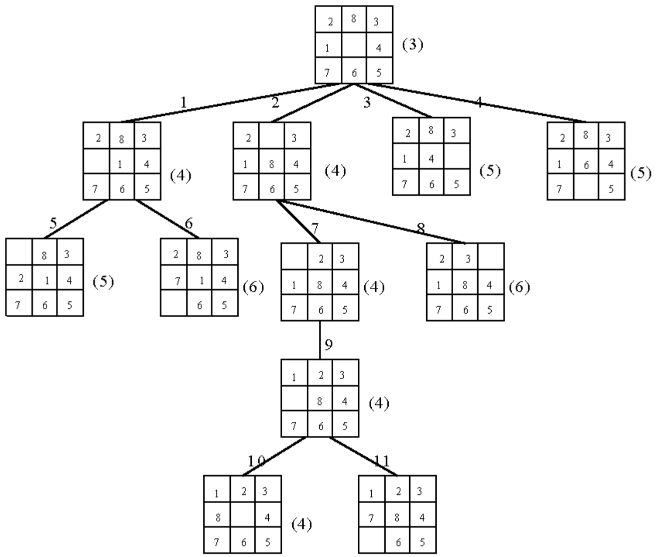
  w(n) is # of misplaced tiles in node n.



An 8-puzzle problem solved by a hill climbing method.

Best-first search strategy

- Combine depth-first search and breadth-first search.
- Selecting the node with the best estimated cost among all nodes.
- This method has a global view.
- The priority queue (heap) can be used as the data structure of best-first search.



An 8-puzzle problem solved by a best-first search scheme.

Best-First Search Scheme

Step1: Form a one-element list consisting of the root node.

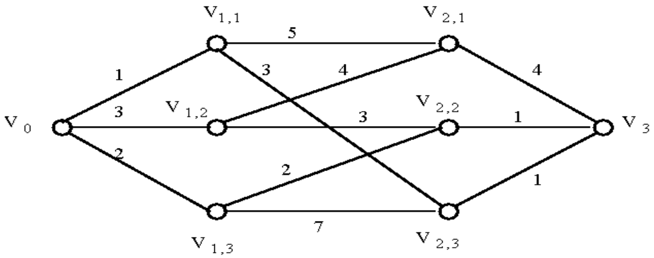
Step2: Remove the first element from the list. Expand the first element. If one of the descendants of the first element is a goal node, then stop; otherwise, add the descendants into the list.

Step3: Sort the entire list by the values of some estimation function.

Step4: If the list is empty, then failure. Otherwise, go to Step 2.

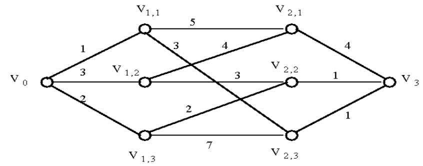
Branch-and-bound strategy

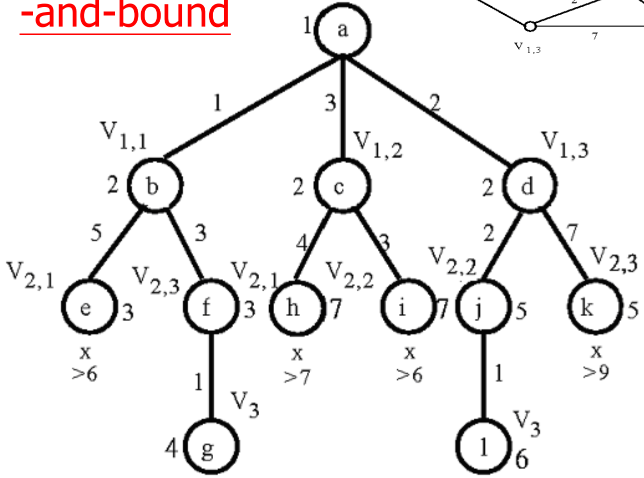
- This strategy can be used to efficiently solve optimization problems.
- e.g.



A multi-stage graph searching problem.

- Solved by branch -and-bound





Personnel assignment problem

- A linearly ordered set of persons $P=\{P_1, P_2, \ldots, P_n\}$ where $P_1<P_2<\ldots<P_n$
- A partially ordered set of jobs $J=\{J_1, J_2, \ldots, J_n\}$
- Suppose that P_i and P_j are assigned to jobs $f(P_i)$ and $f(P_j)$ respectively. If $f(P_i) \leq f(P_j)$, then $P_i \leq P_j$. Cost C_{ij} is the cost of assigning P_i to J_j. We want to find a feasible assignment with the minimum cost. i.e.

  X_{ij} = 1 if P_i is assigned to J_j

  X_{ij} = 0 otherwise.
- Minimize $\sum_{i,j} C_{ij}X_{ij}$

- e.g. A partial ordering of jobs

J_1 J_2

↓ ↘ ↓

J_3 J_4

- After topological sorting, one of the following topologically sorted sequences will be generated:

J_1, J_2, J_3, J_4

J_1, J_2, J_4, J_3

J_1, J_3, J_2, J_4

J_2, J_1, J_3, J_4

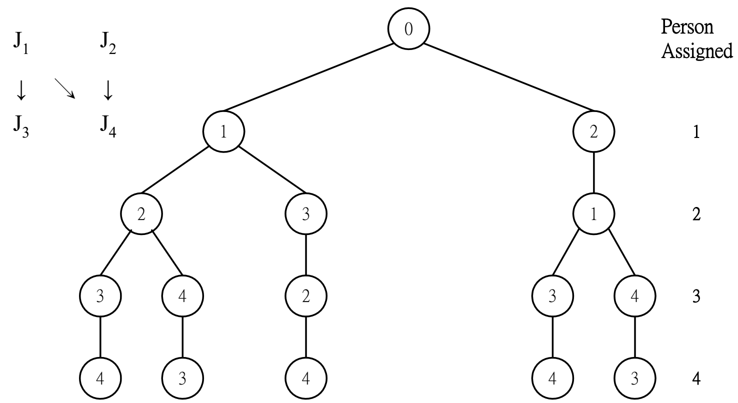
J_2, J_1, J_4, J_3

- One of feasible assignments:

  $P_1 \rightarrow J_1$, $P_2 \rightarrow J_2$, $P_3 \rightarrow J_3$, $P_4 \rightarrow J_4$

A solution tree

- All possible solutions can be represented by a solution tree.

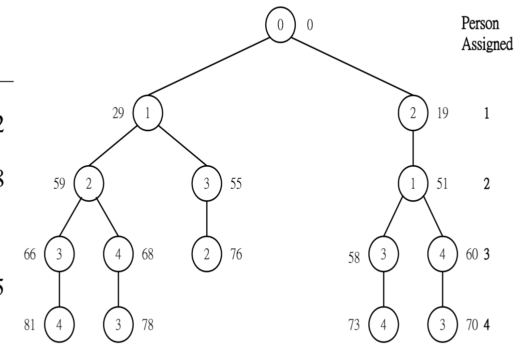


Cost matrix

- Cost matrix

Jobs / Persons	1	2	3	4
1	29	19	17	12
2	32	30	26	28
3	3	21	7	9
4	18	13	10	15

- Apply the best-first search scheme:



Only one node is pruned away.

Reduced cost matrix

- Cost matrix

Jobs / Persons	1	2	3	4
1	29	19	17	12
2	32	30	26	28
3	3	21	7	9
4	18	13	10	15

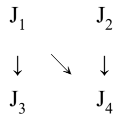
- Reduced cost matrix

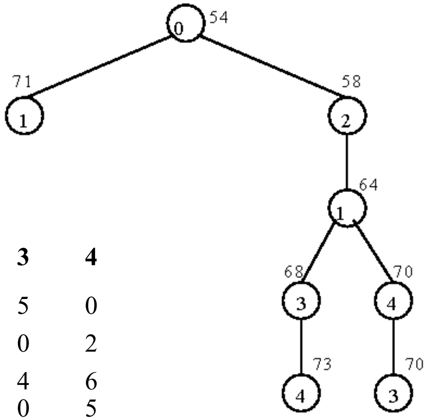
Jobs / Persons	1	2	3	4	
1	17	4	5	0	(-12)
2	6	1	0	2	(-26)
3	0	15	4	6	(-3)
4	8	0	0	5	(-10)
		(-3)			

- A reduced cost matrix can be obtained: subtract a constant from each row and each column respectively such that each row and each column contains at least one zero.
- Total cost subtracted: 12+26+3+10+3 = 54
- This is a lower bound of our solution.

Branch-and-bound for the personnel assignment problem

- Bounding of subsolutions:





Person Assigned

1

2

3

4

Jobs / Persons	1	2	3	4
1	17	4	5	0
2	6	1	0	2
3	0	15	4	6
4	8	0	0	5

The traveling salesperson optimization problem

- It is NP-complete.
- A cost matrix

i \ j	1	2	3	4	5	6	7
1	∞	3	93	13	33	9	57
2	4	∞	77	42	21	16	34
3	45	17	∞	36	16	28	25
4	39	90	80	∞	56	7	91
5	28	46	88	33	∞	25	57
6	3	88	18	46	92	∞	7
7	44	26	33	27	84	39	∞

- A reduced cost matrix

i \ j	1	2	3	4	5	6	7	
1	∞	0	90	10	30	6	54	(-3)
2	0	∞	73	38	17	12	30	(-4)
3	29	1	∞	20	0	12	9	(-16)
4	32	83	73	∞	49	0	84	(-7)
5	3	21	63	8	∞	0	32	(-25)
6	0	85	15	43	89	∞	4	(-3)
7	18	0	7	1	58	13	∞	(-26)

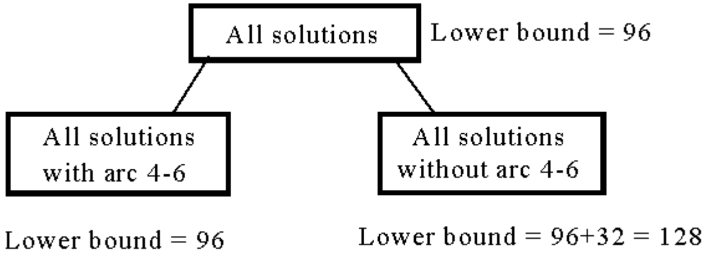
Reduced: 84

- Another reduced matrix

i \ j	1	2	3	4	5	6	7
1	∞	0	83	9	30	6	50
2	0	∞	66	37	17	12	26
3	29	1	∞	19	0	12	5
4	32	83	66	∞	49	0	80
5	3	21	56	7	∞	0	28
6	0	85	8	42	89	∞	0
7	18	0	0	0	58	13	∞
			(-7)	(-1)			(-4)

Total cost reduced: 84+7+1+4 = 96 (lower bound)

- The highest level of a decision tree:



- If we use arc 3-5 to split, the difference on the lower bounds is 17+1 = 18.

- A reduced cost matrix if arc (4,6) is included in the solution.

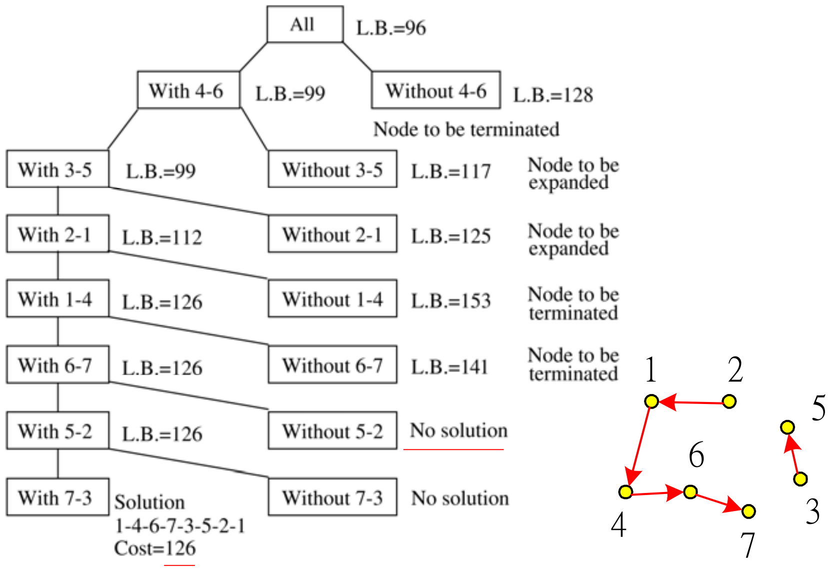
i \ j	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	3	21	56	7	∞	28
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

Arc (6,4) is changed to be infinity since it can not be included in the solution.

- The reduced cost matrix for all solutions with arc 4-6

i \ j	1	2	3	4	5	7	
1	∞	0	83	9	30	50	
2	0	∞	66	37	17	26	
3	29	1	∞	19	0	5	
5	0	18	53	4	∞	25	(-3)
6	0	85	8	∞	89	0	
7	18	0	0	0	58	∞	

- Total cost reduced: 96+3 = 99 (new lower bound)



A branch-and-bound solution of a traveling salesperson problem.

The 0/1 knapsack problem

- Positive integer $P_1, P_2, \ldots, P_n$ (profit)
  $W_1, W_2, \ldots, W_n$ (weight)
  M (capacity)

$$\text{maximize } \sum_{i=1}^{n} P_i X_i$$

$$\text{subject to } \sum_{i=1}^{n} W_i X_i \le M \quad X_i = 0 \text{ or } 1,\ i = 1, \ldots, n.$$

The problem is modified:

$$\text{minimize } -\sum_{i=1}^{n} P_i X_i$$

- e.g. n = 6, M = 34

i	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$(P_i/W_i \ge P_{i+1}/W_{i+1})$

- A feasible solution: $X_1 = 1$, $X_2 = 1$, $X_3 = 0$, $X_4 = 0$, $X_5 = 0$, $X_6 = 0$
  $-(P_1+P_2) = -16$ (upper bound)
  Any solution higher than -16 can not be an optimal solution.

Relax the restriction

- Relax our restriction from $X_i = 0$ or 1 to $0 \le X_i \le 1$ (knapsack problem)

Let $-\sum_{i=1}^{n} P_i X_i$ be an optimal solution for 0/1 knapsack problem and $-\sum_{i=1}^{n} P_i X'_i$ be an optimal solution for knapsack problem. Let $Y = -\sum_{i=1}^{n} P_i X_i$,

$$Y' = -\sum_{i=1}^{n} P_i X'_i.$$

$\Rightarrow Y' \le Y$

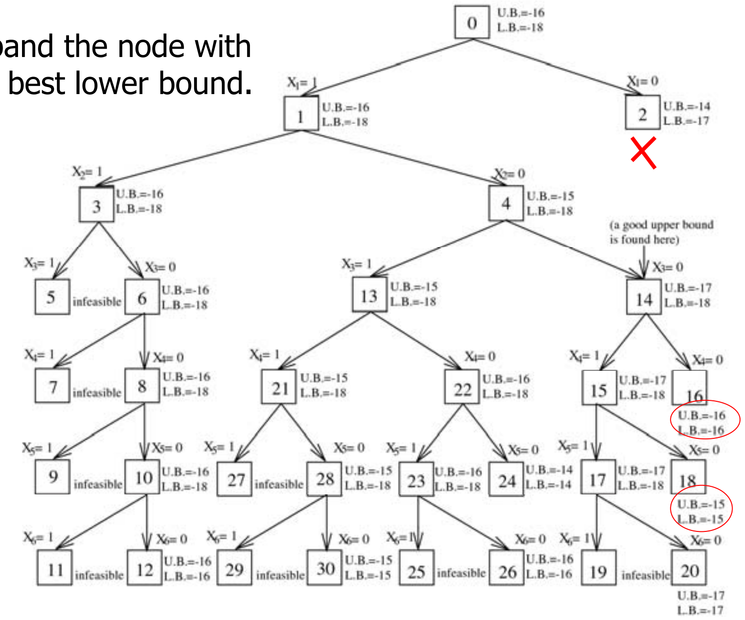
Upper bound and lower bound

- We can use the greedy method to find an optimal solution for knapsack problem:

  $X_1 = 1$, $X_2 = 1$, $X_3 = 5/8$, $X_4 = 0$, $X_5 = 0$, $X_6 = 0$
  $-(P_1+P_2+5/8P_3) = -18.5$ (lower bound)
  -18 is our lower bound. (only consider integers)

$\Rightarrow$ $-18 \le$ optimal solution ≤ -16
optimal solution: $X_1 = 1$, $X_2 = 0$, $X_3 = 0$, $X_4 = 1$, $X_5 = 1$, $X_6 = 0$
$-(P_1+P_4+P_5) = -17$

Expand the node with the best lower bound.



0/1 knapsack problem solved by branch-and-bound strategy.

The A^* algorithm

- Used to solve optimization problems.
- Using the best-first strategy.
- If a feasible solution (goal node) is obtained, then it is optimal and we can stop.
- Cost function of node n : f(n)

  f(n) = g(n) + h(n)

  g(n): cost from root to node n.

  h(n): estimated cost from node n to a goal node.

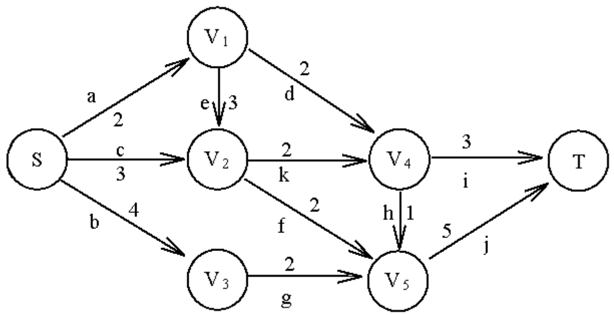
  $h^*(n)$: "real" cost from node n to a goal node.

- If we guarantee $h(n) \le h^*(n)$, then

  $f(n) = g(n) + h(n) \le g(n)+h^*(n) = f^*(n)$

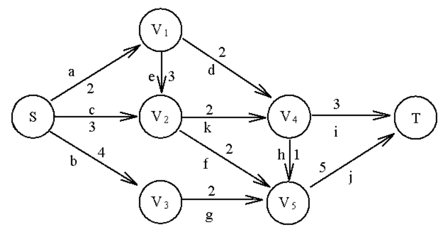
An example for A^* algorithm

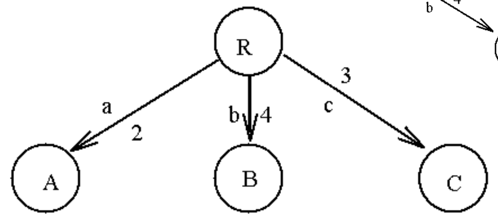
- Find the shortest path with A^* algorithm.



- Stop iff the selected node is also a goal node.

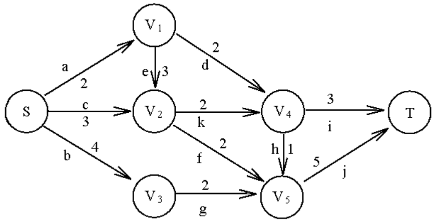
- Step 1:

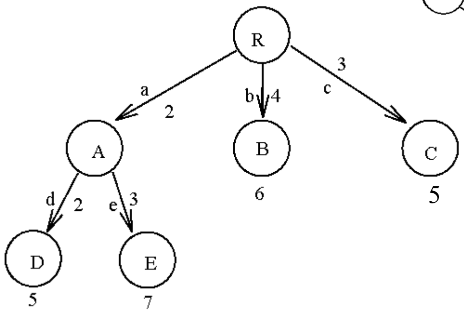




g(A)=2	h(A)=min{2,3}=2	f(A)=2+2=4
g(B)=4	h(B)=min{2}=2	f(B)=4+2=6
g(C)=3	h(C)=min{2,2}=2	f(C)= 3+2=5

- Step 2: Expand node A.

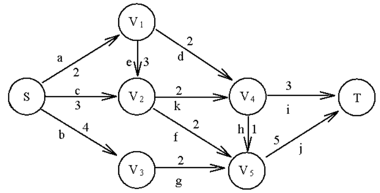


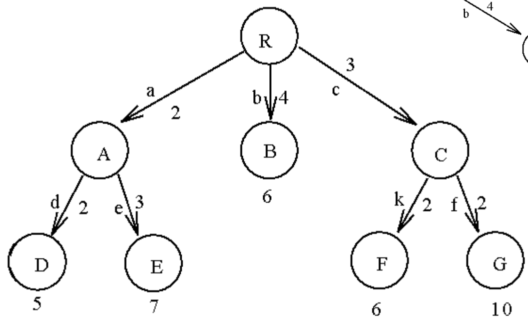


$g(D)=2+2=4$ $\quad h(D)=\min\{3,1\}=1$ $\quad f(D)=4+1=5$

$g(E)=2+3=5$ $\quad h(E)=\min\{2,2\}=2$ $\quad f(E)=5+2=7$

- Step 3: Expand node C.

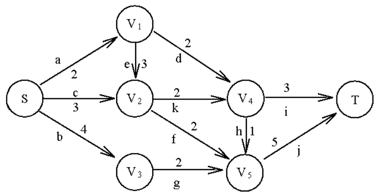


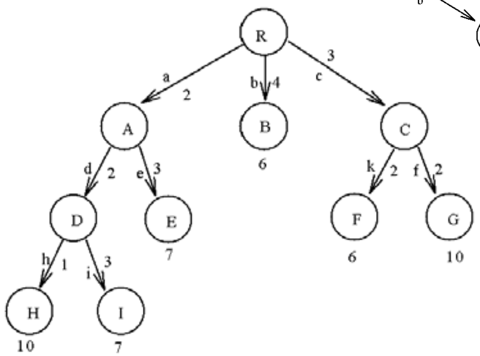


$g(F)=3+2=5$ $\quad h(F)=\min\{3,1\}=1$ $\quad f(F)=5+1=6$

$g(G)=3+2=5$ $\quad h(G)=\min\{5\}=5$ $\quad f(G)=5+5=10$

- Step 4: Expand node D.

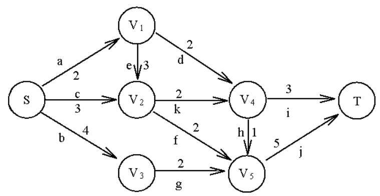


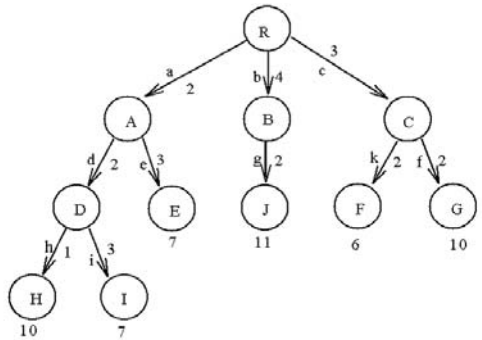


$g(H)=2+2+1=5$ $\quad h(H)=\min\{5\}=5$ $\quad f(H)=5+5=10$

$g(I)=2+2+3=7$ $\quad h(I)=0$ $\quad f(I)=7+0=7$

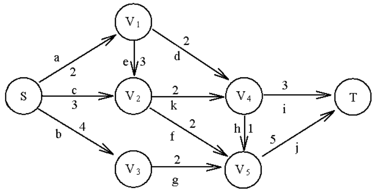
- Step 5: Expand node B.

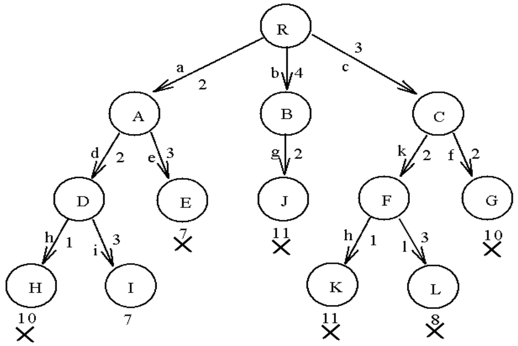




$g(J)=4+2=6$ $\quad h(J)=\min\{5\}=5$ $\quad f(J)=6+5=11$

- Step 6: Expand node F.





$$f(n) \le f^*(n)$$

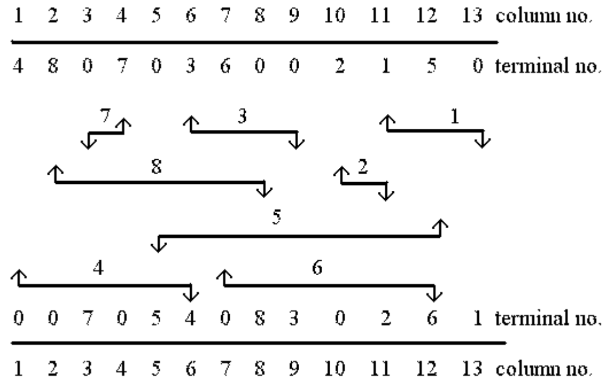
$g(K) = 3 + 2 + 1 = 6$ $\quad h(K) = \min\{5\} = 5$ $\quad f(K) = 6 + 5 = 11$

$g(L) = 3 + 2 + 3 = 8$ $\quad h(L) = 0$ $\quad f(L) = 8 + 0 = 8$

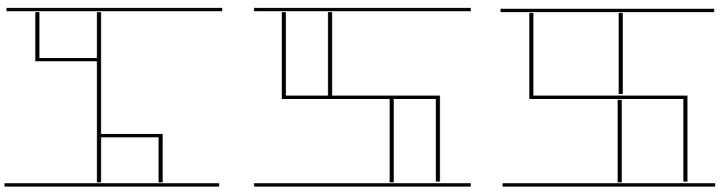
Node I is a goal node. Thus, the final solution has been obtained.

The channel routing problem

- A channel specification

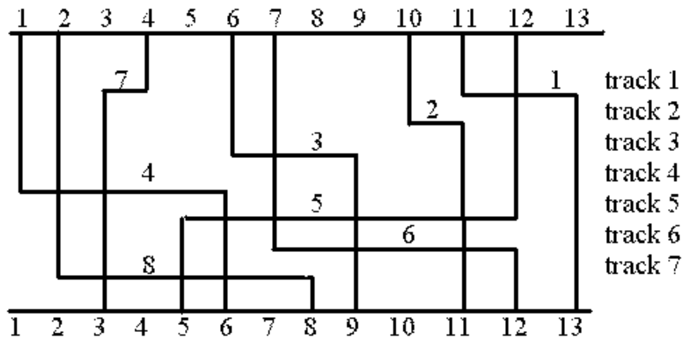


- Illegal wirings:



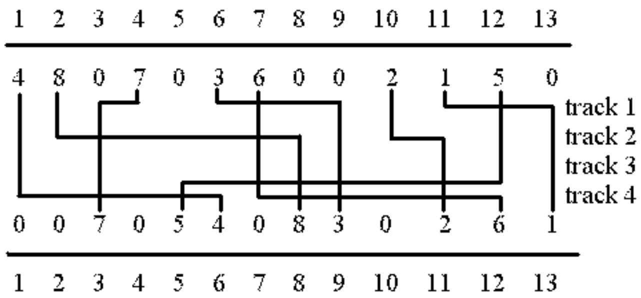
- We want to find a routing which minimizes the number of tracks.

A feasible routing



- 7 tracks are needed.

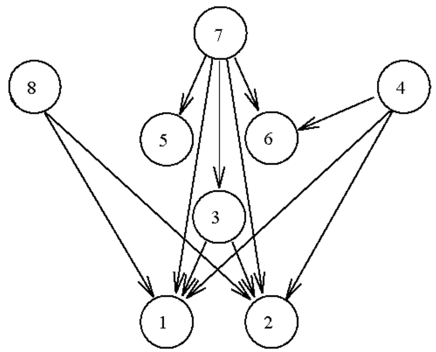
An optimal routing



- 4 tracks are needed.
- This problem is NP-complete.

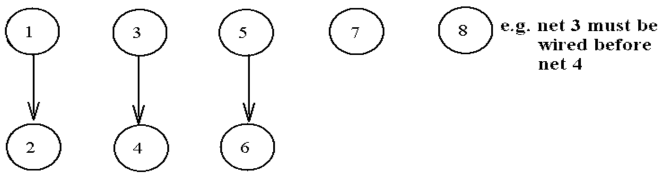
A* algorithm for the channel routing problem

- Horizontal constraint graph (HCG)



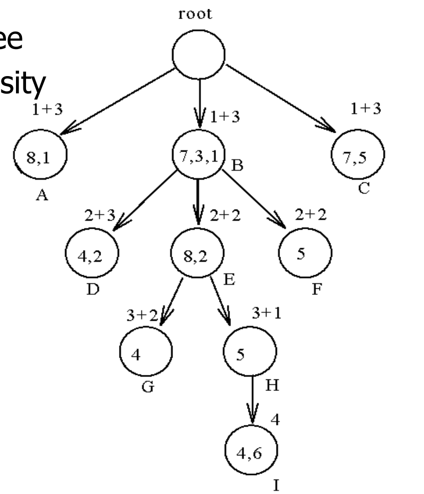
- e.g. net 8 must be to the left of net 1 and net 2 if they are in the same track.

- Vertical constraint graph:



- Maximum cliques in HCG: {1,8}, {1,3,7}, {5,7}. Each maximum clique can be assigned to a track.

- f(n) = g(n) + h(n),
  - g(n): the level of the tree
  - h(n): maximal local density



A partial solution tree for the channel routing problem by using A* algorithm.

Chapter 6

Prune-and-Search

A simple example: Binary search

- sorted sequence : (search 9)

	1	4	5	7	9	10	12	15
step 1				↑				
step 2						↑		
step 3					↑			

- After each comparison, a half of the data set are pruned away.
- Binary search can be viewed as a special divide-and-conquer method, since there exists no solution in another half and then no merging is done.

The selection problem

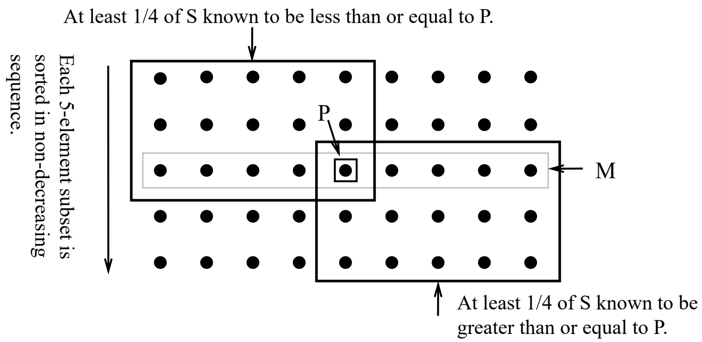
- Input: A set S of n elements
- Output: The kth smallest element of S
- The median problem: to find the $\left\lceil \frac{n}{2} \right\rceil$ -th smallest element.
- The straightforward algorithm:
  - step 1: Sort the n elements
  - step 2: Locate the kth element in the sorted list.
  - Time complexity: O(nlogn)

Prune-and-search concept for the selection problem

- $S=\{a_1, a_2, \ldots, a_n\}$
- Let $p \in S$, use p to partition S into 3 subsets S_1 , S_2 , S_3:
  - $S_1=\{ a_i \mid a_i < p , 1 \le i \le n\}$
  - $S_2=\{ a_i \mid a_i = p , 1 \le i \le n\}$
  - $S_3=\{ a_i \mid a_i > p , 1 \le i \le n\}$
- 3 cases:
  - If $|S_1| \ge k$, then the kth smallest element of S is in S_1, prune away S_2 and S_3.
  - Else, if $|S_1| + |S_2| \ge k$, then p is the kth smallest element of S.
  - Else, the kth smallest element of S is the $(k - |S_1| - |S_2|)$-th smallest element in S_3, prune away S_1 and S_2.

How to select P?

- The n elements are divided into $\left\lceil \frac{n}{5} \right\rceil$ subsets. (Each subset has 5 elements.)



Prune-and-search approach

- Input: A set S of n elements.
- Output: The kth smallest element of S.

Step 1: Divide S into $\lceil n/5 \rceil$ subsets. Each subset contains five elements. Add some dummy ∞ elements to the last subset if n is not a net multiple of S.

Step 2: Sort each subset of elements.

Step 3: Recursively, find the element p which is the median of the medians of the $\lceil n/5 \rceil$ subsets..

Step 4: Partition S into S_1, S_2 and S_3, which contain the elements less than, equal to, and greater than p, respectively.

Step 5: If $|S_1| \geq k$, then discard S_2 and S_3 and solve the problem that selects the kth smallest element from S_1 during the next iteration;

else if $|S_1| + |S_2| \geq k$ then p is the kth smallest element of S;

otherwise, let $k' = k - |S_1| - |S_2|$, solve the problem that selects the k'th smallest element from S_3 during the next iteration.

Time complexity

- At least n/4 elements are pruned away during each iteration.
- The problem remaining in step 5 contains at most 3n/4 elements.
- Time complexity: $T(n) = O(n)$
  - step 1: $O(n)$
  - step 2: $O(n)$
  - step 3: $T(n/5)$
  - step 4: $O(n)$
  - step 5: $T(3n/4)$
  - $T(n) = T(3n/4) + T(n/5) + O(n)$

Let $T(n) = a_0 + a_1 n + a_2 n^2 + \ldots$, $a_1 \neq 0$
$T(3n/4) = a_0 + (3/4)a_1 n + (9/16)a_2 n^2 + \ldots$
$T(n/5) = a_0 + (1/5)a_1 n + (1/25)a_2 n^2 + \ldots$
$T(3n/4 + n/5) = T(19n/20) = a_0 + (19/20)a_1 n + (361/400)a_2 n^2 + \ldots$
$T(3n/4) + T(n/5) \leq a_0 + T(19n/20)$

$$
\begin{aligned}
\Rightarrow T(n) &\leq cn + T(19n/20) \\
&\leq cn + (19/20)cn + T((19/20)^2 n) \\
&\vdots \\
&\leq cn + (19/20)cn + (19/20)^2 cn + \ldots + (19/20)^p cn + T((19/20)^{p+1} n) \quad , (19/20)^{p+1} n \leq 1 \leq (19/20)^p n \\
&= \frac{1-(\frac{19}{20})^{p+1}}{1-\frac{19}{20}} cn + b \\
&\leq 20\,cn + b \\
&= O(n)
\end{aligned}
$$

The general prune-and-search

- It consists of many iterations.
- At each iteration, it prunes away a fraction, say f, $0<f<1$, of the input data, and then it invokes the same algorithm recursively to solve the problem for the remaining data.
- After p iterations, the size of input data will be q which is so small that the problem can be solved directly in some constant time c.

Time complexity analysis

- Assume that the time needed to execute the prune-and-search in each iteration is $O(n^k)$ for some constant k and the worst case run time of the prune-and-search algorithm is $T(n)$. Then

$$T(n) = T((1-f)\,n) + O(n^k)$$

- We have

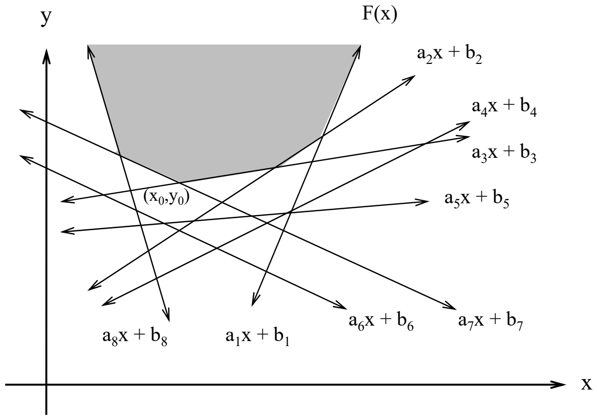
$$
\begin{aligned}
T(n) &\leq T((1-f)\,n) + cn^k \text{ for sufficiently large } n. \\
&\leq T((1-f)^2 n) + cn^k + c(1-f)^k n^k \\
&\vdots \\
&\leq c' + cn^k + c(1-f)^k n^k + c(1-f)^{2k} n^k + \ldots + c(1-f)^{pk} n^k \\
&= c' + cn^k(1 + (1-f)^k + (1-f)^{2k} + \ldots + (1-f)^{pk}).
\end{aligned}
$$

Since $1-f < 1$, as $n \to \infty$,
$\therefore T(n) = O(n^k)$

- Thus, the time-complexity of the whole prune-and-search process is of the same order as the time-complexity in each iteration.

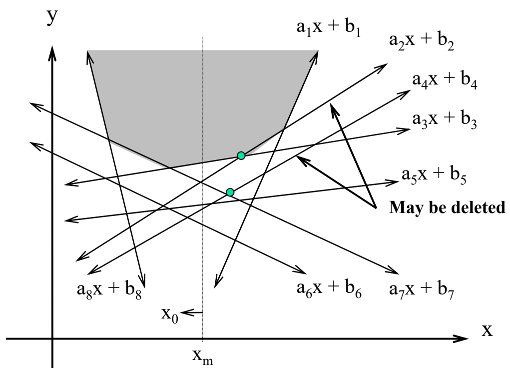
Linear programming with two variables

- Minimize ax + by
  subject to $a_i x + b_i y \geq c_i$, i = 1, 2, …, n
- Simplified two-variable linear programming problem:
  Minimize y
  subject to $y \geq a_i x + b_i$, i = 1, 2, …, n



- The boundary F(x):
  $F(x) = \max_{1 \leq i \leq n} \{a_i x + b_i\}$
- The optimum solution x_0:
  $F(x_0) = \min_{-\infty < x < \infty} F(x)$

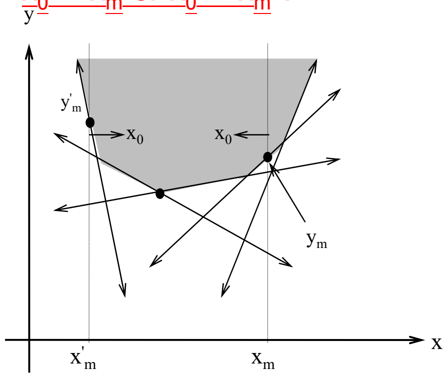
Constraints deletion



- If $x_0 < x_m$ and the intersection of $a_3x + b_3$ and $a_2x + b_2$ is greater than x_m, then one of these two constraints is always smaller than the other for $x < x_m$. Thus, this constraint can be deleted.
- It is similar for $x_0 > x_m$.

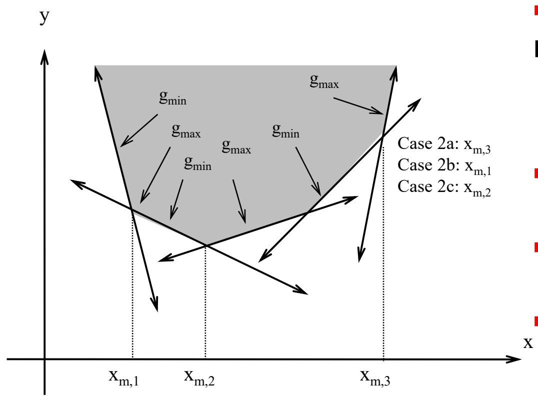
Determining the direction of the optimum solution

Suppose an x_m is known.
How do we know whether $x_0 < x_m$ or $x_0 > x_m$?



- Let $y_m = F(x_m) = \max_{1 \leq i \leq n} \{a_i x_m + b_i\}$
- **Case 1:** y_m is on only one constraint.
  - Let g denote the slope of this constraint.
  - If g > 0, then $x_0 < x_m$.
  - If g < 0, then $x_0 > x_m$.

The cases where x_m is on only one constrain.



Cases of x_m on the intersection of several constraints.

- **Case 2:** y_m is the intersection of several constraints.
  - $g_{max} = \max\limits_{1 \le i \le n} \{a_i \mid a_i x_m + b_i = F(x_m)\}$
    max. slope
    $g_{min} = \min\limits_{1 \le i \le n} \{a_i \mid a_i x_m + b_i = F(x_m)\}$
    min. slop
  - **If** $g_{min} > 0$, $g_{max} > 0$, then $x_0 < x_m$
  - **If** $g_{min} < 0$, $g_{max} < 0$, then $x_0 > x_m$
  - **If** $g_{min} < 0$, $g_{max} > 0$, then (x_m, y_m) is the optimum solution.

How to choose x_m?

- We arbitrarily group the n constraints into n/2 pairs. For each pair, find their intersection. Among these n/2 intersections, choose the median of their x-coordinates as x_m.

Prune-and-Search approach

- Input: Constraints S: $a_i x + b_i$, i=1, 2, …, n.
- Output: The value x_0 such that y is minimized at x_0 subject to the above constraints.

Step 1: If S contains no more than two constraints, solve this problem by a brute force method.

Step 2: Divide S into n/2 pairs of constraints randomly. For each pair of constraints $a_i x + b_i$ and $a_j x + b_j$, find the intersection p_{ij} of them and denote its x-value as x_{ij}.

Step 3: Among the x_{ij}'s, find the median x_m.

Step 4: Determine $y_m = F(x_m) = \max\limits_{1 \le i \le n} \{a_i x_m + b_i\}$

$g_{min} = \min\limits_{1 \le i \le n} \{a_i \mid a_i x_m + b_i = F(x_m)\}$

$g_{max} = \max\limits_{1 \le i \le n} \{a_i \mid a_i x_m + b_i = F(x_m)\}$

Step 5:

Case 5a: If g_{min} and g_{max} are not of the same sign, y_m is the solution and exit.

Case 5b: otherwise, $x_0 < x_m$, if $g_{min} > 0$, and $x_0 > x_m$, if $g_{min} < 0$.

Step 6:

Case 6a: If $x_0 < x_m$, for each pair of constraints whose x-coordinate intersection is larger than x_m, prune away the constraint which is always smaller than the other for $x \le x_m$.

Case 6b: If $x_0 > x_m$, do similarly.

Let S denote the set of remaining constraints. Go to Step 2.

- There are totally $\lfloor n/2 \rfloor$ intersections. Thus, $\lfloor n/4 \rfloor$ constraints are pruned away for each iteration.
- Time complexity:

$$T(n) = T(3n/4)+O(n)$$
$$= O(n)$$

The general two-variable linear programming problem

$$\begin{cases} \text{Minimize } ax + by \\ \text{subject to } a_i x + b_i y \ge c_i \ , i = 1, 2, \ldots, n \end{cases}$$

Let $x' = x$

$y' = ax + by$

$\Downarrow$

$$\begin{cases} \text{Minimize } y' \\ \text{subject to } a_i'x' + b_i'y' \ge c_i' \ , i = 1, 2, \ldots, n \\ \text{where } a_i' = a_i - b_i a/b,\ b_i' = b_i/b,\ c_i' = c_i \end{cases}$$

Change the symbols and rewrite as:

$$\begin{cases} \text{Minimize } y \\ \text{subject to } y \ge a_i x + b_i \ (i \in I_1) \\ \qquad\qquad\quad y \le a_i x + b_i \ (i \in I_2) \\ \qquad\qquad\quad a \le x \le b \end{cases}$$

Define:

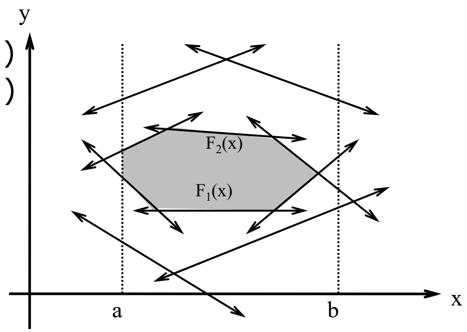
$F_1(x) = \max \{a_i x + b_i \ , i \in I_1\}$

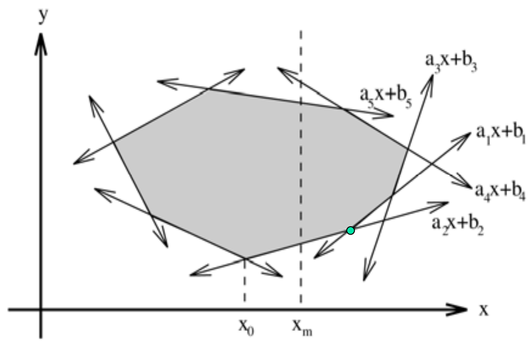
$F_2(x) = \min \{a_i x + b_i \ , i \in I_2\}$

$\Downarrow$

$$\begin{cases} \text{Minimize } F_1(x) \\ \text{subject to } F_1(x) \le F_2(x),\ a \le x \le b \end{cases}$$

Let $F(x) = F_1(x) - F_2(x)$



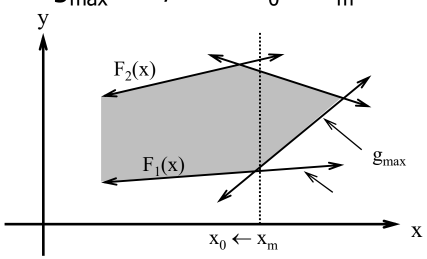


- If we know $x_0 < x_m$, then $a_1x + b_1$ can be deleted because $a_1x + b_1 < a_2x + b_2$ for $x< x_m$.
- Define:
  - $g_{min} = \min \{a_i \mid i \in I_1,\ a_i x_m + b_i = F_1(x_m)\}$, min. slope
  - $g_{max} = \max\{a_i \mid i \in I_1,\ a_i x_m + b_i = F_1(x_m)\}$, max. slope
  - $h_{min} = \min \{a_i \mid i \in I_2,\ a_i x_m + b_i = F_2(x_m)\}$, min. slope
  - $h_{max} = \max\{a_i \mid i \in I_2,\ a_i x_m + b_i = F_2(x_m)\}$, max. slope

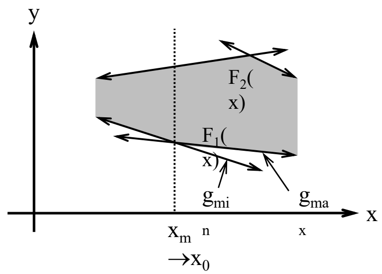
Determining the solution

- <u>Case 1:</u> If $F(x_m) \leq 0$, then x_m is feasible.

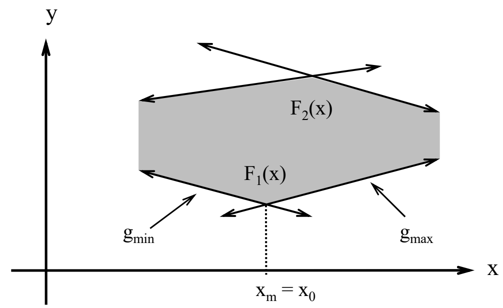
<u>Case 1.a:</u> If $g_{min} > 0$, $g_{max} > 0$, then $x_0 < x_m$.



<u>Case 1.b:</u> If $g_{min} < 0$, $g_{max} < 0$, then $x_0 > x_m$.

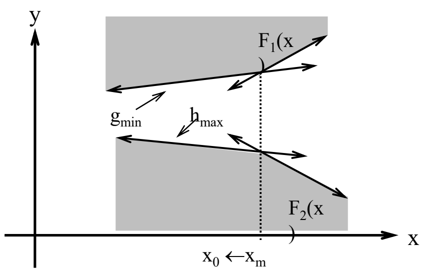


<u>Case 1.c:</u> If $g_{min} < 0$, $g_{max} > 0$, then x_m is the optimum solution.

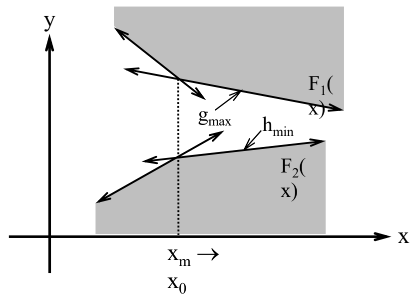


- <u>Case 2:</u> If $F(x_m) > 0$, x_m is infeasible.

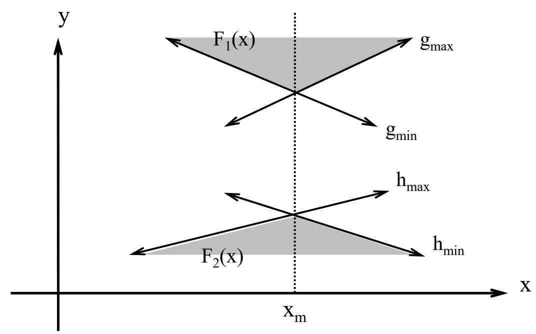
<u>Case 2.a:</u> If $g_{min} > h_{max}$, then $x_0 < x_m$.



<u>Case 2.b:</u> If $g_{min} < h_{max}$, then $x_0 > x_m$.



<u>Case 2.c:</u> If $g_{min} \leq h_{max}$, and $g_{max} \geq h_{min}$, then no feasible solution exists.



Prune-and-search approach

- Input: Constraints:

  I_1: $y \geq a_i x + b_i$, $i = 1, 2, \ldots, n_1$
  I_2: $y \leq a_i x + b_i$, $i = n_1+1, n_1+2, \ldots, n.$
  $a \leq x \leq b$

- Output: The value x_0 such that
  y is minimized at x_0
  subject to the above constraints.

Step 1: Arrange the constraints in I_1 and I_2 into arbitrary disjoint pairs respectively. For each pair, if $a_i x + b_i$ is parallel to $a_j x + b_j$, delete $a_i x + b_i$ if $b_i < b_j$ for $i, j \in I_1$ or $b_i > b_j$ for $i, j \in I_2$. Otherwise, find the intersection p_{ij} of $y = a_i x + b_i$ and $y = a_j x + b_j$. Let the x-coordinate of p_{ij} be x_{ij}.

Step 2: Find the median x_m of x_{ij}'s (at most $\lfloor \frac{n}{2} \rfloor$ points).

Step 3:

a. If x_m is optimal, report this and exit.
b. If no feasible solution exists, report this and exit.
c. Otherwise, determine whether the optimum solution lies to the left, or right, of x_m.

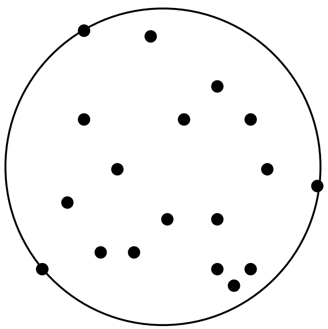
Step 4: Discard at least 1/4 of the constraints. Go to Step 1.

- Time complexity:

$$T(n) = T(3n/4)+O(n) \\ = O(n)$$

The 1-center problem

- Given n planar points, find a smallest circle to cover these n points.

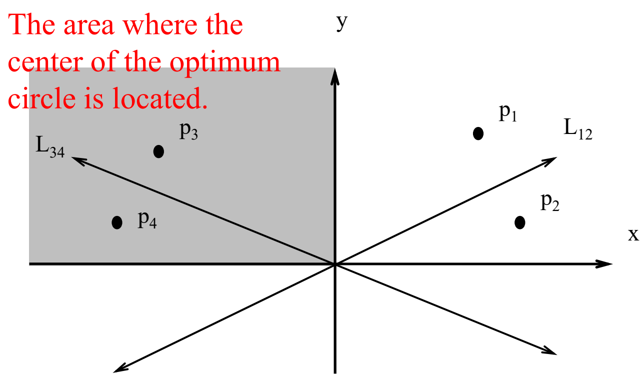


The pruning rule

$L_{1\,2}$: bisector of segment connecting p_1 and p_2 ,

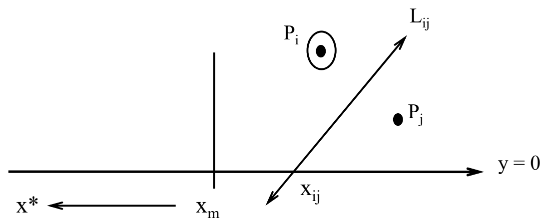
$L_{3\,4}$: bisector of segments connecting p_3 and p_4

P_1 can be eliminated without affecting our solution.



The constrained 1-center problem

- The center is restricted to lying on a straight line.



Prune-and-search approach

- Input : n points and a straight line y = y'.
- Output: The constrained center on the straight line y = y'.

Step 1: If n is no more than 2, solve this problem by a brute-force method.

Step 2: Form disjoint pairs of points (p_1, p_2), (p_3, p_4), …,(p_{n-1}, p_n). If there are odd number of points, just let the final pair be (p_n, p_1).

Step 3: For each pair of points, (p_i, p_{i+1}), find the point $x_{i,i+1}$ on the line y = y' such that $d(p_i, x_{i,i+1}) = d(p_{i+1}, x_{i,i+1})$.

Step 4: Find the median of the $\left\lfloor \frac{n}{2} \right\rfloor$ $x_{i,i+1}$'s. Denote it as x_m.

Step 5: Calculate the distance between p_i and x_m for all i. Let p_j be the point which is farthest from x_m. Let x_j denote the projection of p_j onto y = y'. If x_j is to the left (right) of x_m, then the optimal solution, x^*, must be to the left (right) of x_m.

Step 6: If $x^* < x_m$, for each $x_{i,i+1} > x_m$, prune the point p_i if p_i is closer to x_m than p_{i+1}, otherwise prune the point p_{i+1};

If $x^* > x_m$, do similarly.

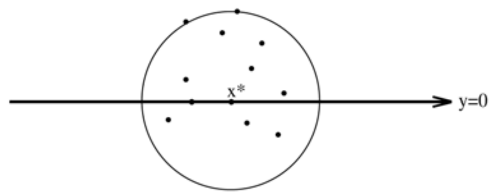
Step 7: Go to Step 1.

- Time complexity

$$T(n) = T(3n/4)+O(n) = O(n)$$

The general 1-center problem

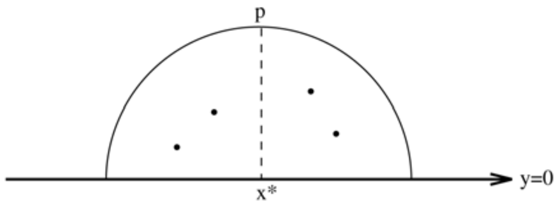
- By the constrained 1-center algorithm, we can determine the center $(x^*,0)$ on the line y=0.
- We can do more
  - Let (x_s, y_s) be the center of the optimum circle.
  - We can determine whether $y_s > 0$, $y_s < 0$ or $y_s = 0$.
  - Similarly, we can also determine whether $x_s > 0$, $x_s < 0$ or $x_s = 0$



The sign of optimal y

- Let I be the set of points which are farthest from $(x^*, 0)$.
- Case 1: I contains one point $P = (x_p, y_p)$.

  y_s has the same sign as that of y_p.



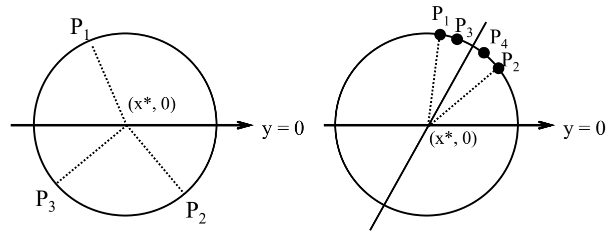
- Case 2 : I contains more than one point.

  Find the smallest arc spanning all points in I.

  Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be the two end points of the smallest spanning arc.

  If this arc $\geq 180^o$, then $y_s = 0$.

  else y_s has the same sign as that of $\frac{y_1 + y_2}{2}$.

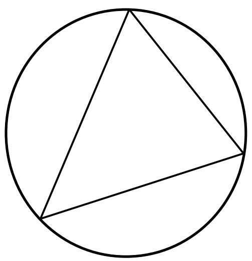


(a) (b)

(See the figure on the next page.)

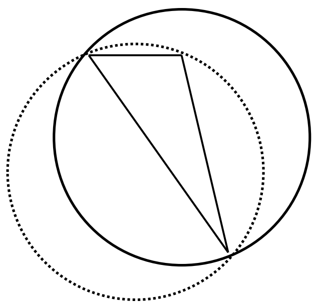
Optimal or not optimal

- an acute triangle:



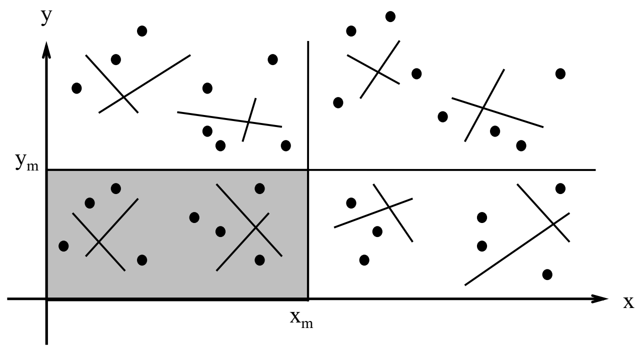
The circle is optimal.

- an obtuse triangle:



The circle is not optimal.

An example of 1-center problem



- One point for each of n/4 intersections of L_{i+} and L_{i-} is pruned away.
- Thus, n/16 points are pruned away in each iteration.

Prune-and-search approach

- Input: A set $S = \{p_1, p_2, \ldots, p_n\}$ of n points.
- Output: The smallest enclosing circle for S.

Step 1: If S contains no more than 16 points, solve the problem by a brute-force method.

Step 2: Form disjoint pairs of points, (p_1, p_2), (p_3, p_4), …,(p_{n-1}, p_n). For each pair of points, (p_i, p_{i+1}), find the perpendicular bisector of line segment $\overline{p_i p_{i+1}}$ .Denote them as $L_{i/2}$, for i = 2, 4, …, n, and compute their slopes. Let the slope of L_k be denoted as s_k, for k = 1, 2, 3, …, n/2.

Step 3: Compute the median of s_k's, and denote it by s_m.

Step 4: Rotate the coordinate system so that the x-axis coincide with $y = s_m x$. Let the set of L_k's with positive (negative) slopes be I^+ (I^-). (Both of them are of size n/4.)

Step 5: Construct disjoint pairs of lines, (L_{i+}, L_{i-}) for i = 1, 2, …, n/4, where $L_{i+} \in I^+$ and $L_{i-} \in I^-$. Find the intersection of each pair and denote it by (a_i, b_i), for i = 1, 2, …, n/4.

Step 6: Find the median of b_i's. Denote it as y^*. Apply the constrained 1-center subroutine to S, requiring that the center of circle be located on $y=y^*$. Let the solution of this constrained 1-center problem be (x', y^*).

Step 7: Determine whether (x', y^*) is the optimal solution. If it is, exit; otherwise, record $y_s > y^*$ or $y_s < y^*$.

- Step 8: If $y_s > y^*$, find the median of a_i's for those (a_i, b_i)'s where $b_i < y^*$. If $y_s < y^*$, find the median of a_i's of those t hose (a_i, b_i)'s where $b_i > y^*$. Denote the median as x^*. Apply the constrained 1-center algorithm to S, requiring that the center of circle be located on $x = x^*$. Let the solution of this contained 1-center problem be (x^*, y').
- Step 9: Determine whether (x^*, y') is the optimal solution. If it is, exit; otherwise, record $x_s > x^*$ and $x_s < x^*$.

Step 10:

- Case 1: $x_s < x^*$ and $y_s < y^*$.

  Find all (a_i, b_i)'s such that $a_i > x^*$ and $b_i > y^*$. Let (a_i, b_i) be the intersection of L_{i+} and L_{i-}. Let L_{i-} be the bisector of p_j and p_k. Prune away $p_j(p_k)$ if $p_j(p_k)$ is closer to (x^*, y^*) than $p_k(p_j)$.
- Case 2: $x_s > x^*$ and $y_s > y^*$. Do similarly.
- Case 3: $x_s < x^*$ and $y_s > y^*$. Do similarly.
- Case 4: $x_s > x^*$ and $y_s < y^*$. Do similarly.

Step 11: Let S be the set of the remaining points. Go to Step 1.

- Time complexity :

$$T(n) = T(15n/16)+O(n)$$
$$= O(n)$$

Chapter 7

Dynamic Programming

Fibonacci sequence (1)

- 0,1,1,2,3,5,8,13,21,34,...
- Leonardo Fibonacci (1170 -1250)

  用來計算兔子的數量

  每對每個月可以生產一對

  兔子出生後, 隔一個月才會生產, 且永不死亡

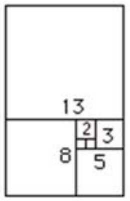
  生產　0 1 1 2 3 ...

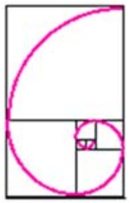
  總數 1 1 2 3 5 8 ...

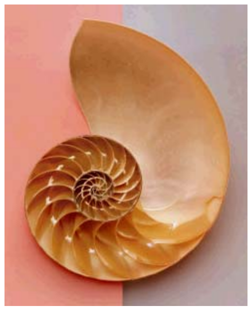
http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibnat.html

Fibonacci sequence (2)

- 0,1,1,2,3,5,8,13,21,34,...





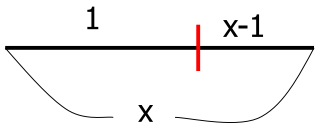


Fibonacci sequence and golden number

- 0,1,1,2,3,5,8,13,21,34,...

$$\begin{cases} f_n = 0 & \text{if } n = 0 \\ f_n = 1 & \text{if } n = 1 \\ f_n = f_{n-1} + f_{n-2} & \text{if } n \geq 2 \end{cases}$$

$$\lim_{n\to\infty} \frac{f_n}{f_{n-1}} = \frac{1+\sqrt{5}}{2} = \text{Golden number}$$



$$\frac{x}{1} = \frac{1}{x-1}$$

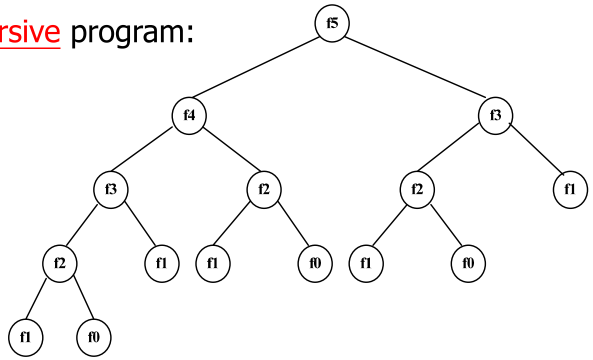
$$x^2 - x - 1 = 0$$

$$x = \frac{1+\sqrt{5}}{2}$$

Computation of Fibonacci sequence

$$f_n = 0 \quad \text{if } n = 0$$
$$f_n = 1 \quad \text{if } n = 1$$
$$f_n = f_{n-1} + f_{n-2} \quad \text{if } n \geq 2$$

- Solved by a recursive program:



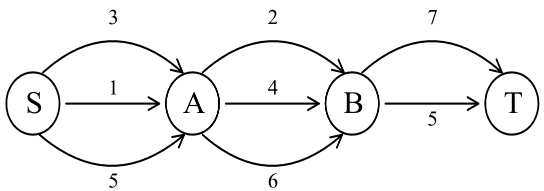
- Much replicated computation is done.
- It should be solved by a simple loop.

Dynamic Programming

- Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions

The shortest path

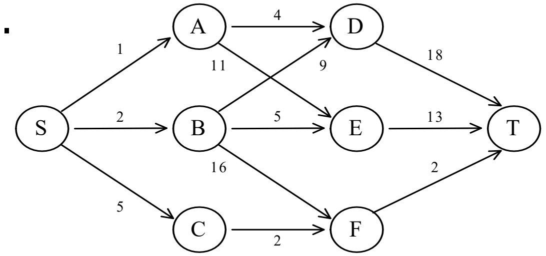
- To find a shortest path in a multi-stage graph



- Apply the greedy method :
  the shortest path from S to T :
  1 + 2 + 5 = 8

The shortest path in multistage graphs

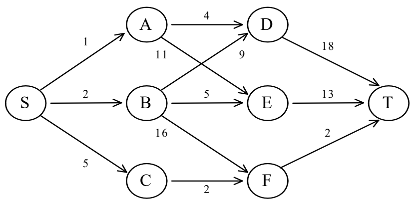
- e.g.

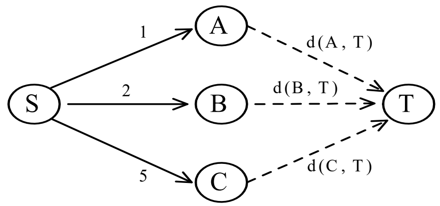


- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:
  (S, C, F, T) 5+2+2 = 9.

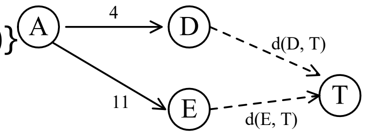
Dynamic programming approach

- Dynamic programming approach (forward approach):

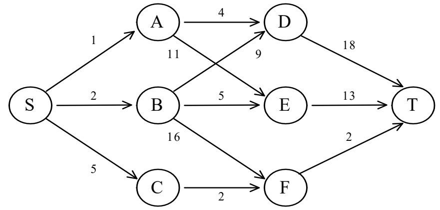


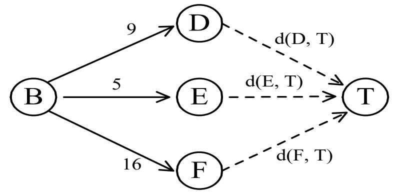


- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
- d(A,T) = min{4+d(D,T), 11+d(E,T)}
  = min{4+18, 11+13} = 22.



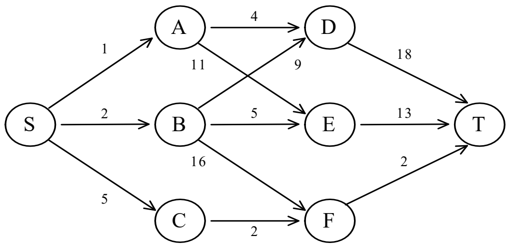
- d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
  = min{9+18, 5+13, 16+2} = 18.





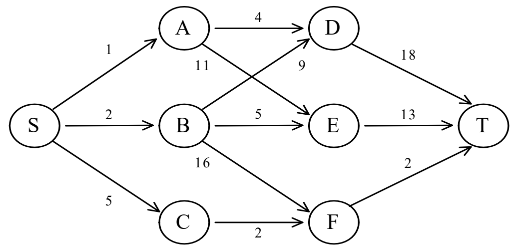
- d(C, T) = min{ 2+d(F, T) } = 2+2 = 4
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
  = min{1+22, 2+18, 5+4} = 9.
- The above way of reasoning is called backward reasoning.

Backward approach (forward reasoning)



- d(S, A) = 1
  d(S, B) = 2
  d(S, C) = 5
- d(S,D)=min{d(S,A)+d(A,D), d(S,B)+d(B,D)}
  = min{ 1+4, 2+9 } = 5
  d(S,E)=min{d(S,A)+d(A,E), d(S,B)+d(B,E)}
  = min{ 1+11, 2+5 } = 7
  d(S,F)=min{d(S,B)+d(B,F), d(S,C)+d(C,F)}
  = min{ 2+16, 5+2 } = 7

- d(S,T) = min{d(S, D)+d(D, T), d(S,E)+
  d(E,T), d(S, F)+d(F, T)}
  = min{ 5+18, 7+13, 7+2 }
  = 9



Principle of optimality

- Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions $D_1, D_2, \ldots, D_n$. If this sequence is optimal, then the last k decisions, $1 < k < n$ must be optimal.
- e.g. the shortest path problem

  If $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i_1 to j
- In summary, if a problem can be described by a multistage graph, then it can be solved by dynamic programming.

Dynamic programming

- Forward approach and backward approach:
  - Note that if the recurrence relations are formulated using the forward approach then the relations are solved backwards . i.e., beginning with the last decision
  - On the other hand if the relations are formulated using the backward approach, they are solved forwards.
- To solve a problem by using dynamic programming:
  - Find out the recurrence relations.
  - Represent the problem by a multistage graph.

The resource allocation problem

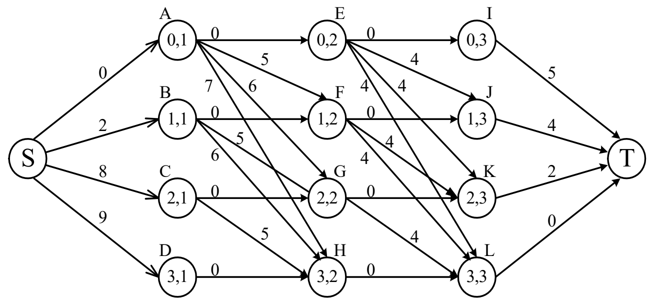
- m resources, n projects

  profit $P_{i,j}$: j resources are allocated to project i.

  maximize the total profit.

Resource / Project	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

The multistage graph solution



- The resource allocation problem can be described as a multistage graph.
- (i, j) : i resources allocated to projects 1, 2, …, j

e.g. node H=(3, 2) : 3 resources allocated to projects 1, 2.

- Find the longest path from S to T :

  (S, C, H, L, T), 8+5+0+0=13

  2 resources allocated to project 1.

  1 resource allocated to project 2.

  0 resource allocated to projects 3, 4.

The longest common subsequence (LCS) problem

- A string : A = b a c a d
- A subsequence of A: deleting 0 or more symbols from A (not necessarily consecutive).

e.g. ad, ac, bac, acad, bacad, bcd.

- Common subsequences of A = b a c a d and

  B = a c c b a d c b : ad, ac, bac, acad.
- The longest common subsequence (LCS) of A and B:

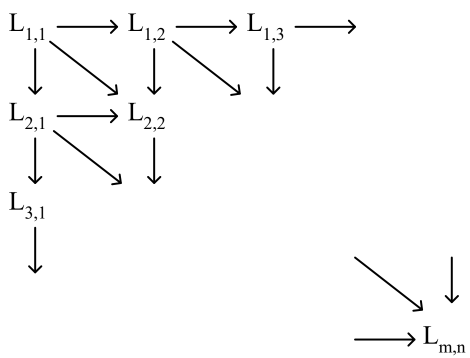
  a c a d.

The LCS algorithm

- Let A = $a_1\ a_2 \dots a_m$ and B = $b_1\ b_2 \dots b_n$
- Let $L_{i,j}$ denote the length of the longest common subsequence of $a_1\ a_2 \dots a_i$ and $b_1\ b_2 \dots b_j$.
- $$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & \text{if } a_i = b_j \\ \max\{ L_{i-1,j}, L_{i,j-1} \} & \text{if } a_i \neq b_j \end{cases}$$

  $L_{0,0} = L_{0,j} = L_{i,0} = 0$ for $1 \le i \le m$, $1 \le j \le n$.

- The dynamic programming approach for solving the LCS problem:



- Time complexity: O(mn)

Tracing back in the LCS algorithm

- e.g. A = b a c a d, B = a c c b a d c b

						B				
			a	c	c	b	a	d	c	b
		0	0	0	0	0	0	0	0	0
	b	0	0	0	0	1	1	1	1	1
	a	0	1	←1	1	1	2	2	2	2
A	c	0	1	2	2	←2	2	2	3	3
	a	0	1	2	2	2	3	3	3	3
	d	0	1	2	2	2	3	4	←4	←4

- After all $L_{i,j}$'s have been found, we can trace back to find the longest common subsequence of A and B.

0/1 knapsack problem

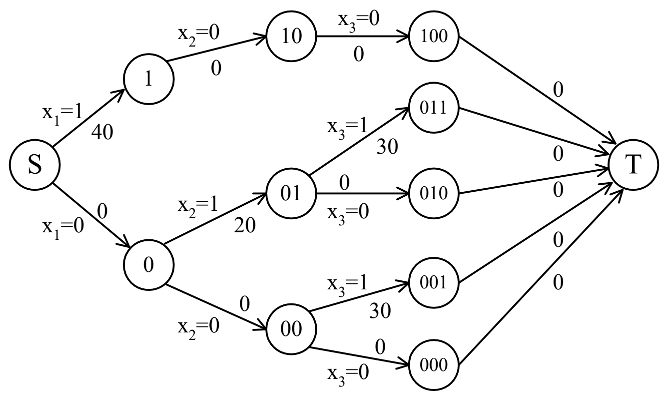
- n objects , weight $W_1, W_2, \dots, W_n$
  profit $P_1, P_2, \dots, P_n$
  capacity M
  maximize $\sum_{1 \le i \le n} P_i x_i$
  subject to $\sum_{1 \le i \le n} W_i x_i \le$ M
  x_i = 0 or 1, $1 \le i \le n$
- e. g.

i	W_i	P_i
1	10	40
2	3	20
3	5	30

M=10

The multistage graph solution

- The 0/1 knapsack problem can be described by a multistage graph.

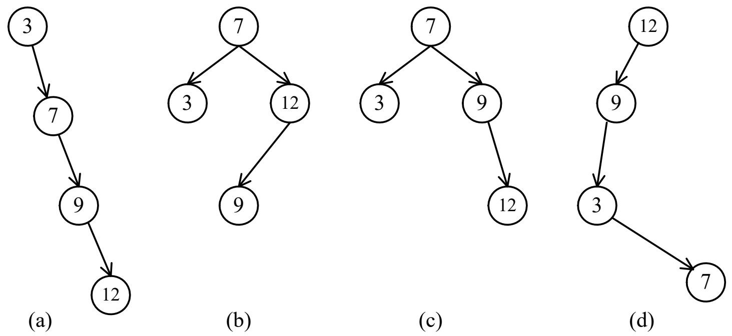


The dynamic programming approach

- The longest path represents the optimal solution:
  x_1=0, x_2=1, x_3=1
  $\sum P_i x_i$ = 20+30 = 50
- Let $f_i(Q)$ be the value of an optimal solution to objects 1,2,3,…,i with capacity Q.
- $f_i(Q) = \max\{ f_{i-1}(Q), f_{i-1}(Q-W_i)+P_i \}$
- The optimal solution is $f_n(M)$.

Optimal binary search trees

- e.g. binary search trees for 3, 7, 9, 12;



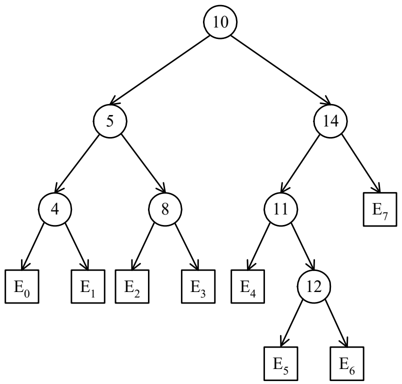
Optimal binary search trees

- n identifiers : $a_1 < a_2 < a_3 < \ldots < a_n$

  P_i, $1 \le i \le n$: the probability that a_i is searched.

  Q_i, $0 \le i \le n$: the probability that x is searched where $a_i < x < a_{i+1}$ ($a_0 = -\infty$, $a_{n+1} = \infty$).

$$\sum_{i=1}^{n} P_i + \sum_{i=0}^{n} Q_i = 1$$



- Identifiers : 4, 5, 8, 10, 11, 12, 14
- Internal node : successful search, P_i
- External node : unsuccessful search, Q_i

- The expected cost of a binary tree:

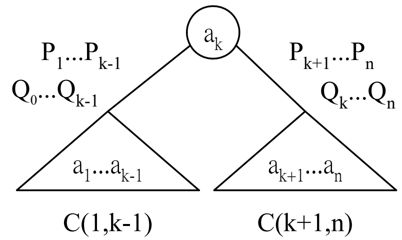
$$\sum_{n=1}^{n} P_i * \text{level}(a_i) + \sum_{n=0}^{n} Q_i * (\text{level}(E_i) - 1)$$

- The level of the root : 1

The dynamic programming approach

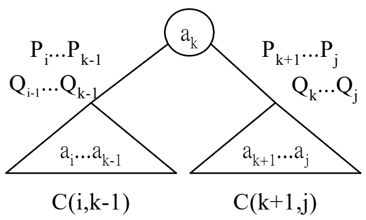
- Let C(i, j) denote the cost of an optimal binary search tree containing $a_i, \ldots, a_j$.
- The cost of the optimal binary search tree with a_k as its root :

$$C(1,n) = \min_{1 \le k \le n} \left\{ P_k + \left[Q_0 + \sum_{m=1}^{k-1} (P_m + Q_m) + C(1,k-1) \right] + \left[Q_k + \sum_{m=k+1}^{n} (P_m + Q_m) + C(k+1,n) \right] \right\}$$



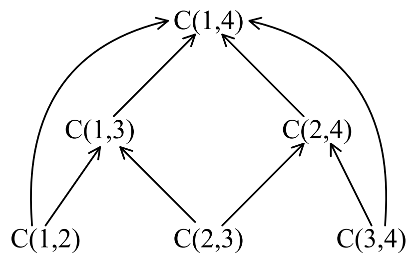
General formula

$$
\begin{aligned}
C(i,j) &= \min_{i \le k \le j}\left\{ P_k + \left[Q_{i-1} + \sum_{m=i}^{k-1}(P_m + Q_m) + C(i,k-1) \right] \right. \\
&\qquad \left. + \left[Q_k + \sum_{m=k+1}^{j}(P_m + Q_m) + C(k+1,j) \right] \right\} \\
&= \min_{i \le k \le j}\left\{ C(i,k-1) + C(k+1,j) + Q_{i-1} + \sum_{m=i}^{j}(P_m + Q_m) \right\}
\end{aligned}
$$



Computation relationships of subtrees

- e.g. n=4



- Time complexity : $O(n^3)$
  (n-m) C(i, j)'s are computed when j-i=m.
  Each C(i, j) with j-i=m can be computed in O(m) time.
  $$O\Big(\sum_{1 \le m \le n} m(n-m)\Big) = O(n^3)$$

Matrix-chain multiplication

- n matrices $A_1, A_2, \ldots, A_n$ with size
  $p_0 \times p_1, p_1 \times p_2, p_2 \times p_3, \ldots, p_{n-1} \times p_n$
  To determine the multiplication order such that # of scalar multiplications is minimized.
- To compute $A_i \times A_{i+1}$, we need $p_{i-1}p_ip_{i+1}$ scalar multiplications.

e.g. n=4, A_1: 3×5, A_2: 5×4, A_3: 4×2, A_4: 2×5

$((A_1 \times A_2) \times A_3) \times A_4$, # of scalar multiplications:
3 * 5 * 4 + 3 * 4 * 2 + 3 * 2 * 5 = 114

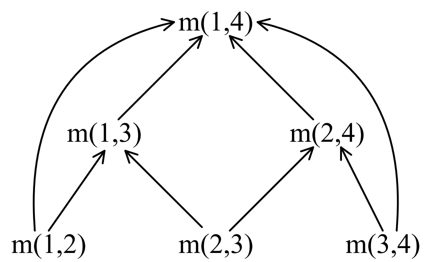
$(A_1 \times (A_2 \times A_3)) \times A_4$, # of scalar multiplications:
3 * 5 * 2 + 5 * 4 * 2 + 3 * 2 * 5 = 100

$(A_1 \times A_2) \times (A_3 \times A_4)$, # of scalar multiplications:
3 * 5 * 4 + 3 * 4 * 5 + 4 * 2 * 5 = 160

- Let m(i, j) denote the minimum cost for computing $A_i \times A_{i+1} \times \ldots \times A_j$

$$
m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k \le j-1} \{ m(i,k) + m(k+1,j) + p_{i-1}p_kp_j \} & \text{if } i < j \end{cases}
$$

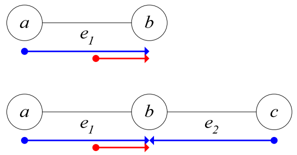
- Computation sequence :

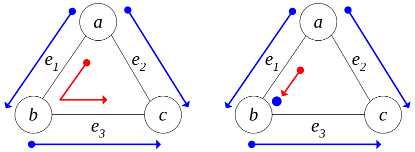


- Time complexity : $O(n^3)$

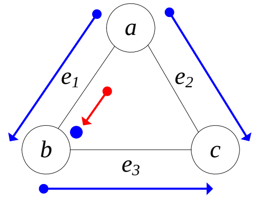
Single step graph edge searching

- fugitive: can move in any speed and is hidden in some edge of an undirected graph G=(V,E)
- edge searcher(guard): search an edge (u, v) from u to v, or stay at vertex u to prevent the fugitive passing through u
- Goal: to capture the fugitive in one step.
- no extra guards needed　　　　extra guards needed



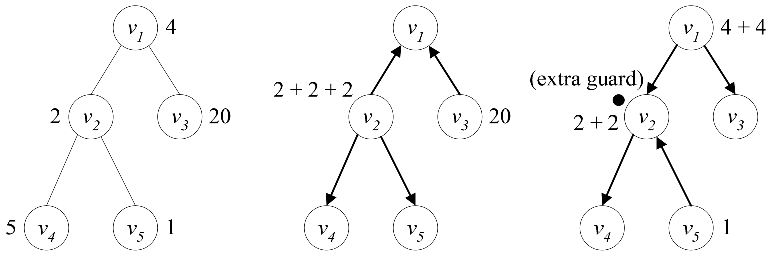


- cost of a searcher from u to v: wt(u)
  a guard staying at u: wt(u)
- Cost of the following: 2wt(a)+wt(b)+wt(b)
  (one extra guard stays at b)



- Problem definition: To arrange the searchers with the minimal cost to capture the fugitive in one step.
- NP-hard for general graphs; P for trees.

The weighted single step graph edge searching problem on trees



Total cost = 26　　　　Total cost = 13

- $T(v_i)$: the tree includes v_i, v_j (parent of v_i) and all descendant nodes of v_i.
- $C(T(v_i), v_i, v_j)$: cost of an optimal searching plan with searching from v_i to v_j.
- $C(T(v_4), v_4, v_2)=5$　　$C(T(v_4), v_2, v_4)=2$
- $C(T(v_2), v_2, v_1)=6$　　$C(T(v_2), v_1, v_2)=9$

The dynamic programming approach

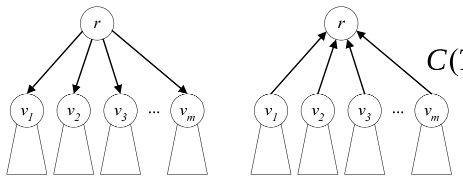
- Rule 1: optimal total cost

$$C(T(r)) = \min\{C(T(r),\bar{r}),\ C(T(r),r)\},$$

where $C(T(r),\bar{r})$: no extra guard at root r

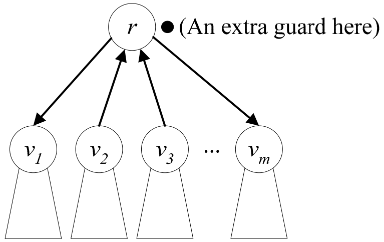
$C(T(r),r)$: one extra guard at root r

- Rule 2.1 : no extra guard at root r: All children must have the same searching direction.



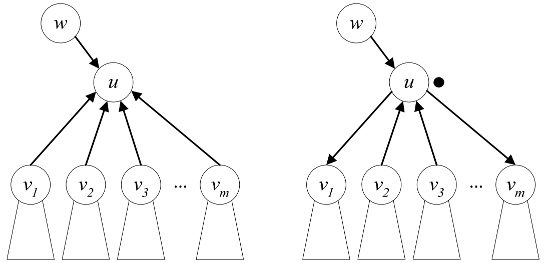
$$C(T(r),\bar{r}) = \min\left\{ \begin{array}{l} \sum_{i=1}^{m} C(T(v_i), r, v_i), \\ \sum_{i=1}^{m} C(T(v_i), v_i, r) \end{array} \right\}$$

- Rule 2.2: one extra guard at root r: Each child can choose its best direction independently.



$$C(T(r),r) = wt(r) + \sum_{i=1}^{m} \min\{C(T(v_i),r,v_i),\ C(T(v_i),v_i,r)\}$$

- Rule 3.1 : Searching to an internal node u from its parent node w

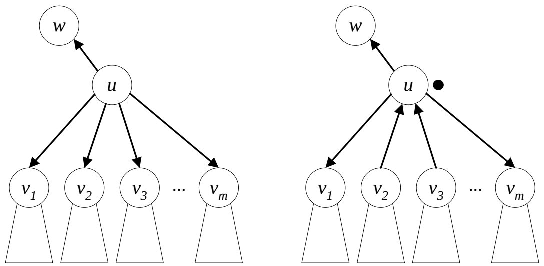


$C(T(u),w,u) = \min\{C(T(u),w,u,\overline{u}), C(T(u),w,u,u)\}$, where

$$\begin{cases} C(T(u),w,u,\overline{u}) = wt(w) + \sum_{i=1}^{m} C(T(v_i),v_i,u) \\ C(T(u),w,u,u) = wt(w) + wt(u) + \sum_{i=1}^{m} \min\{C(T(v_i),v_i,u), C(T(v_i),u,v_i)\} \end{cases}$$

where $\overline{u}$ means no extra guard at u and u means one extra guard at u.

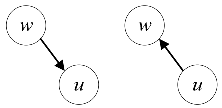
- Rule 3.2 : Searching from an internal node u to its parent node w



$C(T(u),u,w) = \min\{C(T(u),u,w,\overline{u}), C(T(u),u,w,u)\}$, where

$$\begin{cases} C(T(u),u,w,\overline{u}) = wt(u) + \sum_{i=1}^{m} C(T(v_i),u,v_i) \\ C(T(u),u,w,u) = 2wt(u) + \sum_{i=1}^{m} \min\{C(T(v_i),v_i,u), C(T(v_i),u,v_i)\} \end{cases}$$

- Rule 4: A leaf node u and its parent node w.



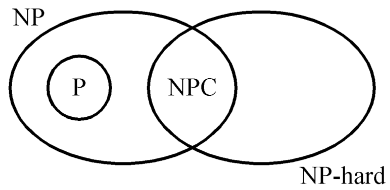
$$C(T(u),w,u) = wt(w)$$
$$C(T(u),u,w) = wt(u)$$

- the dynamic programming approach: working from the leaf nodes and gradually towards the root

- Time complexity : O(n)
  computing minimal cost of each sub-tree and determining searching directions for each edge

Chapter 8

The Theory of NP-Completeness



- **P**: the class of problems which can be solved by a deterministic polynomial algorithm.
- **NP** : the class of decision problem which can be solved by a non-deterministic polynomial algorithm.
- **NP-hard**: the class of problems to which every NP problem reduces.
- **NP-complete (NPC)**: the class of problems which are NP-hard and belong to NP.

Some concepts of NPC

- Definition of reduction: Problem A reduces to problem B ($A \propto B$) iff A can be solved by a deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.
- Up to now, none of the NPC problems can be solved by a deterministic polynomial time algorithm in the worst case.
- It does not seem to have any polynomial time algorithm to solve the NPC problems.

- The theory of NP-completeness always considers the worst case.
- The lower bound of any NPC problem seems to be in the order of an exponential function.
- Not all NP problems are difficult. (e.g. the MST problem is an NP problem.)
- If A, B $\in$ NPC, then $A \propto B$ and $B \propto A$.
- Theory of NP-completeness:

  If any NPC problem can be solved in polynomial time, then all NP problems can be solved in polynomial time. (NP = P)

Decision problems

- The solution is simply "Yes" or "No".
- Optimization problems are more difficult.
- e.g. the traveling salesperson problem
  - Optimization version:
    Find the shortest tour
  - Decision version:
    Is there a tour whose total length is less than or equal to a constant c ?

Solving an optimization problem by a decision algorithm :

- Solving TSP optimization problem by a decision algorithm :
  - Give c_1 and test (decision algorithm)
    Give c_2 and test (decision algorithm)
    ⋮
    Give c_n and test (decision algorithm)
  - We can easily find the smallest c_i

The satisfiability problem

- The satisfiability problem
  - The logical formula :

$$x_1 \vee x_2 \vee x_3$$
$$\& -x_1$$
$$\& -x_2$$

the assignment :

$$x_1 \leftarrow F,\ x_2 \leftarrow F,\ x_3 \leftarrow T$$

will make the above formula true .

$(-x_1, -x_2, x_3)$ represents $x_1 \leftarrow F, x_2 \leftarrow F, x_3 \leftarrow T$

- If there is at least one assignment which satisfies a formula, then we say that this formula is satisfiable; otherwise, it is unsatisfiable.
- An unsatisfiable formula :

$$x_1 \vee x_2$$
$$\& x_1 \vee -x_2$$
$$\& -x_1 \vee x_2$$
$$\& -x_1 \vee -x_2$$

- Definition of the satisfiability problem: Given a Boolean formula, determine whether this formula is satisfiable or not.

- A literal : x_i or $-x_i$
- A clause : $x_1 \vee x_2 \vee -x_3 \equiv C_i$
- A formula : conjunctive normal form (CNF)

$$C_1 \& C_2 \& \ldots \& C_m$$

The resolution principle

- Resolution principle

  $C_1 : x_1 \vee x_2$

  $C_2 : -x_1 \vee x_3$

  $\Rightarrow C_3 : x_2 \vee x_3$
- From $C_1 \& C_2$, we can obtain C_3, and C_3 can be added into the formula.
- The formula becomes:

  $C_1 \& C_2 \& C_3$

x_1	x_2	x_3	$C_1 \& C_2$	C_3
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	1

- Another example of resolution principle

  $C_1 : -x_1 \vee -x_2 \vee x_3$

  $C_2 : x_1 \vee x_4$

  $\Rightarrow C_3 : -x_2 \vee x_3 \vee x_4$
- If no new clauses can be deduced, then it is satisfiable.

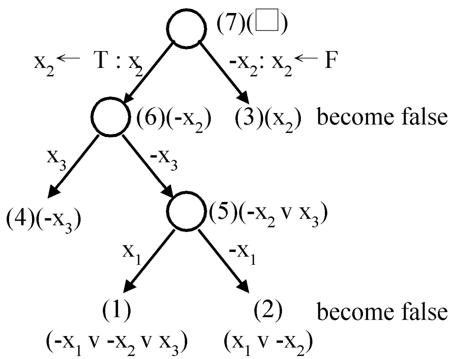
	$-x_1 \vee -x_2 \vee x_3$	(1)
	x_1	(2)
	x_2	(3)
(1) & (2)	$-x_2 \vee x_3$	(4)
(4) & (3)	x_3	(5)
(1) & (3)	$-x_1 \vee x_3$	(6)

- If an empty clause is deduced, then it is unsatisfiable.

	$-x_1 \vee -x_2 \vee x_3$	(1)
	$x_1 \vee -x_2$	(2)
	x_2	(3)
	$-x_3$	(4)
	$\Downarrow$ deduce	
(1) & (2)	$-x_2 \vee x_3$	(5)
(4) & (5)	$-x_2$	(6)
(6) & (3)	$\square$	(7)

Semantic tree

- In a semantic tree, each path from the root to a leaf node represents a class of assignments.
- If each leaf node is attached with a clause, then it is unsatisfiable.



Nondeterministic algorithms

- A nondeterminstic algorithm consists of
  phase 1: guessing
  phase 2: checking
- If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.
- NP problems : (must be decision problems)
  - e.g. searching, MST
    sorting
    satisfiability problem (SAT)
    traveling salesperson problem (TSP)

Decision problems

- Decision version of sorting:
  Given $a_1, a_2, \ldots, a_n$ and c, is there a permutation of a_i's ($a_1', a_2', \ldots, a_n'$) such that $|a_2'-a_1'| + |a_3'-a_2'| + \ldots + |a_n'-a_{n-1}'| < c$?
- Not all decision problems are NP problems
  - E.g. halting problem :
    - Given a program with a certain input data, will the program terminate or not?
    - NP-hard
    - Undecidable

Nondeterministic operations and functions

[Horowitz 1998]

- Choice(S) : arbitrarily chooses one of the elements in set S
- Failure : an unsuccessful completion
- Success : a successful completion
- Nonderministic searching algorithm:

```
j ← choice(1 : n)   /* guessing */
if A(j) = x then success  /* checking */
          else failure
```

- A nondeterministic algorithm terminates unsuccessfully iff there does not exist a set of choices leading to a success signal.
- The time required for *choice*(*1 : n*) is O(1).
- A deterministic interpretation of a non-deterministic algorithm can be made by allowing unbounded parallelism in computation.

Nondeterministic sorting

```
B ← 0
/* guessing */
for i = 1 to n do
    j ← choice(1 : n)
    if B[j] ≠ 0 then failure
    B[j] = A[i]
/* checking */
for i = 1 to n-1 do
   if B[i] > B[i+1] then failure
   success
```


Nondeterministic SAT

/* guessing */

for i = 1 to n do

   x_i ← choice(true, false)

/* checking */

if $E(x_1, x_2, \dots, x_n)$ is true then success

else failure

Cook's theorem

NP = P iff the satisfiability problem is a P problem.

- SAT is NP-complete.
- It is the first NP-complete problem.
- Every NP problem reduces to SAT.

Stephen Arthur Cook

(1939~)

Transforming searching to SAT

- Does there exist a number in { x(1), x(2), …, x(n) }, which is equal to 7?
- Assume n = 2.

  nondeterministic algorithm:

```
i = choice(1,2)

if x(i)=7 then   SUCCESS

else FAILURE
```


i=1 v i=2
& i=1 → i ≠ 2
& i=2 → i ≠ 1
& x(1)=7 & i=1 → SUCCESS
& x(2)=7 & i=2 → SUCCESS
& x(1) ≠ 7 & i=1 → FAILURE
& x(2) ≠ 7 & i=2 → FAILURE
& FAILURE → -SUCCESS
& SUCCESS (Guarantees a successful termination)
& x(1)=7 (Input Data)
& x(2) ≠ 7

- CNF (conjunctive normal form) :

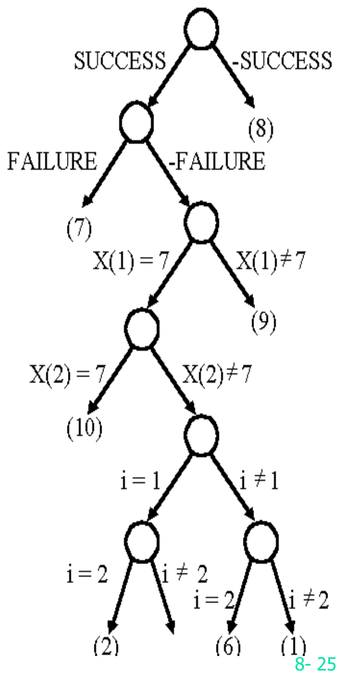
i=1 v i=2	(1)
i ≠ 1 v i ≠ 2	(2)
x(1) ≠ 7 v i ≠ 1 v SUCCESS	(3)
x(2) ≠ 7 v i ≠ 2 v SUCCESS	(4)
x(1)=7 v i ≠ 1 v FAILURE	(5)
x(2)=7 v i ≠ 2 v FAILURE	(6)
-FAILURE v -SUCCESS	(7)
SUCCESS	(8)
x(1)=7	(9)
x(2) ≠ 7	(10)

- Satisfiable at the following assignment :

i=1	satisfying	(1)
i ≠ 2	satisfying	(2), (4) and (6)
SUCCESS	satisfying	(3), (4) and (8)
-FAILURE	satisfying	(7)
x(1)=7	satisfying	(5) and (9)
x(2) ≠ 7	satisfying	(4) and (10)

The semantic tree

i=1 v i=2	(1)
i≠1 v i≠2	(2)
x(1)≠7 v i≠1 v SUCCESS	(3)
x(2)≠7 v i≠2 v SUCCESS	(4)
x(1)=7 v i≠1 v FAILURE	(5)
x(2)=7 v i≠2 v FAILURE	(6)
-FAILURE v -SUCCESS	(7)
SUCCESS	(8)
x(1)=7	(9)
x(2)≠7	(10)



Searching for 7, but x(1)≠7, x(2)≠7

- CNF (conjunctive normal form) :

i=1	v	i=2				(1)
i≠1	v	i≠2				(2)
x(1)≠7	v	i≠1	v	SUCCESS		(3)
x(2)≠7	v	i≠2	v	SUCCESS		(4)
x(1)=7	v	i≠1	v	FAILURE		(5)
x(2)=7	v	i≠2	v	FAILURE		(6)
SUCCESS						(7)
-SUCCESS v -FAILURE						(8)
x(1) ≠ 7						(9)
x(2) ≠ 7						(10)

- Apply resolution principle :

(9) & (5)	i≠1 v FAILURE	(11)
(10) & (6)	i≠2 v FAILURE	(12)
(7) & (8)	-FAILURE	(13)
(13) & (11)	i≠1	(14)
(13) & (12)	i≠2	(15)
(14) & (1)	i=2	(11)
(15) & (16)	□	(17)

We get an empty clause ⇒ unsatisfiable

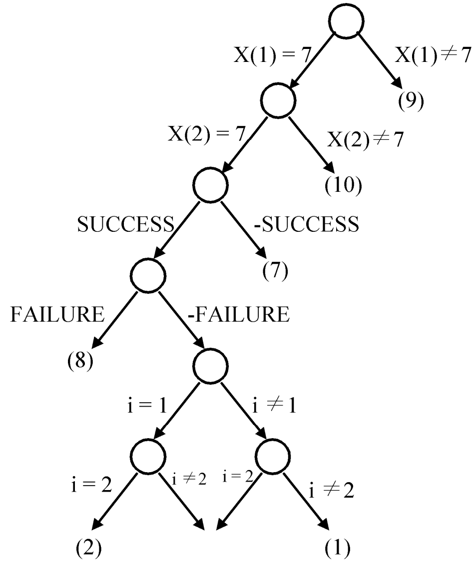
⇒ 7 does not exit in x(1) or x(2).

Searching for 7, where x(1)=7, x(2)=7

- CNF:

i=1	v	i=2			(1)
i≠1	v	i≠2			(2)
x(1)≠7	v	i≠1	v	SUCCESS	(3)
x(2)≠7	v	i≠2	v	SUCCESS	(4)
x(1)=7	v	i≠1	v	FAILURE	(5)
x(2)=7	v	i≠2	v	FAILURE	(6)
SUCCESS					(7)
-SUCCESS v -FAILURE					(8)
x(1)=7					(9)
x(2)=7					(10)

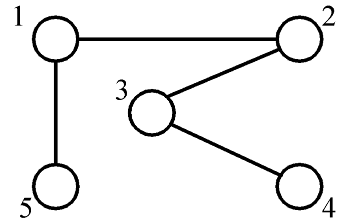
The semantic tree



It implies that both assignments (i=1, i=2) satisfy the clauses.

The node cover problem

- **Def**: Given a graph G=(V, E), S is the node cover if $S \subseteq V$ and for every edge $(u, v) \in E$, either $u \in S$ or $v \in S$.



node cover :
{1, 3}
{5, 2, 4}

- Decision problem : $\exists S \ni |S| \leq K$?

Transforming the node cover problem to SAT

BEGIN
$i_1 \leftarrow$ choice({1, 2, …, n})
$i_2 \leftarrow$ choice({1, 2, …, n} − {i_1})
⋮
$i_k \leftarrow$ choice({1, 2, …, n} − {$i_1, i_2, …, i_{k-1}$}).
For j=1 to m do
BEGIN
if e_j is not incident to one of v_{i_t} $(1 \leq t \leq k)$
then FAILURE
END
SUCCESS

CNF:

$i_1 = 1 \ \vee \ i_1 = 2 \ldots \ \vee \ i_1 = n$
$(i_1 \neq 1 \rightarrow i_1 = 2 \ \vee \ i_1 = 3 \ldots \vee \ i_1 = n)$
$i_2 = 1 \ \vee \ i_2 = 2 \ldots \ \vee \ i_2 = n$
⋮
$i_k = 1 \ \vee \ i_k = 2 \ldots \ \vee \ i_k = n$

$i_1 \neq 1 \ \vee \ i_2 \neq 1$ $(i_1=1 \rightarrow i_2 \neq 1 \ \& \ldots \& \ i_k \neq 1)$
$i_1 \neq 1 \ \vee \ i_3 \neq 1$
⋮
$i_{k-1} \neq n \ \vee \ i_k \neq n$
$v_{i_1} \in e_1 \ \vee \ v_{i_2} \in e_1 \ \vee \ldots \vee \ v_{i_k} \in e_1 \ \vee$ FAILURE
$(v_{i_1} \notin e_1 \ \& \ v_{i_2} \notin e_1 \ \& \cdots \& \ v_{i_k} \notin e_1 \rightarrow \text{Failure})$
$v_{i_1} \in e_2 \ \vee \ v_{i_2} \in e_2 \ \vee \ldots \vee \ v_{i_k} \in e_2 \ \vee$ FAILURE
⋮
$v_{i_1} \in e_m \ \vee \ v_{i_2} \in e_m \ \vee \ldots \vee \ v_{i_k} \in e_m \ \vee$ FAILURE
SUCCESS

(To be continued)

-SUCCESS v -FAILURE

$\nu_{r_1} \in e_1$

$\nu_{s_1} \in e_1$

$\nu_{r_2} \in e_2$

$\nu_{s_2} \in e_2$

$\vdots$

$\nu_{r_m} \in e_m$

$\nu_{s_m} \in e_m$

SAT is NP-complete

(1) SAT has an NP algorithm.

(2) SAT is NP-hard:

- Every NP algorithm for problem *A* can be transformed in polynomial time to SAT [Horowitz 1998] such that SAT is satisfiable if and only if the answer for *A* is "YES".
- That is, every NP problem $\propto$ SAT .

- By (1) and (2), SAT is NP-complete.

Proof of NP-Completeness

- To show that A is NP-complete

  (I) Prove that A is an NP problem.

  (II) Prove that $\exists$ B $\in$ NPC, B $\propto$ A.

  $\Rightarrow$ A $\in$ NPC
- Why ?

3-satisfiability problem (3-SAT)

- **Def**: Each clause contains exactly three literals.
- (I) 3-SAT is an NP problem (obviously)
- (II) SAT $\propto$ 3-SAT

  Proof:

  (1) One literal L_1 in a clause in SAT :

  in 3-SAT :

  $L_1 \vee y_1 \vee y_2$

  $L_1 \vee -y_1 \vee y_2$

  $L_1 \vee y_1 \vee -y_2$

  $L_1 \vee -y_1 \vee -y_2$

(2) Two literals L_1, L_2 in a clause in SAT :

in 3-SAT :

$L_1 \vee L_2 \vee y_1$

$L_1 \vee L_2 \vee -y_1$

(3) Three literals in a clause : remain unchanged.

(4) More than 3 literals $L_1, L_2, \ldots, L_k$ in a clause :

in 3-SAT :

$L_1 \vee L_2 \vee y_1$

$L_3 \vee -y_1 \vee y_2$

$\vdots$

$L_{k-2} \vee -y_{k-4} \vee y_{k-3}$

$L_{k-1} \vee L_k \vee -y_{k-3}$

Example of transforming SAT to 3-SAT

- An instance S in SAT :

$x_1 \vee x_2$

$-x_3$

$x_1 \vee -x_2 \vee x_3 \vee -x_4 \vee x_5 \vee x_6$

- The instance S′ in 3-SAT :

$x_1 \vee x_2 \vee y_1$

$x_1 \vee x_2 \vee -y_1$

$-x_3 \vee y_2 \vee y_3$

$-x_3 \vee -y_2 \vee y_3$

$-x_3 \vee y_2 \vee -y_3$

$-x_3 \vee -y_2 \vee -y_3$

$x_1 \vee -x_2 \vee y_4$

$x_3 \vee -y_4 \vee y_5$

$-x_4 \vee -y_5 \vee y_6$

$x_5 \vee x_6 \vee -y_6$

SAT S $\xrightarrow{\textbf{transform}}$ 3-SAT S′

- Proof : S is satisfiable $\Leftrightarrow$ S′ is satisfiable

"$\Rightarrow$"

$\leq$ 3 literals in S (trivial)

consider $\geq$ 4 literals

S : $L_1 \vee L_2 \vee \ldots \vee L_k$

S′: $L_1 \vee L_2 \vee y_1$

$L_3 \vee -y_1 \vee y_2$

$L_4 \vee -y_2 \vee y_3$

$\vdots$

$L_{k-2} \vee -y_{k-4} \vee y_{k-3}$

$L_{k-1} \vee L_k \vee -y_{k-3}$

- S is satisfiable $\Rightarrow$ at least L_i = T

Assume : $L_j = F \quad \forall j \neq i$

assign : $y_{i-1} = F$

$y_j = T \quad \forall j < i-1$

$y_j = F \quad \forall j > i-1$

$(\because L_i \vee -y_{i-2} \vee y_{i-1})$

$\Rightarrow$ S′ is satisfiable.

- "$\Leftarrow$"

If S′ is satisfiable, then assignment satisfying S′ can not contain y_i's only.

$\Rightarrow$ at least one L_i must be true.

(We can also apply the resolution principle).

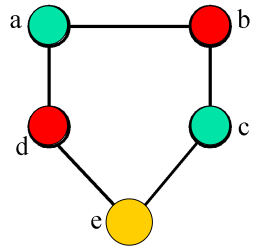
Thus, 3-SAT is NP-complete.

Comment for 3-SAT

- If a problem is NP-complete, its special cases may or may not be NP-complete.

Chromatic number decision problem (CN)

- **Def:** A coloring of a graph G=(V, E) is a function $f : V \rightarrow \{ 1, 2, 3, \ldots, k \}$ such that if $(u, v) \in E$, then $f(u) \neq f(v)$. The CN problem is to determine if G has a coloring for k.



3-colorable
$f(a)=1,\ f(b)=2,\ f(c)=1$
$f(d)=2,\ f(e)=3$

<Theorem> Satisfiability with at most 3 literals per clause (SATY) $\propto$ CN.

SATY $\propto$ CN

Proof :

instance of SATY :

variable : $x_1, x_2, \ldots, x_n$, $n \geq 4$

clause : $c_1, c_2, \ldots, c_r$

instance of CN :

G=(V, E)

$$V=\{ x_1, x_2, \ldots, x_n \} \cup \{ -x_1, -x_2, \ldots, -x_n \} \cup \underbrace{\{ y_1, y_2, \ldots, y_n \}}_{\text{newly added}} \cup \{ c_1, c_2, \ldots, c_r \}$$

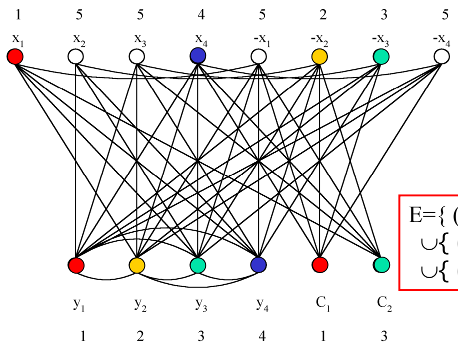
$$E=\{ (x_i, -x_i) \mid 1 \leq i \leq n \} \cup \{ (y_i, y_j) \mid i \neq j \} \cup \{ (y_i, x_j) \mid i \neq j \} \cup \{ (y_i, -x_j) \mid i \neq j \} \cup \{ (x_i, c_j) \mid x_i \notin c_j \} \cup \{ (-x_i, c_j) \mid -x_i \notin c_j \}$$

Example of SATY $\propto$ CN

$x_1 \vee x_2 \vee x_3$ (1)

$-x_3 \vee -x_4 \vee x_2$ (2)

$\Downarrow$



True assignment:

x_1=T

x_2=F

x_3=F

x_4=T

$$E=\{ (x_i, -x_i) \mid 1 \leq i \leq n \} \cup \{ (y_i, y_j) \mid i \neq j \} \cup \{ (y_i, x_j) \mid i \neq j \} \cup \{ (y_i, -x_j) \mid i \neq j \} \cup \{ (x_i, c_j) \mid x_i \notin c_j \} \cup \{ (-x_i, c_j) \mid -x_i \notin c_j \}$$

Proof of SATY $\propto$ CN

- Satisfiable $\Leftrightarrow$ n+1 colorable
  - "$\Rightarrow$"

    (1) $f(y_i) = i$

    (2) if x_i = T, then $f(x_i) = i$, $f(-x_i) = n+1$

    else $f(x_i) = n+1$, $f(-x_i) = i$

    (3)if x_i in c_j and x_i = T, then $f(c_j) = f(x_i)$

    if $-x_i$ in c_j and $-x_i$ = T, then $f(c_j) = f(-x_i)$

    (at least one such x_i)

- "$\Leftarrow$"

  (1) y_i must be assigned with color i.

  (2) $f(x_i) \neq f(-x_i)$

  either $f(x_i) = i$ and $f(-x_i) = n+1$

  or $f(x_i) = n+1$ and $f(-x_i) = i$

  (3) at most 3 literals in c_j and $n \geq 4$

  $\Rightarrow$ at least one x_i, $\ni$ x_i and $-x_i$ are not in c_j

  $\Rightarrow f(c_j) \neq n+1$

  (4) if $f(c_j) = i = f(x_i)$, assign x_i to T

  if $f(c_j) = i = f(-x_i)$, assign $-x_i$ to T

  (5) if $f(c_j) = i = f(x_i) \Rightarrow (c_j, x_i) \notin E$

  $\Rightarrow x_i$ in $c_j \Rightarrow c_j$ is true

  if $f(c_j) = i = f(-x_i) \Rightarrow$ similarly

Set cover decision problem

- **Def**: $F = \{S_i\} = \{ S_1, S_2, \ldots, S_k \}$

  $\bigcup_{S_i \in F} S_i = \{ u_1, u_2, \ldots, u_n \}$

T is a set cover of F if $T \subseteq F$ and $\bigcup_{S_i \in T} S_i = \bigcup_{S_i \in F} S_i$

The set cover decision problem is to determine if F has a cover T containing no more than *c* sets.

- Example

  $F = \{(u_1, u_3), (u_2, u_4), (u_2, u_3), (u_4), (u_1, u_3, u_4)\}$

  $\quad s_1 \quad s_2 \quad s_3 \quad s_4 \quad s_5$

  $T = \{ s_1, s_3, s_4 \}$ set cover

  $T = \{ s_1, s_2 \}$ set cover, exact cover

Exact cover problem

(Notations same as those in set cover.)

Def: To determine if F has an exact cover T, which is a cover of F and the sets in T are pairwise disjoint.

<Theorem> CN $\propto$ exact cover

(No proof here.)

Sum of subsets problem

- **Def**: A set of positive numbers A = { a_1, a_2, …, a_n }
  a constant C
  Determine if ∃ A′ ⊆ A ∋ $\sum_{a_i \in A'} a_i = C$
- e.g. A = { 7, 5, 19, 1, 12, 8, 14 }
  - C = 21, A′ = { 7, 14 }
  - C = 11, no solution

<Theorem> Exact cover ∝ sum of subsets.

Exact cover ∝ sum of subsets

- Proof :

instance of exact cover :

F = { S_1, S_2, …, S_k } $\bigcup_{S_i \in F} S_i = \{u_1, u_2, \ldots, u_n\}$

instance of sum of subsets :

A = { a_1, a_2, …, a_k } where

$a_i = \sum_{1 \le j \le n} e_{ij}(k+1)^j$ where e_{ij} = 1 if $u_j \in S_i$

e_{ij} = 0 if otherwise.

$$C = \sum_{1 \le j \le n} (k+1)^j = (k+1)((k+1)^n - 1)/k$$

- Why k+1? (See the example on the next page.)

Example of Exact cover ∝ sum of subsets

- Valid transformation:

  u_1=6, u_2=8, u_3=9, n=3

  EC: S_1={6,8}, S_2={9}, S_3={6,9}, S_4={8,9}

  $\bigcup_{S_i \in F} S_i = \{u_1, u_2, \ldots, u_n\} = \{6,8,9\}$

  k=4

  SS: $a_1=5^1+5^2=30$

  $a_2=5^3=125$

  $a_3=5^1+5^3=130$

  $a_4=5^2+5^3=150$

  $C=5^1+5^2+5^3=155$

- Invalid transformation:

  EC: S_1={6,8}, S_2={8}, S_3={8}, S_4={8,9}. K=4

  Suppose k-2=2 is used.

  SS: $a_1=2^1+2^2=6$

  $a_2=2^2=4$

  $a_3=2^2=4$

  $a_4=2^2+2^3=12$

  $C=2^1+2^2+2^3=14$

Partition problem

- **Def**: Given a set of positive numbers A = { a_1,a_2,…,a_n },
  determine if ∃ a partition P, ∋ $\sum_{a_i \in P} a_i = \sum_{a_i \notin P} a_i$
- e. g. A = {3, 6, 1, 9, 4, 11}
  partition : {3, 1, 9, 4} and {6, 11}

<Theorem> sum of subsets ∝ partition

Sum of subsets $\propto$ partition

proof :

instance of sum of subsets :

$A = \{ a_1, a_2, \ldots, a_n \}, C$

instance of partition :

$B = \{ b_1, b_2, \ldots, b_{n+2} \}$, where $b_i = a_i$, $1 \le i \le n$

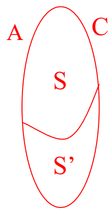
$b_{n+1} = C+1$

$b_{n+2} = (\sum\limits_{1 \le i \le n} a_i) + 1 - C$

$C = \sum\limits_{a_i \in S} a_i \Leftrightarrow (\sum\limits_{a_i \in S} a_i) + b_{n+2} = (\sum\limits_{a_i \notin S} a_i) + b_{n+1}$

$\Leftrightarrow$ partition : $\{ b_i \mid a_i \in S \} \cup \{ b_{n+2} \}$

and $\{ b_i \mid a_i \notin S \} \cup \{ b_{n+1} \}$



- Why $b_{n+1} = C+1$? why not $b_{n+1} = C$?
  - To avoid b_{n+1} and b_{n+2} to be partitioned into the same subset.

Bin packing problem

- **Def:** n items, each of size c_i , $c_i > 0$
  Each bin capacity : C
  Determine if we can assign the items into k bins, $\ni \sum\limits_{i \in bin_j} c_i \le C$, $1 \le j \le k$.

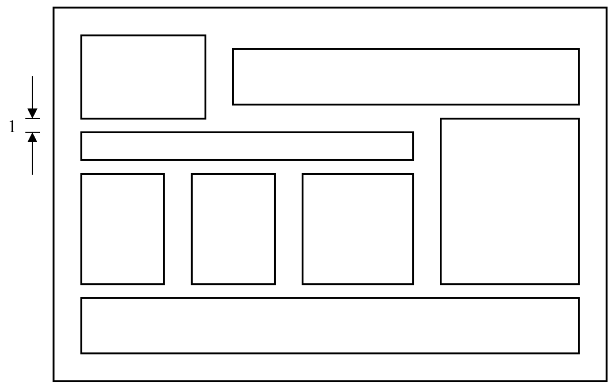
<Theorem> partition $\propto$ bin packing.

VLSI discrete layout problem

- Given: n rectangles, each with height h_i (integer)
  width w_i
  and an area A
  Determine if there is a placement of the n rectangles within the area A according to the rules :

1. Boundaries of rectangles parallel to x axis or y axis.
2. Corners of rectangles lie on integer points.
3. No two rectangles overlap.
4. Two rectangles are separated by at least a unit distance.

(See the figure on the next page.)

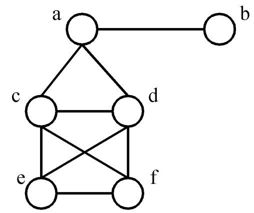


A Successful Placement

<Theorem> bin packing $\propto$ VLSI discrete layout.

Max clique problem

- **Def:** A maximal complete subgraph of a graph G=(V,E) is a clique. The max (maximum) clique problem is to determine the size of a largest clique in G.
- e. g.



maximal cliques :
{a, b}, {a, c, d}
{c, d, e, f}
maximum clique :
(largest)
{c, d, e, f}

<Theorem> SAT $\propto$ clique decision problem.

Node cover decision problem

- **Def:** A set $S \subseteq V$ is a node cover for a graph G = (V, E) iff all edges in E are incident to at least one vertex in S. $\exists$ S, $\ni$ $|S| \leq K$?

<Theorem> clique decision problem $\propto$ node cover decision problem.

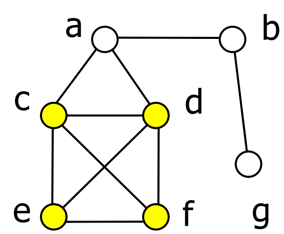
(See proof on the next page.)

Clique decision $\propto$ node cover decision

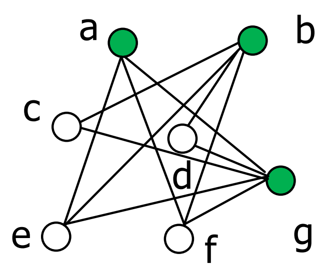
- G=(V,E) : clique Q of size k ($Q \subseteq V$)

G'=(V,E') : node cover S of size n-k, S=V-Q
where $E'=\{(u,v)|u \in V, v \in V \text{ and } (u,v) \notin E\}$

G

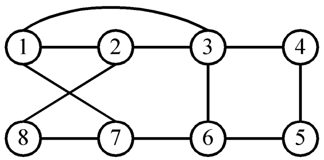


G'



Hamiltonian cycle problem

- **Def:** A Hamiltonian cycle is a round trip path along n edges of G which visits every vertex once and returns to its starting vertex.
- e.g.



Hamiltonian cycle : 1, 2, 8, 7, 6, 5, 4, 3, 1.

<Theorem> SAT $\propto$ directed Hamiltonian cycle (in a directed graph)

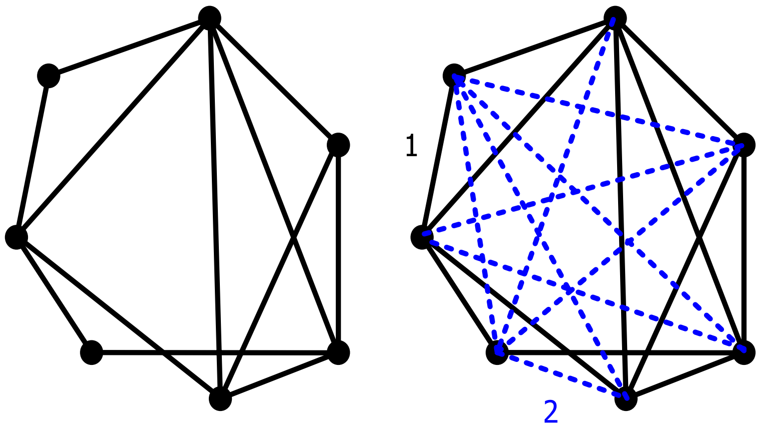
Traveling salesperson problem

- **Def:** A tour of a directed graph G=(V, E) is a directed cycle that includes every vertex in V. The problem is to find a tour of minimum cost.

<Theorem> Directed Hamiltonian cycle $\propto$ traveling salesperson decision problem.

(See proof on the next page.)

Proof of Hamiltonian $\propto$ TSP



0/1 knapsack problem

- **Def:** n objects, each with a weight $w_i > 0$
  a profit $p_i > 0$
  capacity of knapsack : M
  Maximize $\sum_{1\le i\le n} p_i x_i$
  Subject to $\sum_{1\le i\le n} w_i x_i \le M$
  $x_i = 0$ or 1, $1\le i \le n$
- Decision version :
  Given K, $\exists \sum_{1\le i\le n} p_i x_i \ge K$?
- Knapsack problem : $0 \le x_i \le 1$, $1\le i \le n$.

<Theorem> partition $\propto$ 0/1 knapsack decision problem.

- Refer to Sec. 11.3, Sec. 11.4 and its exercises of [Horowitz 1998] for the proofs of more NP-complete problems.
  - [[Horowitz 1998] E. Howowitz, S. Sahni and S. Rajasekaran, *Computer Algorithms*, Computer Science Press, New York, 1998,「台北圖書」代理, 02-23625376

Chapter 9

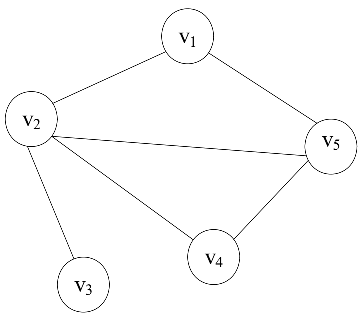
Approximation Algorithms

Approximation algorithm

- Up to now, the best algorithm for solving an NP-complete problem requires exponential time in the worst case. It is too time-consuming.
- To reduce the time required for solving a problem, we can relax the problem, and obtain a feasible solution "close" to an optimal solution

The node cover problem

- **Def**: Given a graph G=(V, E), S is the node cover if $S \subseteq V$ and for every edge $(u, v) \in E$, either $u \in S$ or $v \in S$.



The optimal solution:

$\{v_2, v_5\}$

- The node cover problem is NP-complete.

An approximation algorithm

- Input: A graph G=(V,E).
- Output: A node cover S of G.

Step 1: $S=\phi$ and $E'=E$.

Step 2: While $E' \neq \phi$

Pick an arbitrary edge (a,b) in E'.

$S=S\cup\{a,b\}$.

$E'=E'-\{e| \text{ e is incident to a or b}\}$

- Time complexity: O(|E|)

- Example:

  First: pick (v_2,v_3)

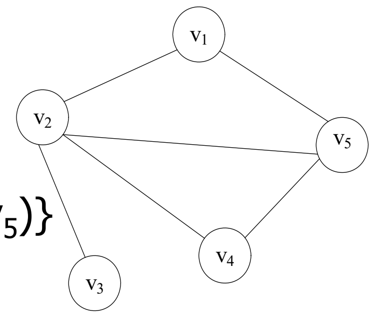
  then $S=\{v_2,v_3\}$

  $E'=\{(v_1,v_5), (v_4,v_5)\}$

  second: pick (v_1,v_5)

  then $S=\{v_1,v_2,v_3,v_5\}$

  $E'=\phi$



How good is the solution ?

- |S| is <u>at most two times</u> the minimum size of a node cover of G.
- L: the number of edges we pick

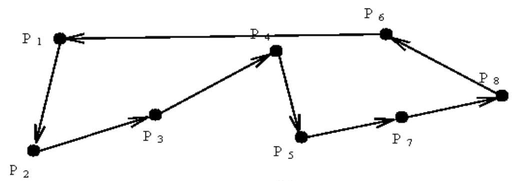
  M*: the size of an optimal solution

(1) $L \le M^*$, because no two edges picked in Step 2 share any same vertex.

(2) $|S| = 2L \le 2M^*$

The Euclidean traveling salesperson problem (ETSP)

- The ETSP is to find a <u>shortest closed path</u> through a set S of n points in the <u>plane</u>.
- The ETSP is NP-hard.



(b)

Fig. 9-8 An Eulerian Cycle and the Resulting Approximate Tour

An approximation algorithm for ETSP

- <u>Input:</u> A set S of n points in the plane.
- <u>Output:</u> An approximate traveling salesperson tour of S.

<u>Step 1:</u> Find a <u>minimal spanning tree</u> T of S.

<u>Step 2:</u> Find a <u>minimal Euclidean weighted matching</u> M on the set of vertices of odd degrees in T. Let $G=M \cup T$.

<u>Step 3:</u> Find an <u>Eulerian cycle</u> of G and then traverse it to find a <u>Hamiltonian cycle</u> as an approximate tour of ETSP by bypassing all previously visited vertices.

An example for ETSP algorithm

- Step1: Find a minimal spanning tree.

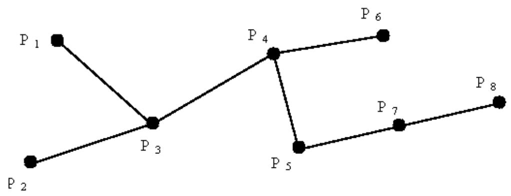


Fig. 9-6 A Minimal Spanning Tree of Eight Points

- Step2: Perform weighted matching. The number of points with odd degrees must be even because $\sum_{i=1}^{n} d_i = 2|E|$ is even.

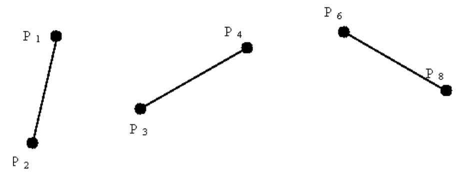
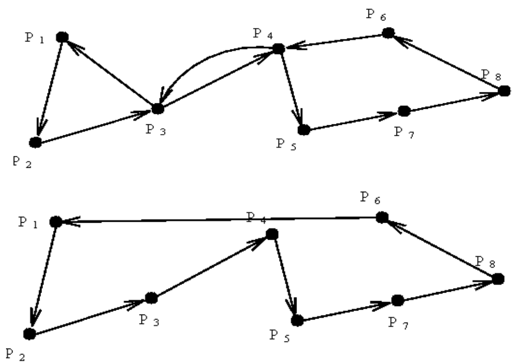


Fig. 9-7 A Minimal Weighted Matching of Six Vertices.

- Step3: Construct the tour with an Eulerian cycle and a Hamiltonian cycle.



(b)

Fig. 9-8 An Eulerian Cycle and the Resulting Approximate Tour

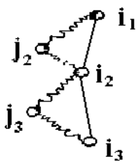
- Time complexity: $O(n^3)$
  Step 1: $O(n\log n)$
  Step 2: $O(n^3)$
  Step 3: $O(n)$

- How close the approximate solution to an optimal solution?
  - The approximate tour is within 3/2 of the optimal one. (The approximate rate is 3/2.)

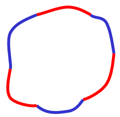
    (See the proof on the next page.)

Proof of approximate rate

- optimal tour L: $j_1 \ldots i_1 j_2 \ldots i_2 j_3 \ldots i_{2m}$
  $\{i_1, i_2, \ldots, i_{2m}\}$: the set of odd degree vertices in T.
  2 matchings: $M_1 = \{[i_1,i_2],[i_3,i_4],\ldots,[i_{2m-1},i_{2m}]\}$
  $M_2 = \{[i_2,i_3],[i_4,i_5],\ldots,[i_{2m},i_1]\}$
  length(L) $\geq$ length(M_1) + length(M_2) (triangular inequality)
  $\geq$ 2 length(M)



$\Rightarrow$ length(M) $\leq$ 1/2 length(L)
G = T $\cup$ M
$\Rightarrow$ length(T) + length(M) $\leq$ length(L) + 1/2 length(L)
= 3/2 length(L)



The bottleneck traveling salesperson problem (BTSP)

- Minimize the longest edge of a tour.
- This is a mini-max problem.
- This problem is NP-hard.
- The input data for this problem fulfill the following assumptions:
  - The graph is a complete graph.
  - All edges obey the triangular inequality rule.

An algorithm for finding an optimal solution

Step1: Sort all edges in G = (V,E) into a nondecresing sequence $|e_1| \leq |e_2| \leq \ldots \leq |e_m|$. Let $G(e_i)$ denote the subgraph obtained from G by deleting all edges longer than e_i.

Step2: $i \leftarrow 1$

Step3: If there exists a Hamiltonian cycle in $G(e_i)$, then this cycle is the solution and stop.

Step4: $i \leftarrow i+1$. Go to Step 3.

An example for BTSP algorithm

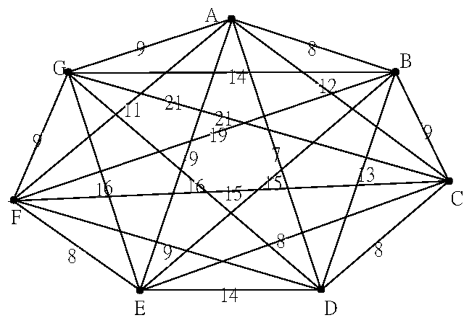


Fig. 9-9 A Complete Graph

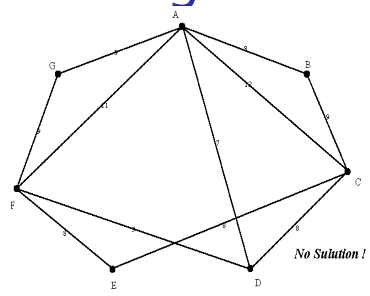


Fig. 9-10 G(AC) of the Graph in Fig. 9-9

- There is a Hamiltonian cycle, A-B-D-C-E-F-G-A, in G(BD).
- The optimal solution is 13.

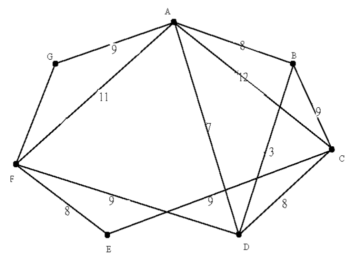


Fig. 9-11 G(BD) of the Graph in Fig. 9-9

Theorem for Hamiltonian cycles

- Def : The t-th power of $G=(V,E)$, denoted as $G^t=(V,E^t)$, is a graph that an edge $(u,v)\in E^t$ if there is a path from u to v with at most t edges in G.
- Theorem: If a graph G is bi-connected, then G^2 has a Hamiltonian cycle.

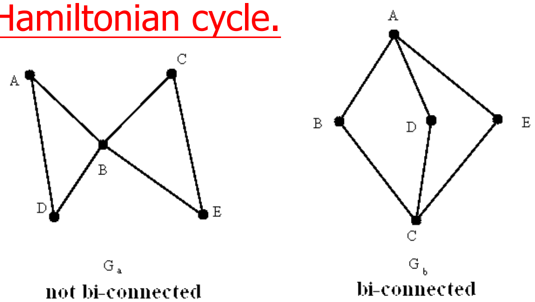


Fig. 9-12 Examples to Illustrate Bi-Connectedness

An example for the theorem

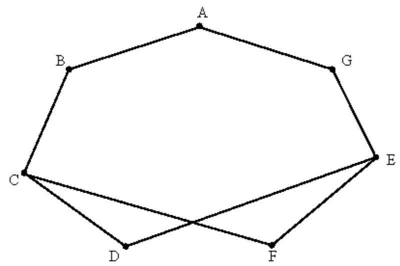
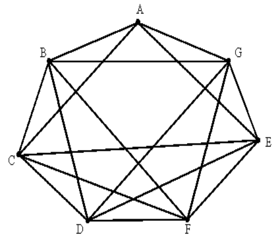


Fig. 9-13 A Bi-Connected Graph



G^2

A Hamiltonian cycle:

A-B-C-D-E-F-G-A

An approximation algorithm for BTSP

- Input: A complete graph $G=(V,E)$ where all edges satisfy triangular inequality.
- Output: A tour in G whose longest edges is not greater than twice of the value of an optimal solution to the special bottleneck traveling salesperson problem of G.

Step 1: Sort the edges into $|e_1|\le|e_2|\le\ldots\le|e_m|$.

Step 2: i := 1.

Step 3: If $G(e_i)$ is bi-connected, construct $G(e_i)^2$, find a Hamiltonian cycle in $G(e_i)^2$ and return this as the output.

Step 4: i := i + 1. Go to Step 3.

An example

Add some more edges. Then it becomes bi-connected.

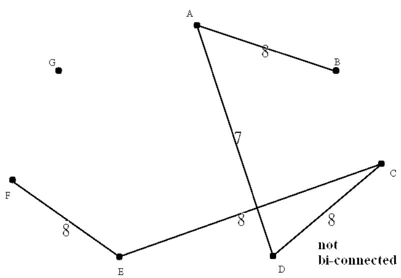


Fig. 9-15 G(FE) of the Graph in Fig. 9-9

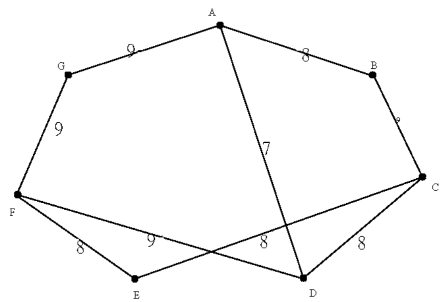


Fig. 9-16 G(FG) of the Graph in Fig. 9-9

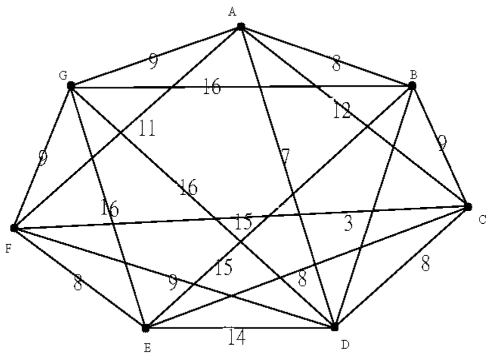


Fig. 9-17 $G(FG)^2$

- A Hamiltonian cycle: A-G-F-E-D-C-B-A.
- The longest edge: 16
- Time complexity: polynomial time

How good is the solution ?

- The approximate solution is bounded by two times an optimal solution.
- Reasoning:

  A Hamiltonian cycle is bi-connected.

  e_{op}: the longest edge of an optimal solution

  $G(e_i)$: the first bi-connected graph

  $|e_i| \le |e_{op}|$

  The length of the longest edge in $G(e_i)^2 \le 2|e_i|$

  (triangular inequality) $\le 2|e_{op}|$

NP-completeness

- Theorem: If there is a polynomial approximation algorithm which produces a bound less than two, then NP=P.

  (The Hamiltonian cycle decision problem reduces to this problem.)
- Proof:

  For an arbitrary graph G=(V,E), we expand G to a complete graph G_c:

  C_{ij} = 1 if $(i,j) \in E$

  C_{ij} = 2 if otherwise

  (The definition of C_{ij} satisfies the triangular inequality.)

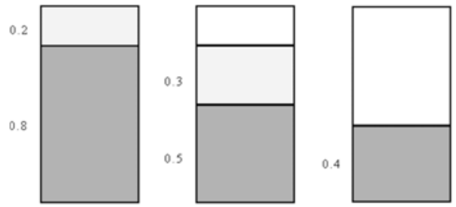
Let V^* denote the value of an optimal solution of the bottleneck TSP of G_c.

$V^* = 1 \Leftrightarrow$ G has a Hamiltonian cycle

Because there are only two kinds of edges, 1 and 2 in G_c, if we can produce an approximate solution whose value is less than $2V^*$, then we can also solve the Hamiltonian cycle decision problem.

The bin packing problem

- n items $a_1, a_2, \ldots, a_n$, $0 < a_i \le 1$, $1 \le i \le n$, to determine the minimum number of bins of unit capacity to accommodate all n items.
- E.g. n = 5, {0.8, 0.5, 0.2, 0.3, 0.4}



- The bin packing problem is NP-hard.

An approximation algorithm for the bin packing problem

- An approximation algorithm:

  (first-fit) place a_i into the lowest-indexed bin which can accommodate a_i.

- Theorem: The number of bins used in the first-fit algorithm is at most twice of the optimal solution.

Proof of the approximate rate

- Notations:
  - $S(a_i)$: the size of item a_i
  - OPT: # of bins used in an optimal solution
  - m: # of bins used in the first-fit algorithm
  - $C(B_i)$: the sum of the sizes of a_j's packed in bin B_i in the first-fit algorithm

- $\text{OPT} \ge \sum_{i=1}^{n} S(a_i)$

  $C(B_i) + C(B_{i+1}) > 1$

  $C(B_1)+C(B_2)+\ldots+C(B_m) > m/2$

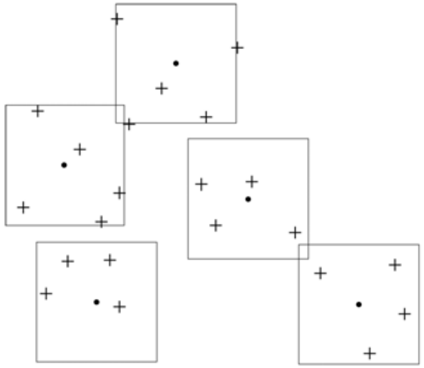
$\Rightarrow m < 2 \sum_{i=1}^{m} C(B_i) = 2 \sum_{i=1}^{n} S(a_i) \le 2\ \text{OPT}$

m < 2 OPT

The rectilinear m-center problem

- The sides of a rectilinear square are parallel or perpendicular to the x-axis of the Euclidean plane.
- The problem is to find m rectilinear squares covering all of the n given points such that the maximum side length of these squares is minimized.
- This problem is NP-complete.
- This problem for the solution with error ratio < 2 is also NP-complete.

(See the example on the next page.)



- Input: $P=\{P_1, P_2, \ldots, P_n\}$
- The size of an optimal solution must be equal to one of the $L_\infty(P_i,P_j)$'s, $1 \le i < j \le n$, where

$L_\infty((x_1,y_1),(x_2,y_2)) = \max\{|x_1-x_2|,|y_1-y_2|\}$.

An approximation algorithm

- Input: A set P of n points, number of centers: m
- Output: SQ[1], …, SQ[m]: A feasible solution of the rectilinear m-center problem with size less than or equal to twice of the size of an optimal solution.

Step 1: Compute rectilinear distances of all pairs of two points and sort them together with 0 into an ascending sequence D[0]=0, D[1], …, D[n(n-1)/2].

Step 2: LEFT := 1, RIGHT := n(n-1)/2 //* Binary search

Step 3: i := $\lceil$(LEFT + RIGHT)/2$\rceil$.

Step 4: If Test(m, P, D[i]) is not "failure" then
  RIGHT := i-1
  else LEFT := i+1

Step 5: If RIGHT = LEFT then
  return Test(m, P, D[RIGHT])
  else go to Step 3.

Algorithm Test(m, P, r)

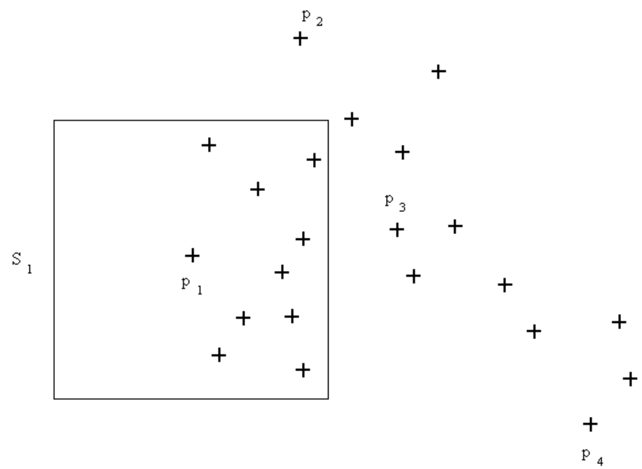
- Input: point set: P, number of centers: m, size: r.
- Output: "failure", or SQ[1], …, SQ[m] m squares of size 2r covering P.

Step 1: PS := P

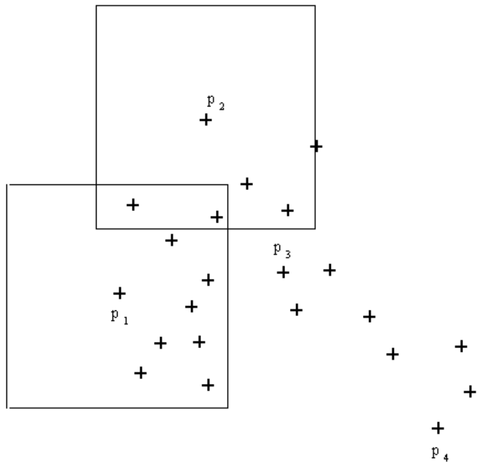
Step 2: For i := 1 to m do
  If PS ≠ ∅ then
    p := the point is PS with the smallest x-value
    SQ[i] := the square of size 2r with center at p
    PS := PS -{points covered by SQ[i]}
  else SQ[i] := SQ[i-1].

Step 3: If PS = ∅ then return SQ[1], …, SQ[m]
  else return "failure". (See the example on the next page.)

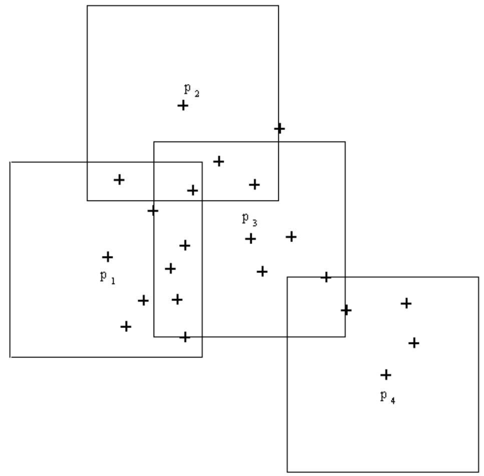
An example for the algorithm



The first application of the relaxed test subroutine.



The second application of the test subroutine.



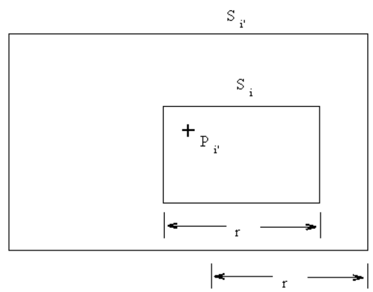
A feasible solution of the rectilinear 5-center problem.

Time complexity

- Time complexity: $O(n^2 \log n)$
  - Step 1: O(n)
  - Step 2: O(1)
  - Step 3 ~ Step 5:
    $O(\log n) * O(mn) = O(n^2 \log n)$

How good is the solution ?

- The approximation algorithm is of error ratio 2.
- Reasoning: If r is feasible, then Test(m, P, r) returns a feasible solution of size 2r.



The explanation of $S_i \subset S_i'$

Chapter 10

Amortized Analysis

An example– push and pop

- A sequence of operations: OP_1, OP_2, … OP_m
  OP_i: several pops (from the stack) and
  one push (into the stack)
  t_i : time spent by OP_i
  the average time per operation:

$$t_{ave} = \frac{1}{m}\sum_{i=1}^{m} t_i$$

- Example: a sequence of push and pop
  p: pop , u: push

i	1	2	3	4	5	6	7	8
OP_i	1u	1u	2p 1u	1u	1u	1u	2p 1u	1p 1u
t_i	1	1	3	1	1	1	3	2

$$\begin{aligned} t_{ave} &= (1+1+3+1+1+1+3+2)/8 \\ &= 13/8 \\ &= 1.625 \end{aligned}$$

- Another example: a sequence of push and pop
  p: pop , u: push

i	1	2	3	4	5	6	7	8
OP_i	1u	1p 1u	1u	1u	1u	1u	5p 1u	1u
t_i	1	2	1	1	1	1	6	1

$$\begin{aligned} t_{ave} &= (1+2+1+1+1+1+6+1)/8 \\ &= 14/8 \\ &= 1.75 \end{aligned}$$

Amortized time and potential function

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

a_i : amortized time of OP_i

Φ_i : potential function of the stack after OP_i

$\Phi_i - \Phi_{i-1}$: change of the potential

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \sum_{i=1}^{m}(\Phi_i - \Phi_{i-1})$$

$$= \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0$$

If $\Phi_m - \Phi_0 \geq 0$, then $\sum_{i=1}^{m} a_i$ represents an upper bound of $\sum_{i=1}^{m} t_i$

Amortized analysis of the push-and-pop sequence

- Φ_i : # of elements in the stack
  We have $\Phi_m - \Phi_0 \geq 0$
- Suppose that before we execute Op_i, there are k elements in the stack and Op_i consists of n pops and 1 push.

$$\Phi_{i-1} = k$$

$$t_i = n + 1$$

$$\text{Then, } \Phi_i = k - n + 1$$

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$= (n+1) + (k\text{-}n+1)\text{-}k$$

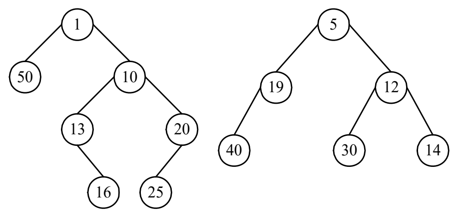
$$= 2$$

- We have $(\sum_{i=1}^{m} a_i)/m = 2$
  Then, $t_{ave} \leq 2$.
- By observation, at most m pops and m pushes are executed in m operations. Thus,

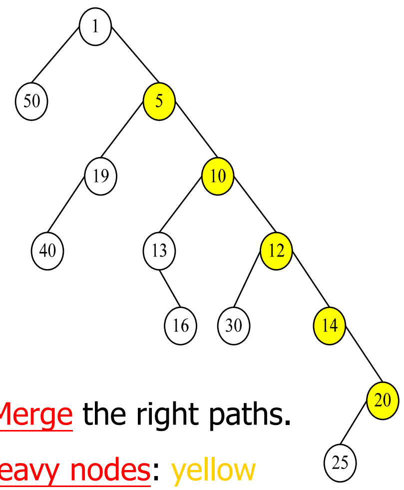
$$t_{ave} \leq 2.$$

Skew heaps

- meld: merge + swapping



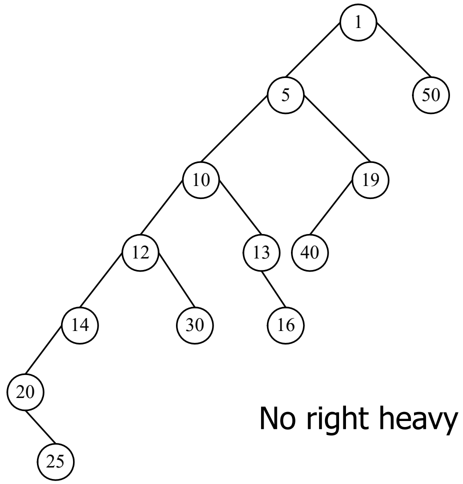
Two skew heaps



Step 1: Merge the right paths.

5 right heavy nodes: yellow

Step 2: Swap the children along the right path.



No right heavy node

Amortized analysis of skew heaps

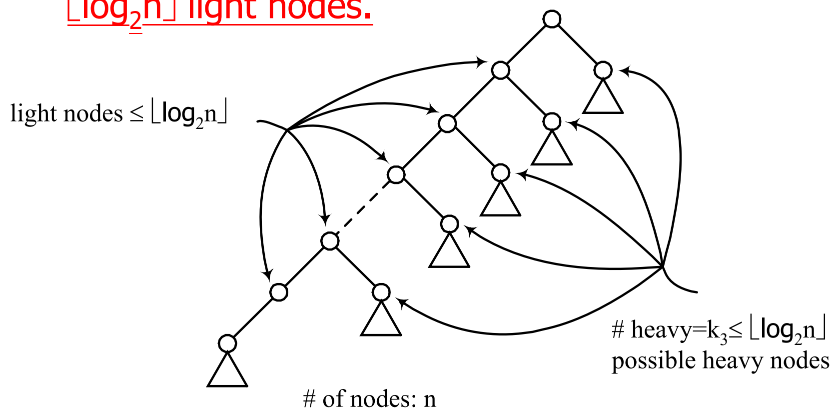
- meld: merge + swapping
- operations on a skew heap:
  - find-min(h): find the min of a skew heap h.
  - insert(x, h): insert x into a skew heap h.
  - delete-min(h): delete the min from a skew heap h.
  - meld(h_1, h_2): meld two skew heaps h_1 and h_2.

  The first three operations can be implemented by melding.

Potential function of skew heaps

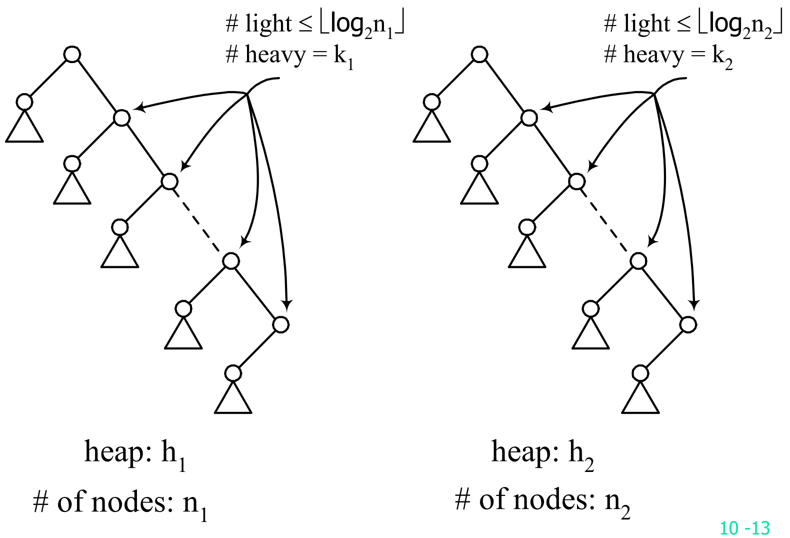
- wt(x): # of descendants of node x, including x.
- heavy node x: wt(x) > wt(p(x))/2, where p(x) is the parent node of x.
- light node : not a heavy node
- potential function Φ_i: # of right heavy nodes of the skew heap.

- Any path in an n-node tree contains at most $\lfloor \log_2 n \rfloor$ light nodes.



- The number of right heavy nodes attached to the left path is at most $\lfloor \log_2 n \rfloor$.

Amortized time



$a_i = t_i + \Phi_i - \Phi_{i-1}$

t_i : time spent by OP_i

$t_i \le 2+\lfloor \log_2 n_1 \rfloor + k_1 + \lfloor \log_2 n_2 \rfloor + k_2$

("2" counts the roots of h_1 and h_2)

$\le 2+2\lfloor \log_2 n \rfloor + k_1 + k_2$

where $n=n_1+n_2$

$\Phi_i - \Phi_{i-1} = k_3-(k_1+k_2) \le \lfloor \log_2 n \rfloor - k_1 - k_2$

$a_i = t_i + \Phi_i - \Phi_{i-1}$

$\le 2+2\lfloor \log_2 n \rfloor + k_1 + k_2 + \lfloor \log_2 n \rfloor - k_1 - k_2$

$= 2+3\lfloor \log_2 n \rfloor$

$\Rightarrow a_i = O(\log_2 n)$

AVL-trees

height balance of node v:

hb(v)= (height of right subtree) – (height of left subtree)

- The hb(v) of every node never differ by more than 1.

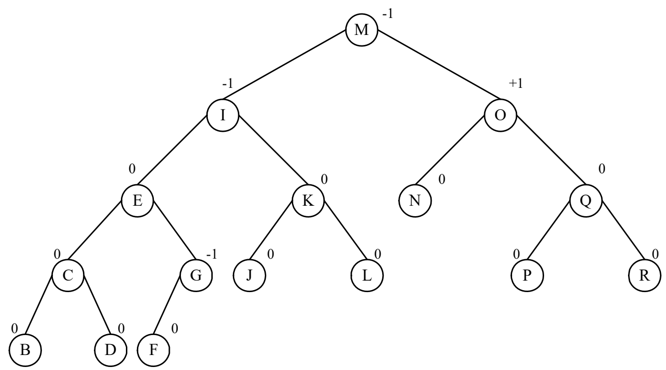
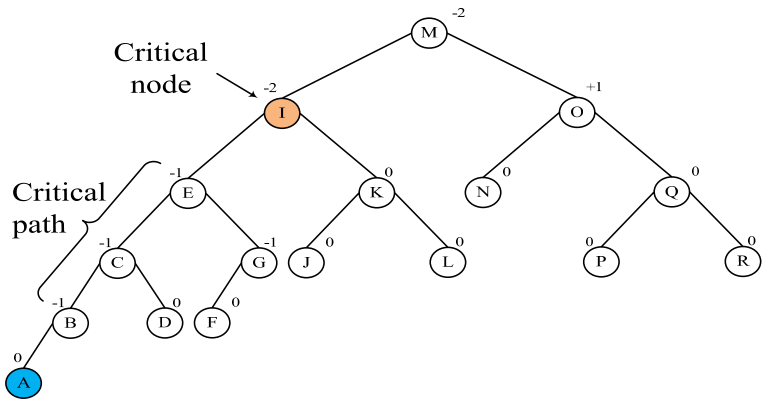


Fig. An AVL-Tree with Height Balance Labeled

- Add a new node A.



Before insertion, hb(B)=hb(C)=hb(E)=0

hb(I)≠0 the first nonzero from leaves.

Amortized analysis of AVL-trees

- Consider a sequence of m insertions on an empty AVL-tree.
  - T_0: an empty AVL-tree.
  - T_i: the tree after the ith insertion.
  - L_i: the length of the critical path involved in the ith insertion.
  - X_1: total # of balance factor changing from 0 to +1 or -1 during these m insertions (the total cost for rebalancing)

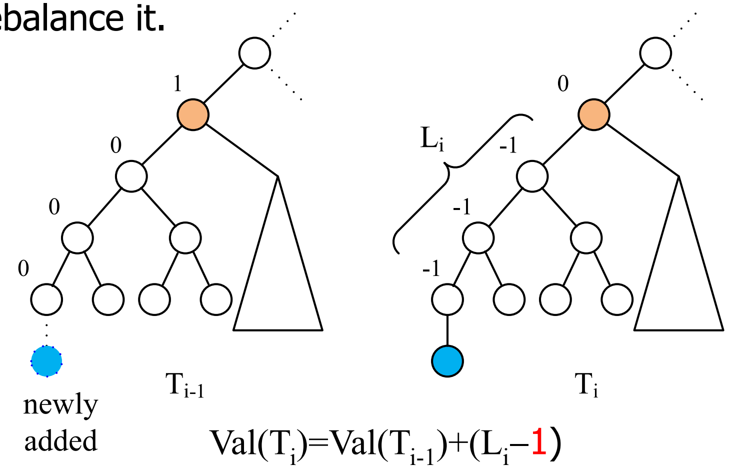
$X_1 = \sum_{i=1}^{m} L_i$, we want to find X_1.

Val(T): # of unbalanced node in T

(height balance $\neq 0$)

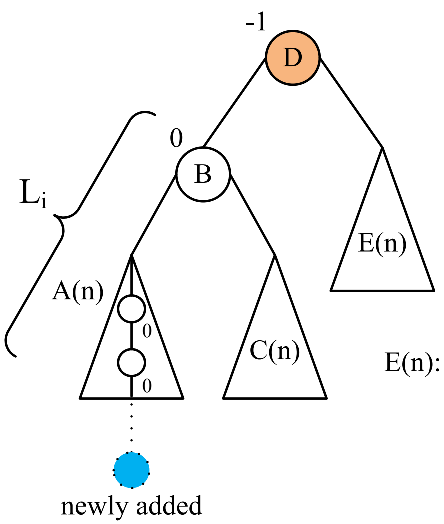
Case 1 : Absorption

- The tree height is not increased, we need not rebalance it.



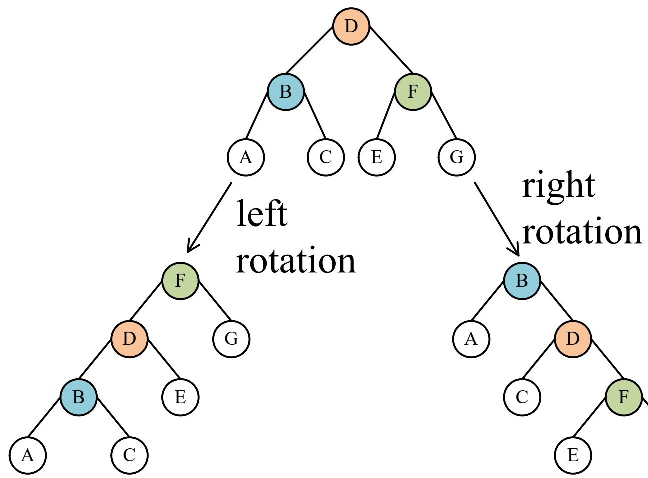
$\mathrm{Val}(T_i)=\mathrm{Val}(T_{i-1})+(L_i-1)$

Case 2.1 Single rotation



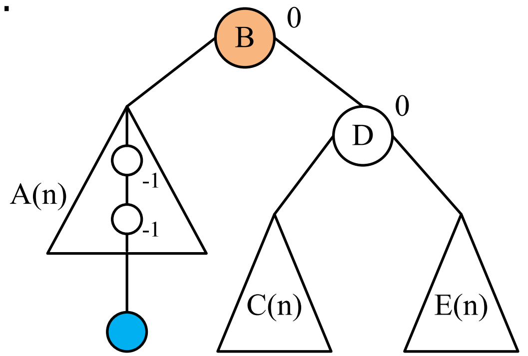
E(n): height of tree E is n

Case 2 : Rebalancing the tree



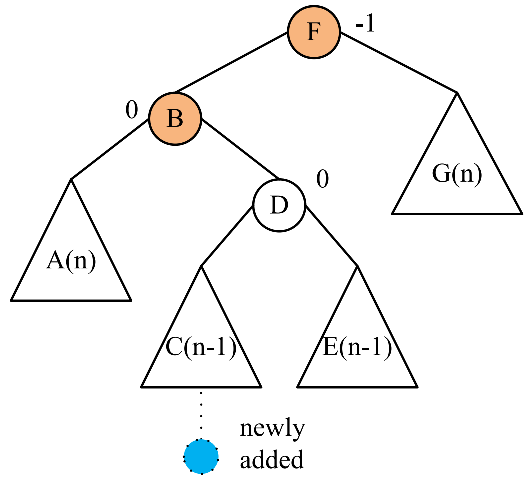
Case 2.1 Single rotation

- After a right rotation on the subtree rooted at D:



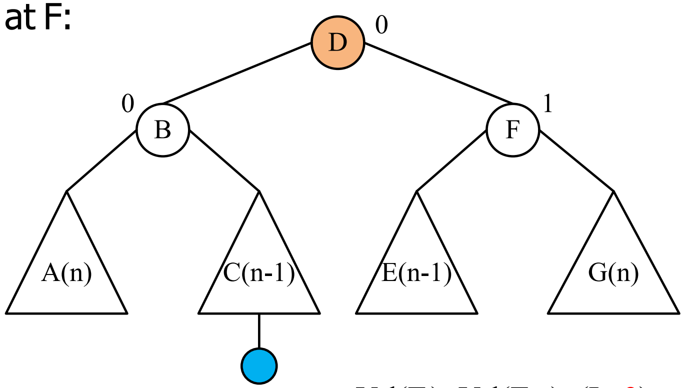
$Val(T_i)=Val(T_{i-1})+(L_i-2)$

Case 2.2 Double rotation



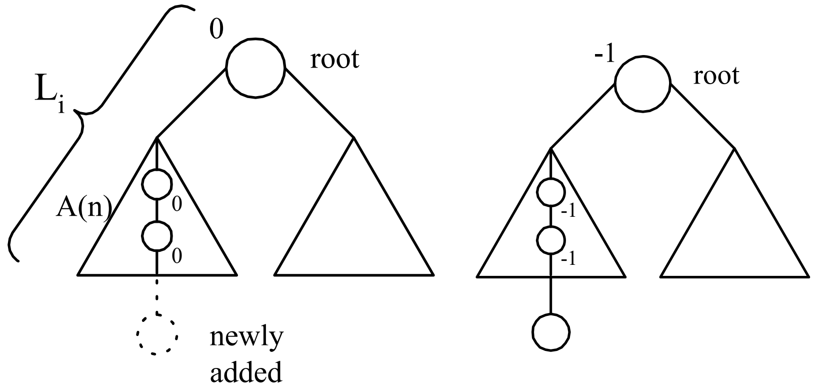
Case 2.2 Double rotation

- After a left rotation on the subtree rooted at B and a right rotation on the subtree rooted at F:



$Val(T_i)=Val(T_{i-1})+(L_i-2)$

Case 3 : Height increase



- L_i is the height of the root.

$Val(T_i)=Val(T_{i-1})+L_i$

Amortized analysis of X_1

X_2: # of absorptions in case 1
X_3: # of single rotations in case 2
X_4: # of double rotations in case 2
X_5: # of height increases in case 3

$$\mathrm{Val}(T_m) = \mathrm{Val}(T_0)+\sum_{i=1}^{m} L_i - X_2 - 2(X_3+X_4)$$
$$= 0+X_1-X_2-2(X_3+X_4)$$

$\mathrm{Val}(T_m) \le 0.618m$ (proved by Knuth)

$$\Rightarrow X_1 = \mathrm{Val}(T_m)+2(X_2+X_3+X_4)-X_2$$
$$\le 0.618m+2m$$
$$= 2.618m$$

A self-organizing sequential search heuristics

- 3 methods for enhancing the performance of sequential search

(1) Transpose Heuristics:

Query	Sequence
B	B
D	D B
A	D A B
D	D A B
D	D A B
C	D A C B
A	A D C B

(2) Move-to-the-Front Heuristics:

Query	Sequence
B	B
D	D B
A	A D B
D	D A B
D	D A B
C	C D A B
A	A C D B

(3) Count Heuristics: (decreasing order by the count)

Query	Sequence
B	B
D	B D
A	B D A
D	D B A
D	D B A
A	D A B
C	D A B C
A	D A B C

Analysis of the move-to-the-front heuristics

- interword comparison: unsuccessful comparison
- intraword comparison: successful comparison
- pairwise independent property:
  - For any sequence S and all pairs P and Q, # of interword comparisons of P and Q is exactly # of interword comparisons made for the subsequence of S consisting of only P's and Q's.

(See the example on the next page.)

Pairwise independent property in move-to-the-front

e.g.

Query	Sequence	(A, B) comparison
C	C	
A	A C	
C	C A	
B	B C A	√
C	C B A	
A	A C B	√

# of comparisons made between A and B: 2

Consider the subsequence consisting of A and B:

Query	Sequence	(A, B) comparison
A	A	
B	B A	√
A	A B	√

# of comparisons made between A and B: 2

Query	Sequence	C	A	C	B	C	A
	(A, B)		0		1		1
	(A, C)	0	1	1		0	1
	(B, C)	0		0	1	1	
		0	1	1	2	1	2

There are 3 distinct interword comparisons:
(A, B), (A, C) and (B, C)

- We can consider them separately and then add them up.
  the total number of interword comparisons:
  0+1+1+2+1+2 = 7

Theorem for the move-to-the-front heuristics

$C_M(S)$: # of comparisons of the move-to-the-front heuristics

$C_O(S)$: # of comparisons of the optimal static ordering

$$\boxed{C_M(S) \leq 2C_O(S)}$$

Proof

Proof:

- $Inter_M(S)$: # of interword comparisons of the move to the front heuristics
- $Inter_O(S)$: # of interword comparisons of the optimal static ordering

Let S consist of a A's and b B's, a < b.

The optimal static ordering: BA

$$\left.\begin{array}{l} Inter_O(S) = a \\ Inter_M(S) \leq 2a \end{array}\right\} \Rightarrow Inter_M(S) \leq 2Inter_O(S)$$

Proof (cont.)

- Consider any sequence consisting of more than two items. Because of the pairwise independent property, we have $Inter_M(S) \leq 2Inter_O(S)$
- $Intra_M(S)$: # of intraword comparisons of the move-to-the-front heuristics
- $Intra_O(S)$: # of intraword comparisons of the optimal static ordering
- $Intra_M(S) = Intra_O(S)$
- $Inter_M(S) + Intra_M(S) \leq 2Inter_O(S) + Intra_O(S)$
  $\Rightarrow C_M(S) \leq 2C_O(S)$

The count heuristics

- The count heuristics has a similar result: $C_C(S) \leq 2C_O(S)$, where $C_C(S)$ is the cost of the count heuristics

The transposition heuristics

- The transposition heuristics does not possess the pairwise independent property.
- We can not have a similar upper bound for the cost of the transposition heuristics.

e.g.

Consider pairs of distinct items independently.

Query	Sequence	C	A	C	B	C	A
	(A, B)		0		1		1
	(A, C)	0	1	1		0	1
	(B, C)	0		0	1	1	
		0	1	1	2	1	2

\# of interword comparisons: 7 (not correct)

the correct interword comparisons:

Query Sequence	C	A	C	B	C	A
Data Ordering	C	AC	CA	CBA	CBA	CAB
Number of Interword Comparisons	0	1	1	2	0	2

$\underbrace{0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2}_{6}$

Chapter 11

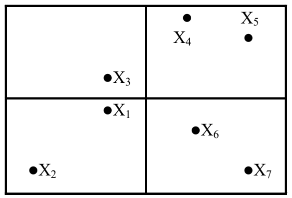
Randomized Algorithms

Randomized algorithms

- In a randomized algorithm (probabilistic algorithm), we make some random choices.
- 2 types of randomized algorithms:
  - For an optimization problem, a randomized algorithm gives an optimal solution. The average case time-complexity is more important than the worst case time-complexity.
  - For a decision problem, a randomized algorithm may make mistakes. The probability of producing wrong solutions is very small.

The closest pair problem

- This problem can be solved by the divide-and-conquer approach in O(nlogn) time.
- The randomized algorithm:
  - Partition the points into several clusters:



  - We only calculate distances among points within the same cluster.
  - Similar to the divide-and-conquer strategy. There is a dividing process, but no merging process.

A randomized algorithm for closest pair finding

- Input: A set S consisting of n elements $x_1, x_2, \ldots, x_n$, where $S \subseteq R^2$.
- Output: The closest pair in S.

Step 1: Randomly choose a set $S_1=\{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\}$ where $m=n^{2/3}$. Find the closest pair of S_1 and let the distance between this pair of points be denoted as δ .

Step 2: Construct a set of squares T with mesh-size δ.

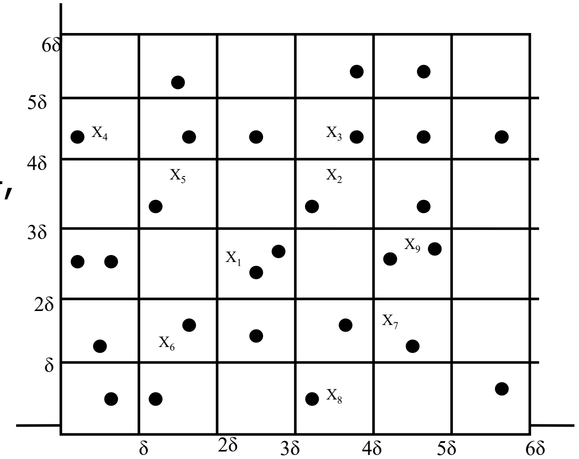
Step 3: Construct four sets of squares T_1, T_2, T_3 and T_4 derived from T by doubling the mesh-size to 2δ .

Step 4: For each T_i, find the induced decomposition $S=S_1^{(i)} \cup S_2^{(i)} \cup \cdots \cup S_j^{(i)}$, $1 \le i \le 4$, where $S_j^{(i)}$ is a non-empty intersection of S with a square of T_i.

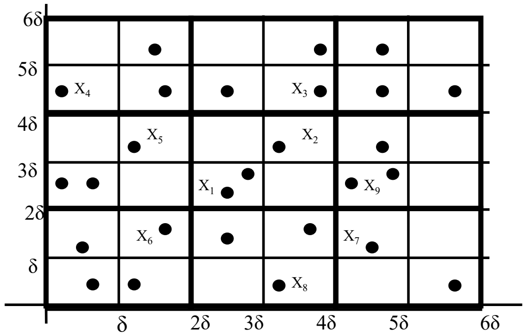
Step 5: For each $x_p, x_q \in S_j^{(i)}$, compute $d(x_p, x_q)$. Let x_a and x_b be the pair of points with the shortest distance among these pairs. Return x_a and x_b as the closest pair.

An example

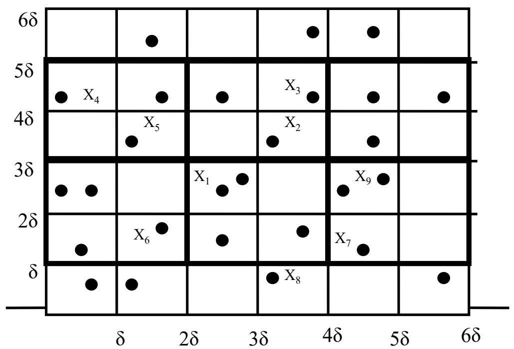
- n=27 points.
  $m=n^{2/3}$
  $S_1 = \{x_1, x_2, \ldots, x_9\}$,
  $\delta = d(x_1, x_2)$



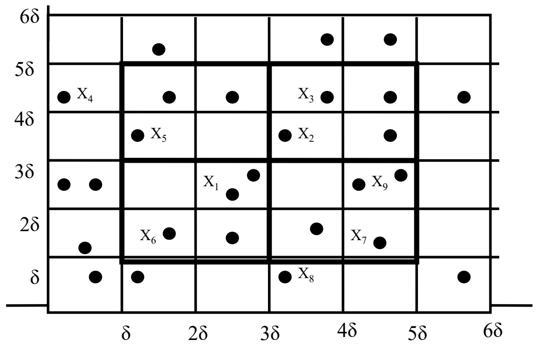
T_1



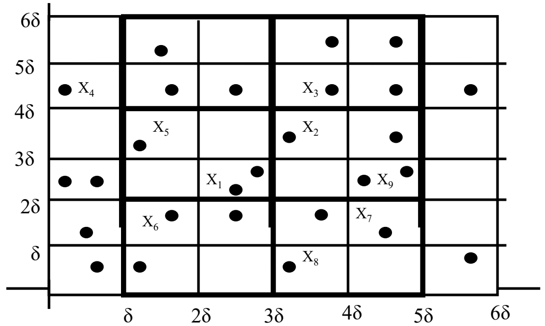
T_2



T_3



T_4



Time complexity

- Time complexity: O(n) in average
- step 1: O(n)

  method: Recursively apply the algorithm once,

  i.e. randomly choose $(n^{2/3})^{2/3} = n^{4/9}$

  points from the $n^{2/3}$ points, then solve

  it with a straightforward method for the

  $n^{4/9}$ points: $O(n^{8/9})$
- step 2 ~ Step 4: O(n)
- step 5: O(n)with probability $1\text{-}2e^{-cn^{\frac{1}{6}}}$

Analysis of Step 5

- How many distance computations in step 5?

  δ : mesh-size in step 1

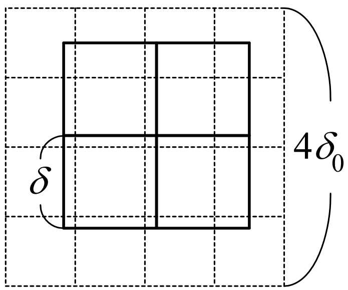
  T_i: partition in step 5

  $N(T_i)$: # of distance computations in partitionT_i

  Fact: There exists a particular partition R_0, whose mesh-size is δ_0 such that

  (1) $N(R_0) \le c_0n$.

  (2) the probability that $\delta \le \sqrt{2}\delta_0$ is $1\text{-}2e^{-cn^{\frac{1}{6}}}$.



- Construct R_1, R_2, …, R_{16} mesh-size: $4\delta_0$
- The probability that each square in T_i falls into at least one square of R_i , $1 \le i \le 16$ is $1\text{-}2e^{-cn^{\frac{1}{6}}}$.
- The probability that

  $N(T_i) \le \sum_{i=1}^{16} N(R_i)$ is $1\text{-}2e^{-cn^{\frac{1}{6}}}$.

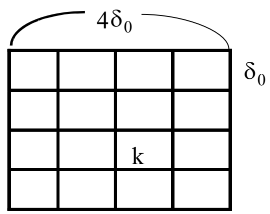
- Let the square in R_0 with the largest number of elements among the 16 squares have k elements.

  $$\frac{k(k-1)}{2} = O(k^2), \frac{16k(16k-1)}{2} = O(k^2)$$

- $N(R_0) \le c_0n$ => $N(R_i) \le c_in$

$$\boxed{N(T_i) \le \sum_{i=1}^{16} N(R_i) = O(n) \text{ with probability } 1\text{-}2e^{-cn^{\frac{1}{6}}}.}$$

R_i:



A randomized algorithm to test whether a number is prime.

- This problem is very difficult and no polynomial algorithm has been found to solve this problem
- Traditional method:

  use 2,3,… $\sqrt{N}$ to test whether N is prime.

  input size of N : B=$\log_2$N (binary representation)

  $\sqrt{N}$ =$2^{B/2}$, exponential function of B

Thus $\sqrt{N}$ can not be viewed as a polynomial function of the input size.

Randomized prime number testing algorithm

- Input: A positive number N, and a parameter m.
- Output: Whether N is a prime or not, with probability of being correct at least $1-\varepsilon = 1-2^{-m}$.

Step 1: Randomly choose m numbers $b_1, b_2, \ldots, b_m$, $1 \le b_1, b_2, \ldots, b_m < N$, where $m \ge \log_2(1/\varepsilon)$.

Step 2: For each b_i, test whether $W(b_i)$ holds where $W(b_i)$ is defined as follows:

(1) $b_i^{N-1} \neq 1 \bmod N$ or

(2) $\exists$ j such that $\frac{N-1}{2^j}$ = k is an integer and the greatest common divisor of $(b_i)^k-1$ and N is not 1 or N.

If any $W(b_i)$ holds, then return N as a composite number, otherwise, return N as a prime.

Examples for randomized prime number testing

- Example 1: N = 12

  Randomly choose 2, 3, 7

  $2^{12-1} = 2048 \neq 1 \bmod 12$

  $\Rightarrow$ 12 is a composite number.

- Example 2: N = 11

  Randomly choose 2, 5, 7

  (1) $2^{11-1}=1024 \equiv 1 \bmod 11$

  j=1, $(N-1)/2^j=5$

  GCD(2^5-1, 11) = GCD(31,11) = 1

  W(2) does not hold .

  (2) $5^{11-1}=9765625 \equiv 1 \bmod 11$

  GCD(5^5-1, 11) = GCD(3124,11) = 11

  W(5) does not hold .

  (3) $7^{11-1}=282475249 \equiv 1 \bmod 11$

  GCD(7^5-1, 11) = GCD(16806,11) = 1

  W(7) does not hold .

- Thus, 11 is a prime number with the probability of correctness being at least $1-2^{-3}= 7/8$.

Theorem for number theory

- Theorem:
  - If W(b) holds for any $1 \leq b<N$, then N is a composite number .
  - If N is composite, then

  $(N-1)/2 \leq |\{ b \mid 1 \leq b<N, W(b) \text{ holds} \}|$.

Pattern matching

- Pattern string : X length : n
  Text string : Y length : m, $m \geq n$
  To find the first occurrence of X as a consecutive substring of Y .
  Assume that X and Y are binary strings.
- e.g. X = 01001 , Y = $1010\underbrace{01001}_{X}11$
- Straightforward method : O(mn)
- Knuth-Morris-Pratt's (KMP) algorithm : O(m)
- The randomized algorithm : O(mk) with a mistake of small probability. (k:# of testings)

Binary representation

- $X = x_1 x_2 \ldots x_n \in \{0,1\}$
  $Y = y_1 y_2 \ldots y_m \in \{0,1\}$
  Let $Y(i)=y_i y_{i+1} \ldots y_{i+n-1}$
  A match occurs if X=Y(i) for some i .
- Binary values of X and Y(i):
  $B(X) = x_1 \cdot 2^{n-1} + x_2 \cdot 2^{n-2} + \ldots + x_n$
  $B(Y(i)) = y_i \cdot 2^{n-1} + y_{i+1} \cdot 2^{n-2} + \ldots + y_{i+n-1}$,
  $1 \leq i \leq m-n+1$

Fingerprints of binary strings

- Let p be a randomly chosen prime number in $\{1,2,\ldots,nt^2\}$, where t = m - n + 1.
- Notation: $(x_i)_p = x_i \bmod p$
- Fingerprints of X and Y(i):
  $B_p(x) = ((((x_1 \cdot 2)_p + x_2)_p \cdot 2)_p + x_3)_p \cdot 2 \ldots$
  $B_p(Y(i)) = ((((y_i \cdot 2)_p + y_{i+1})_p \cdot 2 + y_{i+2})_p \cdot 2 \ldots$
  $\Rightarrow B_p(Y(i+1)) = ((B_p(Y_i) - 2^{n-1} \cdot y_i) \cdot 2 + Y_{i+n})_p$
  $= (((B_p(Y_i) - ((2^{n-1})_p \cdot y_i)_p)_p \cdot 2)_p + y_{i+n})_p$
- If X=Y(i), then $B_p(X) = B_p(Y(i))$, but not vice versa.

Examples for using fingerprints

- Example: X = 10110 , Y = 110110
  $n = 5$, $m = 6$, $t = m - n + 1 = 2$
  suppose P=3.
  $B_p(X) = (22)_3 = 1$
  $B_p(Y(1)) = (27)_3 = 0$
  $\Rightarrow X \neq Y(1)$
  $B_p(Y(2)) = ((0-2^4)_3\ 2+0)_3 = 1$
  $\Rightarrow X = Y(2)$

- e.g. X = 10110 , Y = 10011 , P = 3
  $B_p(X) = (22)_3 = 1$
  $B_p(Y(1)) = (19)_3 = 1$
  $\Rightarrow$ X= Y(1) WRONG!

- If $B_p(X) \neq B_p(Y(i))$, then $X \neq Y(i)$.
- If $B_p(X) = B_p(Y(i))$, we may do a bit by bit checking or compute k different fingerprints by using k different prime numbers in $\{1,2,\ldots nt^2\}$.

A randomized algorithm for pattern matching

- Input: A pattern $X = x_1\ x_2 \ldots x_n$, a text $Y = y_1\ y_2 \ldots y_m$ and a parameter k.
- Output:
  (1) No, there is no consecutive substring in Y which matches with X.
  (2) Yes, $Y(i) = y_i\ y_{i+1} \ldots y_{i+n-1}$ matches with X which is the first occurrence.

If the answer is "No" , there is no mistake.

If the answer is "Yes" , there is some probability that a mistake is made.

Step 1: Randomly choose k prime numbers $p_1, p_2, \ldots, p_k$ from $\{1,2,\ldots,nt^2\}$, where $t = m - n + 1$.

Step 2: i = 1.

Step 3: j = 1.

Step 4: If $B(X)_{Pj} \neq (B(Y_i))_{pj}$, then go to step 5.
If j = k, return Y(i) as the answer.
j = j + 1.
Go to step 4.

Step5: If i = t, return "No, there is no consecutive substring in Y which matches with X."
i = i + 1.
Go to Step 3.

An example for the algorithm

- $X = 10110$, $Y = 100111$, $P_1 = 3$, $P_2 = 5$
  $B_3(X) = (22)_3 = 1$
  $B_5(X) = (22)_5 = 2$
  $B_3(Y(2)) = (7)_3 = 1$
  $B_5(y(2)) = (7)_5 = 2$
  Choose one more prime number, $P_3 = 7$
  $B_7(x) = (22)_7 = 1$
  $B_7(Y(2)) = (7)_7 = 0$
  $\Rightarrow X \neq Y(2)$

How often does a mistake occur?

- If a mistake occurs in X and Y(i), then
  $B(X) - B(Y(i)) \neq 0$, and
  p_j divides $| B(X) - B(Y(i)) |$ for all p_j's.
- Let Q = $\prod_{i \text{ where } p_j \text{ divides} |B(X)-B(Y(i))|} |B(X)-B(Y(i))|$
- $Q<2^{n(m-n+1)}$
  reason: $B(x)<2^n$, and at most $(m-n+1)$ $B(Y(i))$'s

$$\underbrace{2^n 2^n \ldots 2^n}_{m-n+1}$$

Theorem for number theory

- Theorem: If $u \geq 29$ and $q<2^u$, then q has fewer than $\pi(u)$ diffferent prime number divisors where $\pi(u)$ is the number of prime numbers smaller than u.
- Assume $nt \geq 29$.
  $Q < 2^{n(m-n+1)} = 2^{nt}$
  $\Rightarrow$ Q has fewer than $\pi(nt)$ different prime number divisors.
- If p_j is a prime number selected from $\{1, 2, \ldots, M\}$, the probability that p_j divides Q is less than $\frac{\pi(nt)}{\pi(M)}$.
- If k different prime numbers are selected from $\{1, 2, \ldots nt^2\}$, the probability that a mistake occurs is less than $\left(\frac{\pi(nt)}{\pi(nt^2)}\right)^k$ provided $nt \geq 29$.

An example for mistake probability

- How do we estimate $\left(\frac{\pi(nt)}{\pi(nt^2)}\right)^k$
- Theorem: For all $u \geq 17$, $\frac{u}{\ln u} \leq \pi(u) \leq 1.25506 \frac{u}{\ln u}$
- $\frac{\pi(nt)}{\pi(nt^2)} \leq 1.25506 \cdot \frac{nt}{\ln nt} \cdot \frac{\ln(nt^2)}{nt^2}$

$$= \frac{1.25506}{t}(1 + \frac{\ln(t)}{\ln(nt)})$$

- Example: $n = 10$, $m = 100$, $t = m - n + 1 = 91$

$$\frac{\pi(nt)}{\pi(nt^2)} \leq 0.0229$$

Let k=4 $(0.0229)^4 \approx 2.75 \times 10^{-7}$ // very small

Interactive proofs: method I

- Two persons: A : a spy
  B : the boss of A
  When A wants to talk to B , how does B know that A is the real A, not an enemy imitating A ?
- Method I : a trivial method
  B may ask the name of A's mother (a private secret)
- Disadvantage:
  The enemy can collect the information, and imitate A the next time.

Interactive proofs: method II

- Method II:
  B may send a Boolean formula to A and ask A to determine its satisfiability (an NP-complete problem).
  It is assumed that A is a smart person and knows how to solve this NP-complete problem.
  B can check the answer and know whether A is the real A or not.
- Disadvantage:
  The enemy can study methods of mechanical theorem proving and sooner or later he can imitate A.
- In Methods I and II, A and B have revealed too much.

A randomized algorithm for interactive proofs

- Method III:
  B can ask A to solve a quadratic nonresidue problem in which the data can be sent back and forth without revealing much information.
- Definition:
  GCD(x, y) = 1, y is a quadratic residue mod x if $z^2 \equiv y$ mod x for some z, 0 < z < x, GCD(x, z) = 1, and y is a quadratic nonresidue mod x if otherwise.

(See the example on the next page.)

An example for quadratic residue/nonresidue

- Let
  QR = {(x, y) | y is a quadratic residue mod x}
  QNR = {(x, y) | y is a quadratic nonresidue mod x}
- Try to test x = 9, y = 7:
  $1^2 \equiv 1$ mod 9 $2^2 \equiv 4$ mod 9
  $3^2 \equiv 0$ mod 9 $4^2 \equiv 7$ mod 9
  $5^2 \equiv 7$ mod 9 $6^2 \equiv 0$ mod 9
  $7^2 \equiv 4$ mod 9 $8^2 \equiv 1$ mod 9
- We have (9,1), (9,4), (9,7) $\in$ QR
  but (9,5), (9,8) $\in$ QNR

Detailed method for interactive proofs

1) A and B know x and keep x confidential .
   B knows y.
2) Action of B:
   Step 1: Randomly choose m bits: $b_1, b_2, \ldots, b_m$,
   where m is the length of the binary representation of x.
   Step 2: Find $z_1, z_2, \ldots, z_m$ s.t. $GCD(z_i, x)=1$ for all i .
   Step 3:Compute $w_1, w_2, \ldots, w_m$:
   $w_i \leftarrow z_i^2 \bmod x$ if $b_i=0$ $\quad$ //$(x, w_i) \in QR$
   $w_i \leftarrow (z_i^2 y) \bmod x$ if $b_i=1$ $\quad$ //$(x, w_i) \in QNR$
   Step 4: Send $w_1, w_2, \ldots, w_m$ to A.

3) Action of A:
   Step 1: Receive $w_1, w_2, \ldots, w_m$ from B.
   Step 2: Compute $c_1, c_2, \ldots, c_m$:
   $c_i \leftarrow 0$ if $(x, w_i) \in QR$
   $c_i \leftarrow 1$ if $(x, w_i) \in QNR$
   Send $c_1, c_2, \ldots, c_m$ to B.
4) Action of B:
   Step 1: Receive $c_1, c_2, \ldots, c_m$ from A.
   Step 2: If $(x, y) \in QNR$ and $b_i = c_i$ for all i, then A is the real A (with probability $1-2^{-m}$).