Chapter 1

Introduction

大學程式設計競賽

- 臺灣與世界競賽時程
  - 全國大專軟體設計競賽：每年10月
  - 私立大學程式設計競賽：2011年起，每年6、7月
  - 科技大學程式設計競賽：2016年起，每年6、7月
  - ICPC (International Collegiate Programming Contest) 亞洲台灣區域賽：每年10、11月
  - ICPC亞洲其他區域賽：每年10~12月
  - ICPC世界總決賽：每年2~6月
- 2024-2025年，全球共有39個區域賽(region)，其中亞洲共有16個區域域賽(台灣為其中之一)。
- 2024-2025參賽統計：超過100國， 超過2500大學， 超過10000隊伍。

ICPC Regional Contests (2024)

- ICPC: International Collegiate Programming Contest

Region	# of contests
Africa and the Middle East	2
Asia	16
Europe	5
Latin America	4
North America	11
South Pacific	1

ICPC

- 緣起：1970年美國Texas A&M University大學程式設計比賽
- 1977年：第一次總決賽
- 1977~1989：參與比賽的大學主要為美國與加拿大。
- 1989年：開始建立區域賽(regional)的制度
- 1991年：亞洲首支隊伍參加世界總決賽--國立交通大學。
- 1995年。台灣首度舉辦亞洲區域賽
- 1996年以前，歷年的贊助廠商依先後順序分別為Apple、AT&T 和Microsoft 。
- 1997年~：IBM公司為此競賽主要贊助商。
- 1997年，參賽隊伍1100隊，來自560個大學
- 2002年，上海交大首度獲得總決賽冠軍
- 2010年，參賽隊伍7900隊，台灣大學獲得總決賽第三名
- 2013年、2014年，台灣大學獲得總決賽第四名
- 2020年：因COVID-19而延後一年

ICPC World Champions

year	champions
1977	Michigan State University (USA)
1978	Massachusetts Institute of Technology (USA)
1979	Washington University in St. Louis (USA)
1980	Washington University in St. Louis Louis (USA)
1981	University of Missouri–Rolla Louis (USA)
1982	Baylor University Louis (USA)
1983	University of Nebraska Louis (USA)
1984	Johns Hopkins University Louis (USA)
1985	Stanford University Louis (USA)
1986	California Institute of Technology Louis (USA)

year	champions
1987	Stanford University (USA)
1988	California Institute of Technology (USA)
1989	University of California, Los Angeles (USA)
1990	University of Otago (New Zealand)
1991	Stanford University (USA)
1992	University of Melbourne (Australia)
1993	Harvard University (USA)
1994	University of Waterloo (Canada)
1995	Albert-Ludwigs-Universitat (Germany)
1996	University of California, Berkeley (USA)

ICPC World Champions

year	place	country	university	site	team	final	champions
1997	San Jose		560		1100	50	Harvey Mudd College (USA)
1998	Atlanta			49	1250	54	Charles University (Czech)
1999	Eindhoven	59	839	63	1900	62	University of Waterloo (Canada)
2000	Orlando				2400	60	St. Petersburg State University (Russia)
2001	Vancouver	70	1079		2700	64	St. Petersburg State University (Russia)
2002	Honolulu	67	1300		3082	64	Shanghai Jiaotong University (China)
2003	Beverly Hills	68	1329	106	3835	70	Warsaw University (Poland)
2004	Prague	75	1411	127	3105	73	St. Petersburg State University of IT, Mechanics, and Optics (Russia
2005	Shanghai					78	Shanghai Jiaotong University (China)
2006	San Antonio	84	1737	183	5606	83	Saratov State University (Russia)

ICPC World Champions

year	place	country	university	site	team	final	champions
2007	Tokyo	82	1756	205	6099	88	Warsaw University (Poland)
2008	Banff		1821	213	6700	100	St. Petersburg State University of IT, Mechanics, and Optics (Russia)
2009	Stockholm	88	1838		7109	100	St. Petersburg State University of IT, Mechanics, and Optics (Russia)
2010	Harbin	82	1931	242	7900	103	Shanghai Jiaotong University (China)
2011	Orlando	88	2070	280	8305	100	Zhejiang University (China)
2012	Warsaw	85	2219		8478	112	St. Petersburg National Research University of IT, Mechanics, and Optics (Russia)
2013	St. Petersburg	91	2322		9800	119	St. Petersburg National Research University of IT, Mechanics, and Optics (Russia)
2014	Ekaterinburg	94	2286		10681	122	St. Petersburg State University (Russia)

ICPC World Champions

year	place	country	university	site	team	final	champions
2015	Marrakesh	102	2736	481		128	ITMO University (Russia)
2016	Phuket					128	St. Petersburg State University (Russia)
2017	Rapid City					133	ITMO University (Russia)
2018	Bejing					140	Moscow State University (Russia)
2019	Porto	111				140	Moscow State University (Russia)
2021	Moscow	104	3406				Nizhni Novgorod University(Russia)
2022	Dhaka						Massachusetts Institute of Technology (MIT)
2024	Luxor						Peking University (46th)
2024	Luxor						National Research University Higher School of Economics (HSE) (47th)
2024	Astana						Peking University (48th)
2025	Baku						

大學程式設計競賽組隊方式

- 每隊正好三人，共同使用一部電腦
- 基本要求(確定要求請見競賽規程)
  - 每位隊員最多可參加五年(每年最多兩個亞洲區域賽)
  - 每位隊員最多可參加兩次世界總決賽
- 隊員資格(確定資格請見競賽規程)，下列二者之一：
  - 每位隊員進入大學後，不得超過5年
  - 不得超過24歲(例如參加2025區域賽，須2002年之後出生)
- 競賽評分系統DOMjudge (或其他評分系統)

計分方式

- 比賽時間為5小時
- 每個題目結果只有「對」與「錯」
- 答對題數較多者，排名較前
- 答對題數相同者，以解題時間總和決定排名
- 解題時間為比賽開始至解題正確所花時間，再加上罰扣時間(每送出題解錯誤一次罰加20分鐘)
- 答錯的題目不計時間及罰扣時間
- 計分範例：甲隊開賽後10分鐘答對A題，15分鐘送出B題(但錯誤)，32分答對B題。總時間為10+32+20*1=62分

2014 ACM ICPC World Finals (122 Teams)

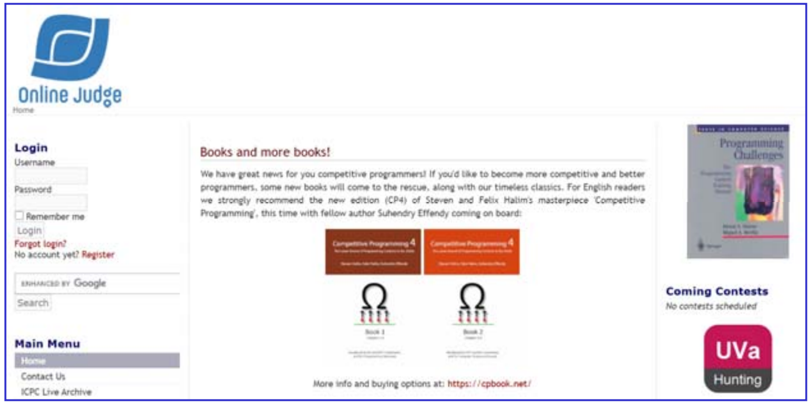
Place	University	Solved	Time	Last solved		Place	University	Solved	Time
1	St. Petersburg State University	7	1359	298		10	National Research University Higher School of Economics	4	428
2	Moscow State University	7	1398	290		11	Tsinghua University	4	444
3	Peking University	6	1275	297		12	Comenius University	4	454
4	National Taiwan University	6	1483	296		13	Belarusian State University	4	
5	University of Warsaw	5	796	266		13	New York University	4	
6	Shanghai Jiao Tong University	5	938	289		13	Taras Shevchenko Kiev National University	4	
7	The University of Tokyo	5	960	287		13	University of Electronic Science and Technology of China	4	
8	University of Zagreb	5	970	242		13	University of Wroclaw	4	
9	St. Petersburg National Research University of IT, Mechanics and Optics	5	1000	294		13	Zhejiang University	4	

2024 ICPC World Finals (141 Teams)

Place	University	Solved	Time		Place	University	Solved	Time
1	Peking University	9	935		9	National University	8	1112
2	Moscow Institute of Physics and Technology	9	1212		10	Zhejiang University	8	1166
3	Tsinghua University	9	1218		11	Massachusetts Institute of Technology (MIT)	8	1324
4	Tokyo Institute of Technology	9	1322		12	Swarthmore College	7	605
5	Korea Advanced Institute of Science and Technology(KAIST)	8	868					
6	National University of Singapore	8	934					
7	Beijing Jiaotong University	8	960					
8	The University of Tokyo	8	1031					

Online Judge

- 線上即時評分系統(電腦自動評分）
- 題目來源：ICPC
- 題目總數：超過4500題



https://onlinejudge.org/

Online Judge

- 統計每題被解的情形，讓學習者知道題目困難度

Title	Total Submissions / Solving %		Total Users / Solving %	
100 - The 3n + 1 problem	935603	26.65%	144134	73.86%
101 - The Blocks Problem	125304	25.86%	25228	70.27%
102 - Ecological Bin Packing	111030	39.41%	32393	87.25%
103 - Stacking Boxes	46143	26.73%	12354	61.27%
104 - Arbitrage	33726	25.23%	7170	67.99%
105 - The Skyline Problem	65333	26.07%	15687	69.23%
106 - Fermat vs. Pythagoras	29972	34.28%	7095	67.36%
107 - The Cat in the Hat	53265	21.73%	9225	68.28%
108 - Maximum Sum	74699	44.60%	24044	83.58%
109 - SCUD Busters	14201	32.37%	3772	68.35%
110 - Meta-Loopless Sorts	13199	32.18%	3640	66.13%
111 - History Grading	36031	45.61%	13641	77.89%
112 - Tree Summing	39611	28.30%	8167	78.18%
113 - Power of Cryptography	78790	40.97%	25019	89.10%
114 - Simulation Wizardry	8996	32.60%	2616	75.34%
115 - Climbing Trees	7711	35.24%	2319	81.37%
116 - Unidirectional TSP	70748	25.42%	12785	72.71%

The Format of One Problem

- General Description
- Input Format
- Output Format
- Sample Input
- Sample Output

題目難易程度分級

- 一顆星：學習完計算機概論之後即可解答(solved in 10 minutes
- 兩顆星：學習完資料結構之後才能解答或是苦工題(solved in 10~30 minutes)
- 三顆星：要有好的演算法或數學方法才能解答(solved in 30~100 minutes)
- 四顆星：要有特殊的演算法或是綜合多種演算法才能解答(solved in more than 100 minutes)
- 五顆星：超越四顆星的極特殊題目

已評星等列表

題目編號	星等	題目編號	星等	題目編號	星等	題目編號	星等	題目編號	星等	題目編號	星等
108	1	108	1	151	2	147	3	166	4	10021	5
118	1	118	1	245	2	532	3	481	4	10059	5
136	1	136	1	299	2	793	3	719	4	10119	5
147	3	256	1	374	2	908	3	10029	4	10206	5
151	2	264	1	495	2	926	3	10051	4	10330	5
166	4	272	1	572	2	928	3	10065	4	10418	5
245	2	305	1	612	2	929	3	10067	4	10904	5

何謂演算法

- Algorithm
- 解決問題的方法。將抽象的解法變成實際具體可行的方法或程式。
- 利用電腦解決問題的步驟

  Step 1: 明確定義問題（將其模式化）

  Step 2: 設計演算法，並估計所需時間

  Step 3: 撰寫程式，並加以測試

解決問題範例

- 問題：計算大學聯考英文之前標
- 明確定義：位於75%(前25%)之考生英文成績
- 演算法：

  Step 1: 將所有考生英文成績排序（由低至高）

  Step 2: 選出位於75%的成績
- 撰寫程式： ……

各種排序演算法所需時間比較

資料量	Bubble	Quick	Radix
500	0.004	0.053	0.000
1000	0.030	0.108	0.000
5000	0.850	0.636	0.004
10000	3.266	1.099	0.014
50000	94.449	5.745	0.100
100000	384.72	11.225	0.200

CPU: K6-2 350　　　時間單位：秒

演算法範例

【問題】將50元硬幣換成等值的1元、5元、10元硬幣的方法共有多少種？

【方法-1】

採用窮舉法，每種硬幣可能的個數如下：


```
i (10元):0,1,2,3,4,5
j (5 元):0,1,2,…,10
k (1 元):0,1,2,…,50
```

假設 i, j, k 分別代表10元、5元、1元的個數，則我們可以嘗試各種組合，並利用下面的判斷式：

`i*10 + j*5 + k = 50`

<執行迴圈次數> 6 * 11 * 51 = **3366**

```
main()
{
  int loop = 0, number = 0;
  int i, j, k;
  for (i = 0; i <= 5; i++)
    for (j = 0; j <= 10; j++)
      for (k = 0; k <= 50; k++)
      {
        loop++;
        if (i*10 + j*5+ k == 50)
          number++;
      }
  printf("共%d種,執行迴圈%d次\n",number,loop);
}
```

【執行結果】

共36種,執行迴圈3366次

【方法-2】

若 k 不為 5 之倍數，根本不可能轉換，所以只需考慮 k 為 5 之倍數的情況。

i (10元):0,1,2,3,4,5
j (5 元):0,1,2,…,10
k (1 元):0,5,10,…,50

<執行迴圈次數> 6 * 11 * 11 = 726


```
main()
{
  int loop = 0, number = 0;
  int i, j, k;
  for (i = 0; i <= 5; i++)
    for (j = 0; j <= 10; j++)
      for (k = 0; k <= 50; k+=5)
      {
        loop++;
        if (i*10 + j*5+ k == 50)
          number++;
      }
  printf("共%d種,執行迴圈%d次\n",number,loop);
}
```

【執行結果】

共36種,執行迴圈726次

【方法-3】

當 i*10 + j*5 + k = 50時，應立即跳出最內層迴圈，因為再變化 k 之值，i*10 + j*5 + k 均已大於 50。

<執行迴圈次數> 491

```c
main()
{
  int loop = 0, number = 0;
  int i, j, k;
  for (i = 0; i <= 5; i++)
    for (j = 0; j <= 10; j++)
      for (k = 0; k <= 50; k+=5)
      {
        loop++;
        if (i*10 + j*5+ k == 50)
        {
          number++; break;
        }
      }
  printf("共%d種,執行迴圈%d次\n",number,loop);
}
```

【執行結果】

共36種,執行迴圈491次

【方法-4】

當 i 和 j 之值固定後，k 之值只有唯一的選擇，因此不必考慮 k 的變化情形。

i=0,j可能為 0,1,2,…,10 (50-i*10)/5=10
i=1,j可能為 0,1,2,…,8 (50-i*10)/5=8
i=2,j可能為 0,1,2,…,6 (50-i*10)/5=6
.
.
.
i=5,j可能為 0 (50-i*10)/5=0

<執行迴圈次數> **36**


```c
main()
{
  int loop = 0, number = 0;
  int i, j;
  for (i = 0; i <= 5; i++)
    for (j = 0; j <= (50-i*10)/5; j++)
    {
      loop++;
      number++;
    }
  printf("共%d種,執行迴圈%d次\n",number,loop);
}
```

【執行結果】

共36種,執行迴圈36次

【方法-5】

由上一個方法知，當 i 的值固定後，j 的變化情形只有 (50-i*10)/5 種，因此只需對 i 做迴圈。

<執行迴圈次數> **6**

```c
main()
{
  int loop = 0, number = 0;
  int i;
  for (i = 0; i <= 5; i++)
  {
    loop++;
    number += (50-i*10)/5 + 1;
  }
  printf("共%d種,執行迴圈%d次\n",number,loop);
}
```

【執行結果】

共36種,執行迴圈6次

【方法-6】

我們計算的值其實是一個等差級數，即
11+9+7+…+1=6*(11+1)/2=36
將等差級數的公式寫成程式即可計算。

```
main()
{
    int number = 0, a, b, n = 50;
    a = n / 5 + 1;
    if (a % 2 == 0) b = 2;
    else b = 1;
    number = (a+b)*((a-b)/2+1)/2;
    printf("共%d種\n", number);
}
```

【執行結果】

共36種

Recursion

- Write a recursive C function to compute the sum of the elements of the array *a*.
- Solution :

```
int sum(int a[], int size)
/*size is the number of elements in array a[] */
{
    if(size == 0)
        return 0;
    else
        return a[size-1] + sum(a, size - 1);
}
```


- Wrong solution :

```
int sum(int a[], int size, int s)
   //size is the number of elements in array a[]
   // s is the sum of a[], initial value of s is 0
{
    if(size == 0)
        return s;
    else {
        s = s + a[size-1]
        return  sum(a, size - 1 , s);
}
```


- Determine the maximum of the contents of all nodes in a binary tree.
- Solution:

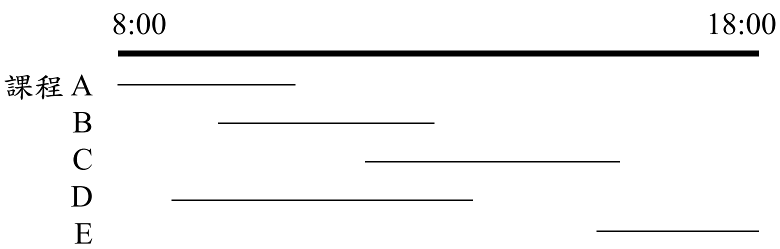
```
sturct nodetype {
    int info;
    struct nodetype *left;
    struct nodetype *right;
}
int max(struct nodetype *p)
{
   int a,b;
```

```
    if(p==NULL)
        return(-MAXINT); //MAXINT is infinite
    else {
        a=max(p->left);
        b=max(p->right);
        if (a>=b)
            return (p->info >= a ? p->info : a)
        else
            return (p->info >= b ? p->info : b)
    }
} /* end of max() */
```

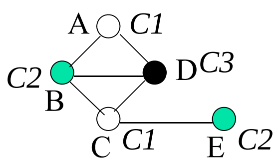

- Wrong solution:

```
int max(struct nodetype *p)
{
    if(p==NULL)
        return(-MAXINT); //MAXINT is infinite
    else {
        if (max(p->left) >= max(p->right)
            return (p->info >= max(p->left)
                    ? p->info : max(p->left))
        else
            return (p->info >= max(p->right)
                    ? p->info : max(p->right))
    }
} /* end of max() */
```


上課教室與圖形著色



區間圖形著色問題(interval graph coloring)：



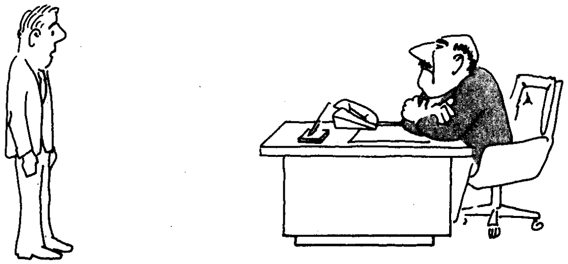
C1：第一個顏色

C2：第二個顏色

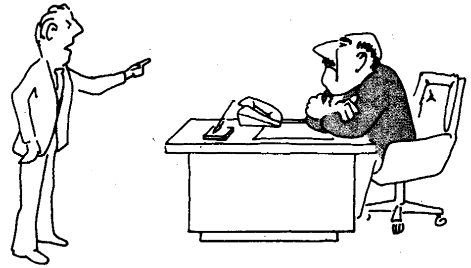
C3：第三的顏色

問題難易度

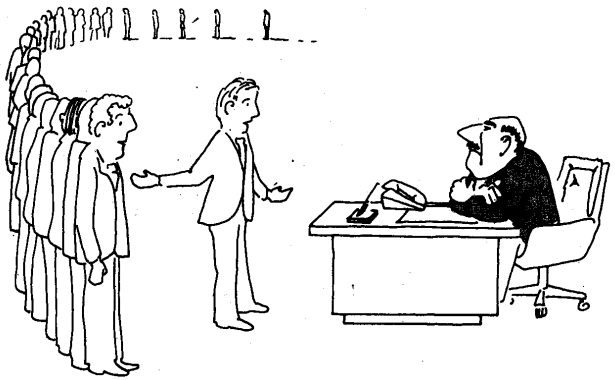
- 容易的問題：在多項式時間(polynomial time)可解決的問題
  - 如：排序，找最大值
- 困難的問題：NP-complete，NP-hard
  - 如：分割問題(Partition Problem)
  - 推銷員問題(Traveling Salesperson Problem)
- 不可解的問題：用演算法無法解決的問題
  - 如：停止問題(Halting Problem)
- lower bound：解題所需時間之底限



我想不出好方法，我可能太笨了！



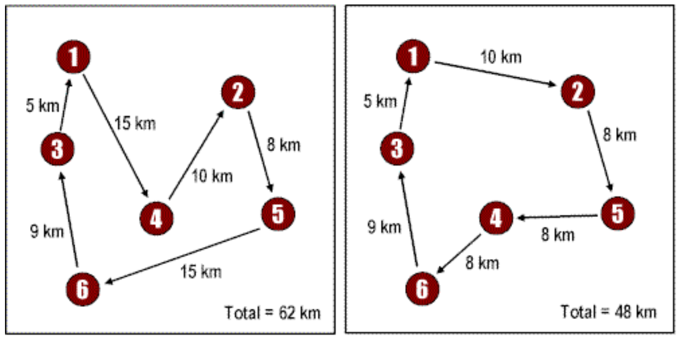
我想不出好方法，
因為不可能有這種好方法！



我想不出好方法，
因為這些名人專家也不會！

環球旅遊與推銷員問題

平面上給予 n 個點，從某一點出發，經過每個點一次，再回到出發點，而其<u>總長度為最短</u>



此為 <u>NP-complete</u> 問題

職棒比賽與分割問題

給予一個正整數的集合$A=\{a_1, a_2, \ldots , a_n\}$，是否可以將其分割成兩個子集合，而此兩個子集合的個別總和相等。

例：A = { 1, 3, 8, 4, 10}

可以分割：{1, 8, 4} 及 {3, 10}

此為 NP-complete 問題

股票投資與0/1 knapsack問題

有n個東西，每個東西有其個別價值(value)與重量(weight)另有一個袋子，其容量為M，如何選取某些東西，使其總重要不超過M，而其總價值為最高。

例：

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

M = 14

最佳(optimal)解法：P_1、P_2、P_3、P_5

0/1 knapsack問題為NP-complete

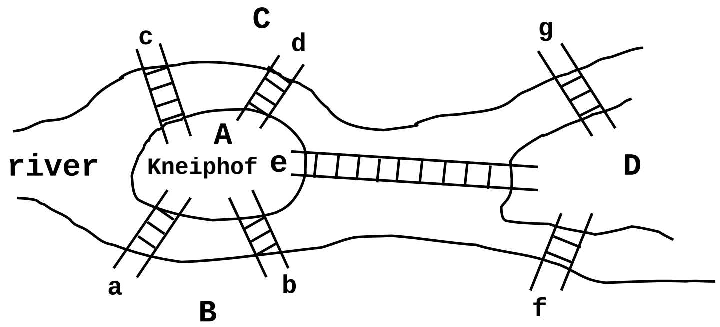
生物資訊與演算法

- 人類DNA序列由30億(3×10^9)個鹼基對(base pair)所組成
- 生物資訊之研究需要大量計算，如字串比對、序列排列、相似度計算、演化樹

Chapter 2

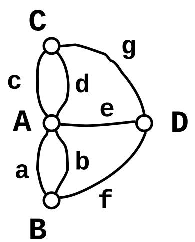
Graph Algorithms

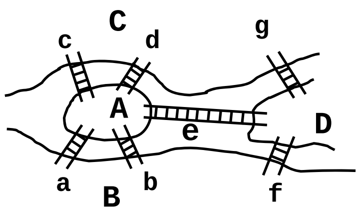
Koenigsberg bridge problem



- Is it possible to walk across all bridges exactly once from some land area and returning to the starting land area ?

Graph representation





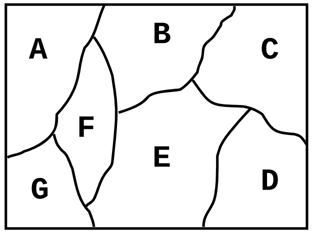
- vertex: land area
  edge: bridge
- The above problem is the Euler circuit (cycle) problem. In the circuit, each edge is visited exactly once from one source vertex and returning to the source vertex.

Euler path

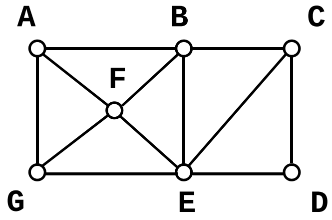
- Euler path: In a graph, each edge is visited exactly once, but it needs not return to the source vertex.
- Theorem: An undirected graph possesses an Euler path if and only if it is connected and has no, or exactly two, vertices that are of odd degree.
- Euler circuit <=> all vertices are of even degree
- An Euler cycle can be constructed in O(|E|) time, where E denotes the set of edges.

The four-color problem

map:



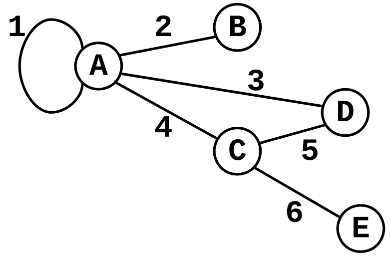
graph:



- Is it possible to use four colors to color any map such that no two adjacent areas have the same color?
- Five colors are sufficient, which was proved by Heawood in 1890.

Graph definition

- graph G = (V, E)
  V: a set of vertices (nodes)
  E: a set of edges ("arcs" used in digraphs)
- undirected graph:
  Each edge is an
  unordered pair
  of vertices.



- V = {A, B, C, D, E}
  E = {1, 2, 3, 4, 5, 6}

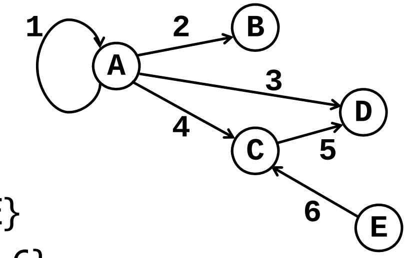
or {(A,A), (A,B), (A,D), (A,C), (C,D), (C,E)}

or (B,A), (D,A), (C,A), (D,C), (E,C)

- degree of a node u: # of edges linked to u.
  - e.g. degree of node B: 1 A: 5 C: 3
- A node u is adjacent to a node v: There is an edge from u to v
  - e.g. Node A is adjacent to nodes B, C and D.
- Each edge is incident to the two nodes which are linked by this edge.
  - e.g. Edge 2 is incident to nodes A and B.

Directed graph

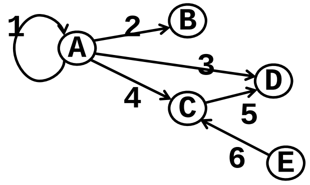
- directed graph (digraph):
  Each edge (arc) is an
  ordered pair
  of vertices.



- V = {A, B, C, D, E}
  E = {1, 2, 3, 4, 5, 6}

or {<A,A>, <A,B>, <A,D>, <A,C>, <C,D>, <E,C>}

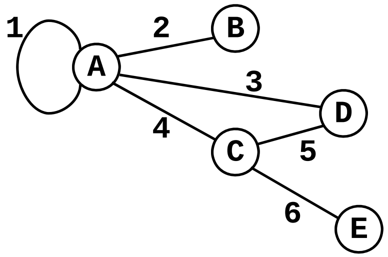
- indegree of a node u:
  # of edges which have u as the head. (entering u)
- outdegree of a node u:
  # of edges which have u as the tail (leaving u)
- e.g. indegree of node A: 1
  B: 1
  C: 2
  outdegree of node A: 4
  B: 0
  C: 1



- degree = indegree + outdegree

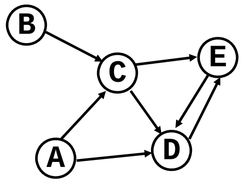
Paths in graphs

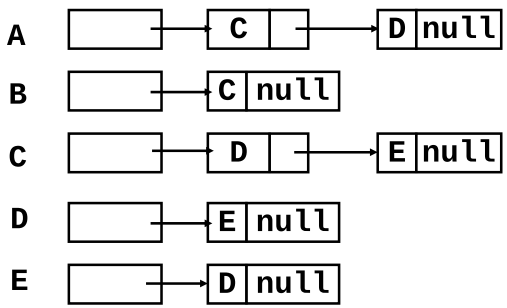
- A binary relation can be represented by a graph.
- path of length k from node u to node v:
  a sequence k+1 nodes $u_1, u_2, \ldots, u_{k+1}$
  (i) $u_1 = u$, $u_{k+1} = v$
  (ii) u_i and u_{i+1} are adjacent, for $1 \leqq i \leqq k$.
- e.g. (A,D,C,E) is a path of length 3



Graph representation (1)

- adjacency list:
  The nodes adjacent to one node are maintained by a linked list.

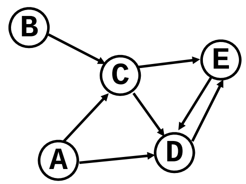




Graph representation (2)

- adjacency matrix:
  Each entry a_{ij} in the matrix has value 1 if and only if there is one edge connecting vertex v_i to vertex v_j.

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0



matrix $A = (a_{ij})_{1 \leqq i \leqq n,\ 1 \leqq j \leqq n}$

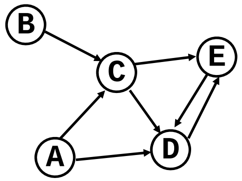
Transitive closure

- The reachability matrix for a graph G is the transitive closure T of the adjacency matrix of G.

  matrix $T = (t_{ij})_{1 \le i \le n,\ 1 \le j \le n}$

$$t_{ij} = \begin{cases} 1 & \text{If there is a path (not of length 0) from } v_i \text{ to } v_j. \\ 0 & \text{otherwise} \end{cases}$$

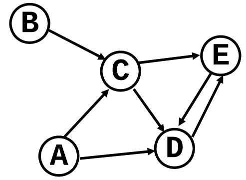
	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1



Construction of transitive closure

- The construction method for T:

$a_{ik} = 1$ and $a_{kj} = 1$
<=> a path of length 2 from v_i to v_j passing through v_k



	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

matrix A

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

A^2

path matrix of length 2

Calculation with matrix multiplication

- a_{ij} of A^2 is computed by:

$$\mathop{\mathrm{OR}}_{k=1}^{n}(a_{ik} \text{ and } a_{kj}) = \begin{cases} 1 & \text{a path of length 2 from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

- The above can be calculated by the matrix multiplication method.

boolean product		matrix multiplication
or	⟷	+
and	⟷	*

- meaning of (A or A^2)?

- The matrix multiplication can be applied repeatedly.

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

$A^3 = A^2 \times A$

$A^4 = A^3 \times A = (A^2)^2$

$A^5 = A^4 \times A$

- Finally, the following matrix is obtained.

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

$T = A$ or A^2 or A^3 or A^4 or A^5

path of length ≤ 5

- Matrix T is the transitive closure of matrix A.
- Required time: O(n) matrix multiplications

Connectivity

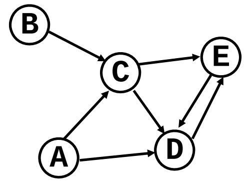
- Connectivity matrix C:

  reflexive and transitive closure of G

matrix $C = (c_{ij})_{1 \leq i \leq n,\ 1 \leq j \leq n}$

$$c_{ij} = \begin{cases} 1 & \text{if there is a path of length 0 or more from } v_i \text{ to } v_j. \\ 0 & \text{otherwise} \end{cases}$$

	A	B	C	D	E
A	1	0	1	1	1
B	0	1	1	1	1
C	0	0	1	1	1
D	0	0	0	1	1
E	0	0	0	1	1



Construction of connectivity

- Construct matrix B:

$b_{ij} = a_{ij}$, for $i \neq j$

$b_{ii} = 1$

that is,

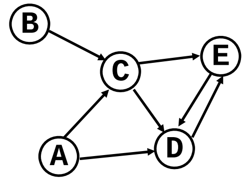
$$b_{ij} = \begin{cases} 1 & \text{a path of length 0 or 1 from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

- B^2 represents path of length 2 or less.
- Then, compute B^4, B^8,…B^{n-1}.
- Time: O(logn) matrix multiplications

(see the example on the next page)

B =

	A	B	C	D	E
A	1	0	1	1	0
B	0	1	1	0	0
C	0	0	1	1	1
D	0	0	0	1	1
E	0	0	0	1	1



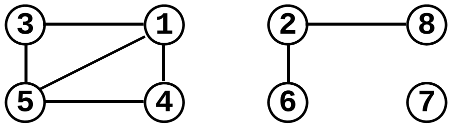
B^2 =

	A	B	C	D	E
A	1	0	1	1	1
B	0	1	1	1	1
C	0	0	1	1	1
D	0	0	0	1	1
E	0	0	0	1	1

$C = B^4 = (B^2)^2$, same as B^2

Connected components

- For every pair of nodes in a connected component, there is a path connecting them.

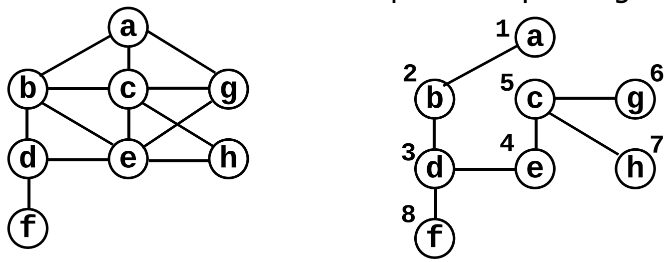


- This undirected graph consists of 3 connected components.
- Algorithms for solving the connected component problem:
  - connectivity matrix
  - depth-first search
  - breadth-first search

Depth-first search (DFS)

- Depth-first search (traversal)

depth-first spanning tree

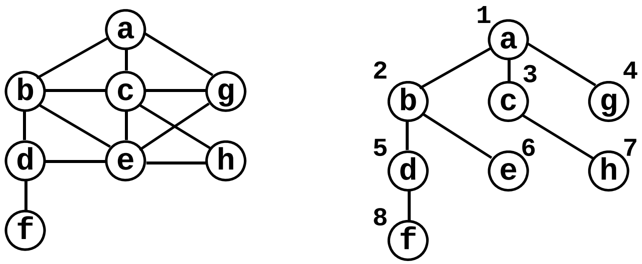


- depth-first order: a b d e c g h f
- It can be solved by a stack.
- Time: O(|E|)

Breadth-first search (BFS)

- Breadth-first search (traversal)

breadth-first spanning tree



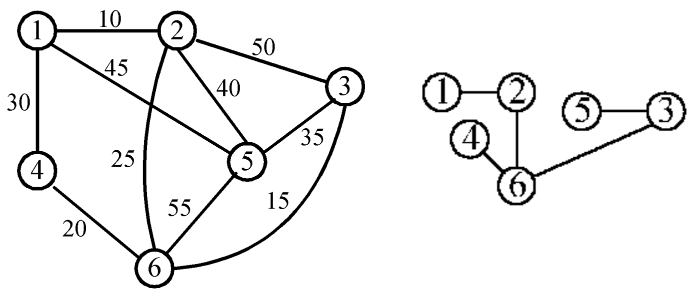
- breadth-first order: a b c g d e h f
- It can be solved by a queue.
- Time: O(|E|)

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



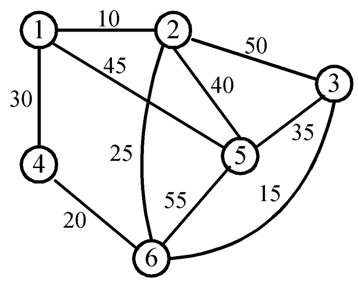
Kruskal's algorithm for finding MST

Step 1: Sort all edges into nondecreasing order.

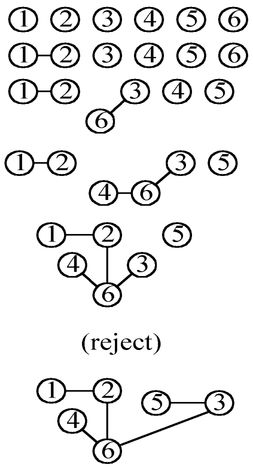
Step 2: Add the next smallest weight edge to the forest if it will not cause a cycle.

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

An example of Kruskal's algorithm



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



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

- $$\begin{aligned} &A(1,j) = 2^j && \text{for } j \ge 1 \\ &A(i,1) = A(i-1,2) && \text{for } i \ge 2 \\ &A(i,j) = A(i-1, A(i,j-1)) && \text{for } i, j \ge 2 \end{aligned}$$

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

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

Inverse of Ackermann's function

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

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

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

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

Prim's algorithm for finding MST

Step 1: x ∈ V, Let A = {x}, B = V - {x}.

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

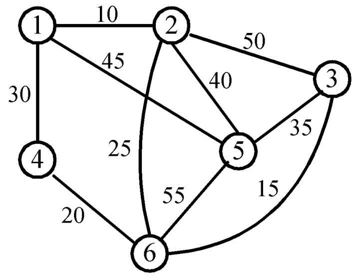
Step 3: Put (u, v) in the tree. A = A ∪ {v}, B = B - {v}

Step 4: If B = ∅, stop; otherwise, go to Step 2.

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

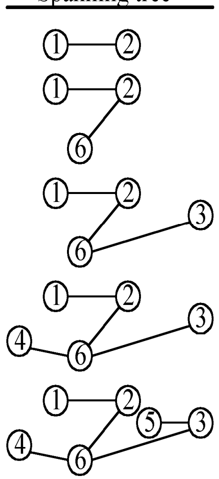
(see the example on the next page)

An example for Prim's algorithm



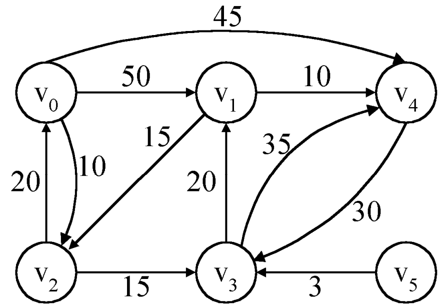
Edge	Cost
(1,2)	10
(2,6)	25
(3,6)	15
(6,4)	20
(3,5)	35

Spanning tree



The single-source shortest path problem

- shortest paths from v_0 to all destinations

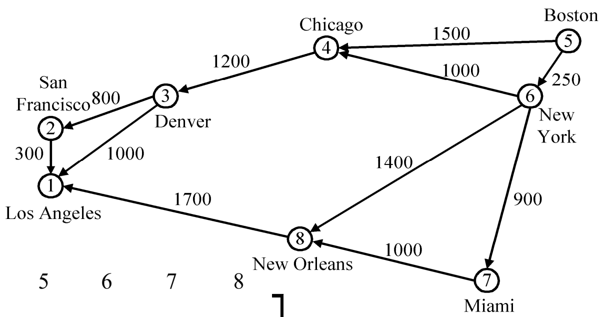


(a)

	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

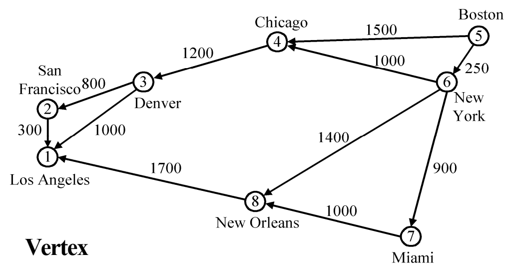
(b)

Dijkstra's algorithm



In the cost adjacency matrix, all entries not shown are +∞.

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0



Iteration	S	Vertex Selected	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Initial		----								
1	5	6	+∞	+∞	+∞	1500	0	250	+∞	+∞
2	5,6	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	5,6,7	4	+∞	+∞	+∞	1250	0	250	1150	1650
4	5,6,7,4	8	+∞	+∞	2450	1250	0	250	1150	1650
5	5,6,7,4,8	3	3350	+∞	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|.

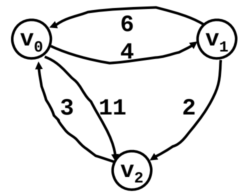
All pairs shortest paths

- The all pairs shortest path problem can be solved by a dynamic programming method.
- a^k_{ij}: the length of a shortest path from v_i to v_j going through no vertex of label greater than k.

$$
\begin{aligned}
&\text{for } k = 0 \text{ to } n-1 \text{ do} \\
&\quad \text{for } i = 0 \text{ to } n-1 \text{ do} \\
&\qquad \text{for } j = 0 \text{ to } n-1 \text{ do} \\
&\qquad\quad a^k_{ij} = \min\left\{ a^{k-1}_{ij}, a^{k-1}_{ik} + a^{k-1}_{kj} \right\} \\
&\qquad\qquad \text{where } a^{-1}_{ij} = \text{weight of edge } (i, j)
\end{aligned}
$$

- Time complexity: $O(n^3)$, n = |V|.

An example for all pairs shortest paths



$$A = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

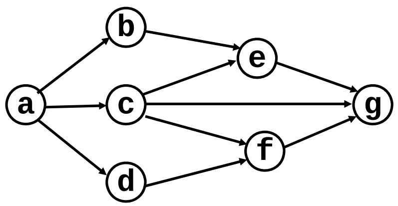
$$A^{(0)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \qquad A^{(1)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

shortest paths

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Topological order

- topological order (topological sort) for acyclic digraphs



topological order:

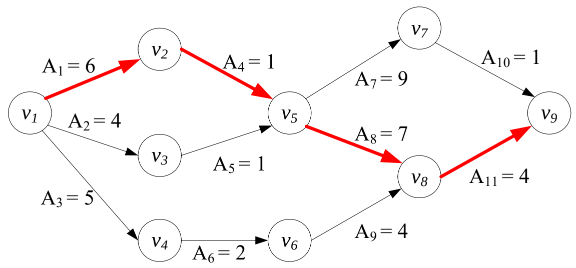
a b c d e f g
a b c e d f g
a d c f b e g

⋮

Critical path

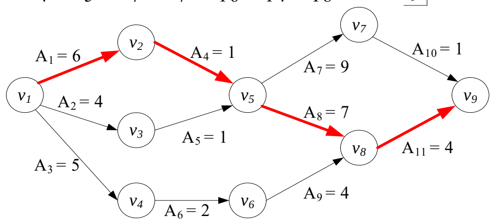
- critical path for acyclic digraphs
- maximum time needed for the given activities

Activity	Time	Prior activities
A_1	6	--
A_2	4	--
A_3	5	--
A_4	1	A_1
A_5	1	A_2
A_6	2	A_3
A_7	9	A_4, A_5
A_8	7	A_4, A_5
A_9	4	A_6
A_{10}	1	A_7
A_{11}	4	A_8, A_9



Action of modified topological order

output	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Stack
	0	0	0	0	0	0	0	0	0	1
v_1	0	6	4	5	0	0	0	0	0	432
v_4	0	6	4	5	0	7	0	0	0	632
v_6	0	6	4	5	0	7	0	11	0	32
v_3	0	6	4	5	5	7	0	11	0	2
v_2	0	6	4	5	7	7	0	11	0	5
v_5	0	6	4	5	7	7	16	14	0	78
v_7	0	6	4	5	7	7	16	14	17	8
v_8	0	6	4	5	7	7	16	14	18	9
v_9										



Chapter 3

The Greedy Method

A simple example

- <u>Problem</u>: Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- <u>Algorithm</u>**:**

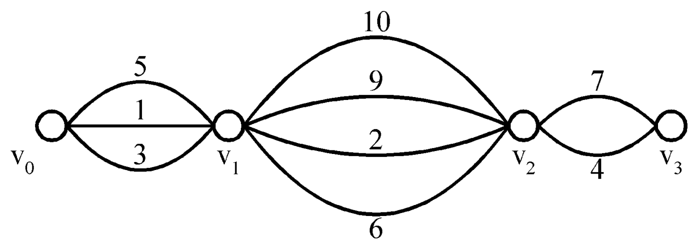
```
FOR i = 1 to k
    pick out the largest number and
    delete this number from the input.
ENDFOR
```


The greedy method

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

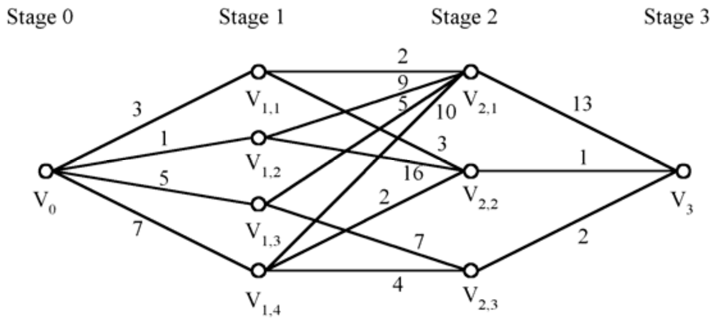
Shortest paths on a special graph

- <u>Problem</u>: Find a shortest path from v_0 to v_3.
- The <u>greedy</u> 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.

The activity selection problem

- Problem: n activities, S = {1, 2, …, n}, each activity i has a start time s_i and a finish time f_i, $s_i \le f_i$.
- Activity i occupies time interval $[s_i, f_i]$.
- i and j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$.
- The problem is to select a maximum-size set of mutually compatible activities

Example:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

The solution set = {1, 4, 8, 11}

Algorithm:

Step 1: Sort f_i into nondecreasing order. After sorting, $f_1 \le f_2 \le f_3 \le \ldots \le f_n$.

Step 2: Add the next activity i to the solution set if i is compatible with each in the solution set.

Step 3: Stop if all activities are examined. Otherwise, go to step 2.

Time complexity: O(nlogn)

Solution of the example:

i	s_i	f_i	accept
1	1	4	Yes
2	3	5	No
3	0	6	No
4	5	7	Yes
5	3	8	No
7	6	10	No
8	8	11	Yes
9	8	12	No
10	2	13	No
11	12	14	Yes

Solution = {1, 4, 8, 11}

Job sequencing with deadlines

- Problem: n jobs, S={1, 2, …, n}, each job i has a deadline $d_i \geq 0$ and a profit $p_i \geq 0$. We need one unit of time to process each job and we can do at most one job each time. We can earn the profit p_i if job i is completed by its deadline.

i	1	2	3	4	5
p_i	20	15	10	5	1
d_i	2	2	1	3	3

The optimal solution = {1, 2, 4}.
The total profit = 20 + 15 + 5 = 40.

Algorithm:

Step 1: Sort p_i into nonincreasing order. After sorting $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n$.

Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot [r-1, r], where r is the largest integer such that $1 \leq r \leq d_i$ and [r-1, r] is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

Time complexity: $O(n^2)$

e.g.

i	p_i	d_i	
1	20	2	assign to [1, 2]
2	15	2	assign to [0, 1]
3	10	1	reject
4	5	3	assign to [2, 3]
5	1	3	reject

solution = {1, 2, 4}

total profit = 20 + 15 + 5 = 40

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: $x_1 = 0$, $x_2 = 1$, $x_3 = 1/2$
  total profit $= 24 + 7.5 = 31.5$

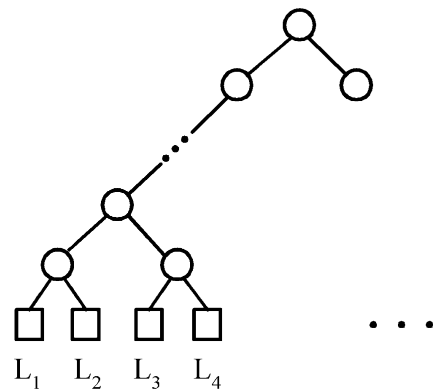
The 2-way merging problem

- # 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 ?

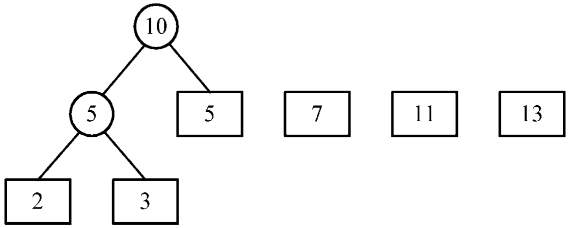
Extended binary trees

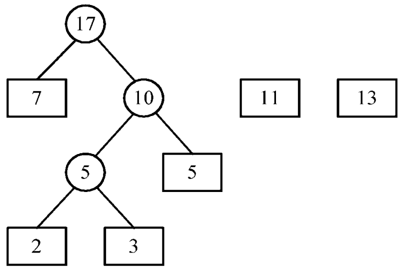
- An extended binary tree representing a 2-way merge

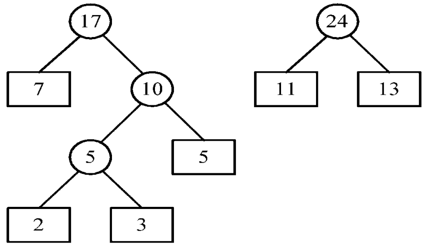


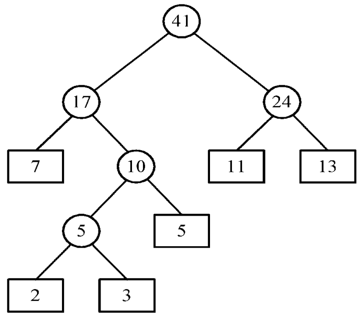
An example of 2-way merging

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









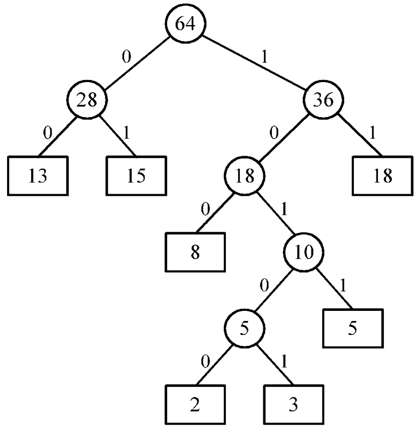
- Time complexity for generating an optimal extended binary tree:O(n log n)
- Using min-heap

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



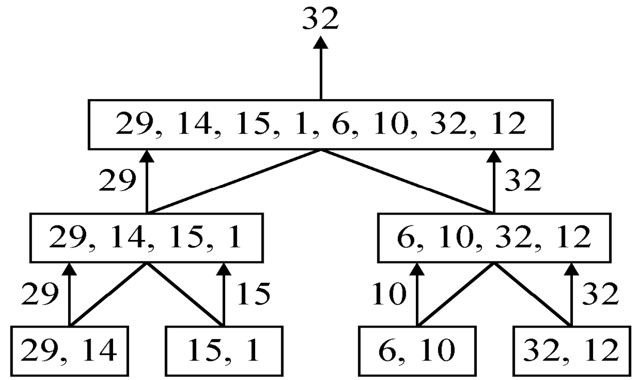
A Huffman code Tree

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): # of comparisons

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

- Calculation of T(n):

Assume $n = 2^k$,

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

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

Binary search

- e.g. 2 4 5 6 7 8 9

search 7: needs 3 comparisons

- time: O(log n)
- The binary search can be used only if the elements are sorted and stored in an array.

Algorithm binary-search

Input: A sorted sequence of n elements stored in an array.
Output: The position of x (to be searched).
Step 1: If only one element remains in the array, solve it directly.
Step 2: Compare x with the middle element of the array.
Step 2.1: If x = middle element, then output it and stop.
Step 2.2: If x < middle element, then recursively solve the problem with x and the left half array.
Step 2.3: If x > middle element, then recursively solve the problem with x and the right half array.

Algorithm BinSearch(a, low, high, x)

```
// a[]: sorted sequence in nondecreasing order
// low, high: the bounds for searching in a []
// x: the element to be searched
// If x = a[j], for some j, then return j else return –1
  if (low > high) then return –1          // invalid range
  if (low = high) then                    // if small P
    if (x == a[i]) then return i
    else return -1
  else                // divide P into two smaller subproblems
    mid = (low + high) / 2
    if (x == a[mid]) then return mid
    else if (x < a[mid]) then
         return BinSearch(a, low, mid-1, x)
    else return BinSearch(a, mid+1, high, x)
```

Quicksort

- Sort into nondecreasing order

```
[26   5  37   1  61  11  59  15  48  19]
[26   5  19   1  61  11  59  15  48  37]
[26   5  19   1  15  11  59  61  48  37]
[11   5  19   1  15] 26 [59  61  48  37]
[11   5   1  19  15] 26 [59  61  48  37]
[ 1   5] 11 [19  15] 26 [59  61  48  37]
  1   5  11  15  19  26 [59  61  48  37]
  1   5  11  15  19  26 [59  37  48  61]
  1   5  11  15  19  26 [48  37] 59 [61]
  1   5  11  15  19  26  37  48  59  61
```


Algorithm Quicksort

Input: A set S of n elements.

Output: The sorted sequence of the inputs in nondecreasing order.

Step 1: If $|S| \leq 2$, solve it directly.

Step 2: (Partition step) Use a pivot to scan all elements in S. Put the smaller elements in S_1, and the larger elements in S_2.

Step 3: Recursively solve S_1 and S_2.

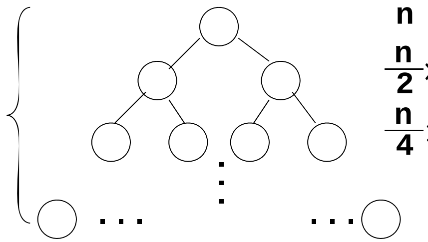
Time complexity of Quicksort

- time in the worst case:

  $(n-1)+(n-2)+...+1 = n(n-1)/2= O(n^2)$
- time in the best case:

  In each partition, the problem is always divided into two subproblems with almost equal size.

$\log_2 n$



n

$\frac{n}{2} \times 2 = n$

$\frac{n}{4} \times 4 = n$

Time complexity of the best case

- T(n): time required for sorting n elements
- $T(n) \leq cn+2T(n/2)$, for some constant c.

$$
\begin{aligned}
&\leq cn+2(c \cdot n/2 + 2T(n/4)) \\
&\leq 2cn + 4T(n/4) \\
&\quad \dots \\
&\leq cn\log_2 n + nT(1) = O(n\log n)
\end{aligned}
$$

Two-way merge

- Merge two sorted sequences into a single one.

```
[25  37  48  57][12  33  86  92]
              ⇩ merge
[12  25  33  37  48  57  86  92]
```

- time complexity: O(m+n),
  m and n: lengths of the two sorted lists

Merge sort

- Sort into nondecreasing order

```
        [25][57][48][37][12][92][86][33]
pass 1
        [25  57][37  48][12  92][33  86]
pass 2
        [25  37  48  57][12  33  86  92]
pass 3
        [12  25  33  37  48  57  86  92]
```

- $\log_2 n$ passes are required.
- time complexity: O(nlogn)

Algorithm Merge-Sort

Input: A set S of n elements.

Output: The sorted sequence of the inputs in nondecreasing order.

Step 1: If $|S| \leq 2$, solve it directly.

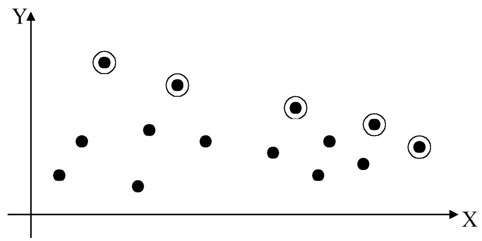
Step 2: Recursively apply this algorithm to solve the left half part and right half part of S, and the results are stored in S_1 and S_2, respectively.

Step 3: Perform the two-way merge scheme on S_1 and S_2.

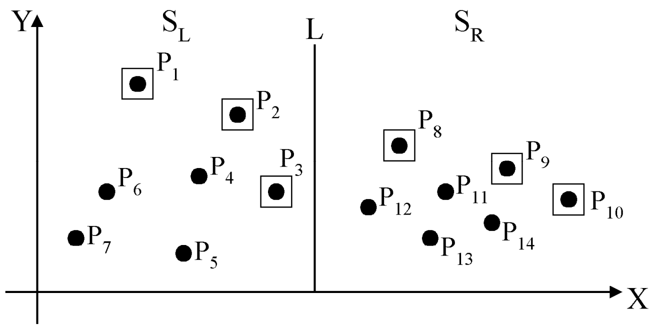
2-D maxima finding problem

- **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:
$C(n,2)=O(n^2)$



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 y_R.

- Time complexity: T(n)
  Step 1: O(n)
  Step 2: 2T(n/2)
  Step 3: O(n)

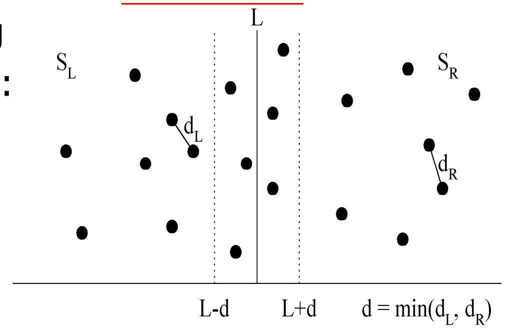
$$T(n)=\begin{cases} 2T(n/2)+O(n)+O(n) & , n > 1 \\ 1 & , n = 1 \end{cases}$$

Assume $n = 2^k$

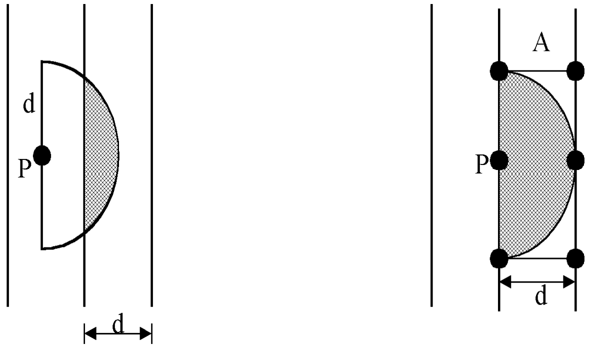
$T(n) = O(n \log n)$

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)

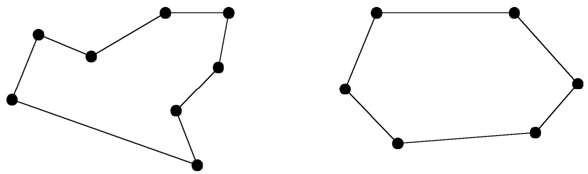
  Steps 2~5:

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

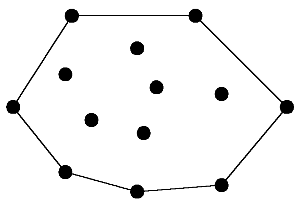
$\Rightarrow$T(n) = O(n log n)

The convex hull problem

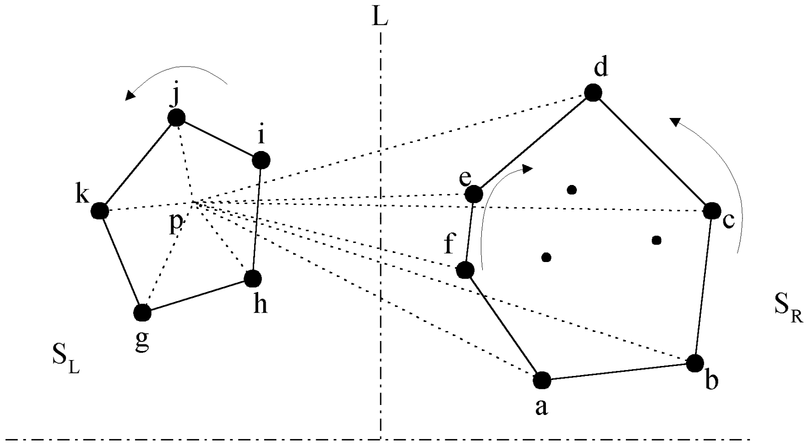
concave polygon: convex polygon:



- 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:

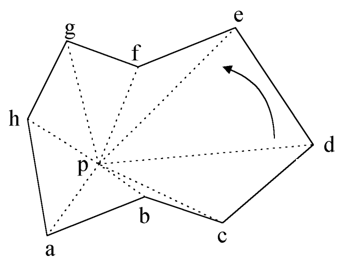


- The merging procedure:

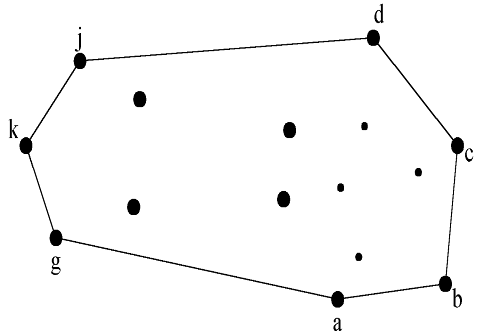
1. Select an interior point p.
2. There are 3 sequences of points which have increasing polar angles with respect to p.
   (1) g, h, i, j, k
   (2) a, b, c, d
   (3) f, e
3. Merge these 3 sequences into 1 sequence: g, h, a, b, f, c, e, d, i, j, k.
4. Apply Graham scan to examine the points one by one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

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

  $= O(n \log n)$

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

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

$$\boxed{\begin{aligned} C_{11} &= A_{11} B_{11} + A_{12} B_{21} \\ C_{12} &= A_{11} B_{12} + A_{12} B_{22} \\ C_{21} &= A_{21} B_{11} + A_{22} B_{21} \\ C_{22} &= A_{21} B_{12} + A_{22} B_{22} \end{aligned}}$$

- $C_{11} = P + S - T + V$
  $C_{12} = R + T$
  $C_{21} = Q + S$
  $C_{22} = P + R - Q + U$

Time complexity

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

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

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

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

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

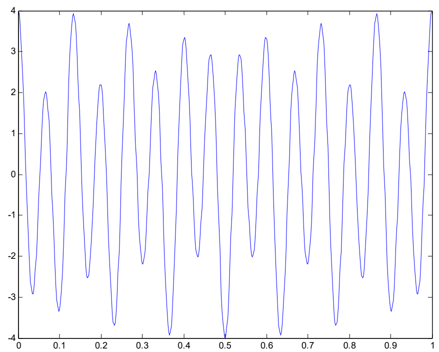
- Discrete Fourier transform(DFT)

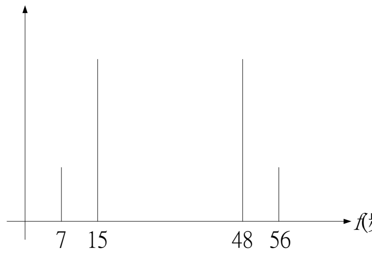
Given $a_0, a_1, \ldots, a_{n-1}$, compute

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

DFT and waveform(1)

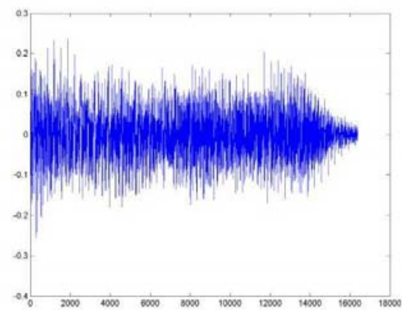
- Any periodic waveform can be decomposed into the linear sum of sinusoid functions (sine or cosine).



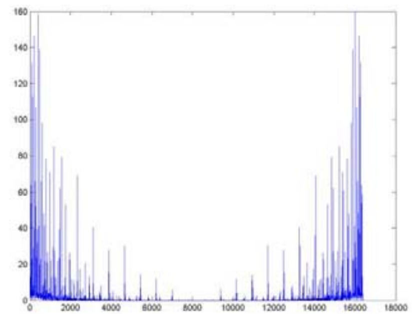


$$f(t) = \cos(2\pi(7)t) + 3\cos(2\pi(15)t) + 3\cos(2\pi(48)t) + \cos(2\pi(56)t)$$

DFT and waveform (2)



The waveform of a music signal of 1 second

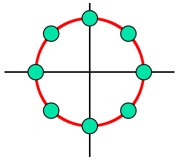


The frequency spectrum of the music signal with DFT

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, $w=e^{2\pi/4}$, w^4=1, w^2=-1

  $b_0=a_0+a_1+a_2+a_3$

  $b_1=a_0+a_1w+a_2w^2+a_3w^3$

  $b_2=a_0+a_1w^2+a_2w^4+a_3w^6$

  $b_3=a_0+a_1w^3+a_2w^6+a_3w^9$
- 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

- n=8, $w=e^{2\pi/8}$, 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 $w=e^{i2\pi/n}$ (assume n is even.)

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

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

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 $0 \le j \le n/2 - 1$.

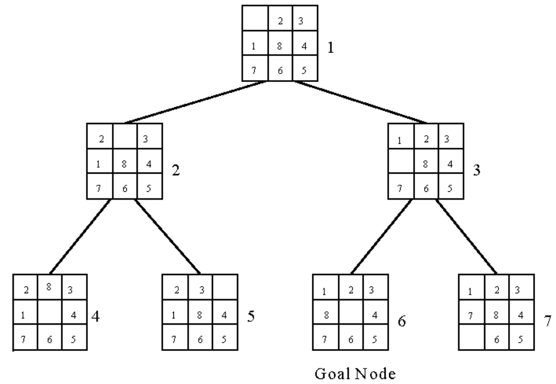
- Time complexity:
  $T(n) = 2T(n/2) + O(n)$
  $= O(n \log n)$

Chapter 5

Tree Searching Strategies

Breadth-first search (BFS)

- 8-puzzle problem



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

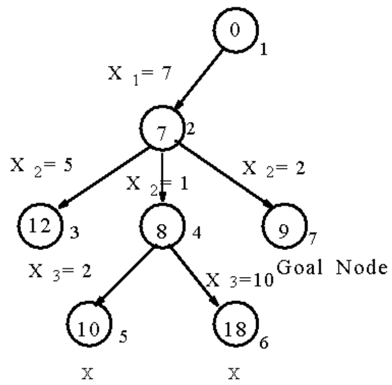
Depth-first search (DFS)

- e.g. sum of subset problem

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

  ∃ S' ⊆ S ∋ 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

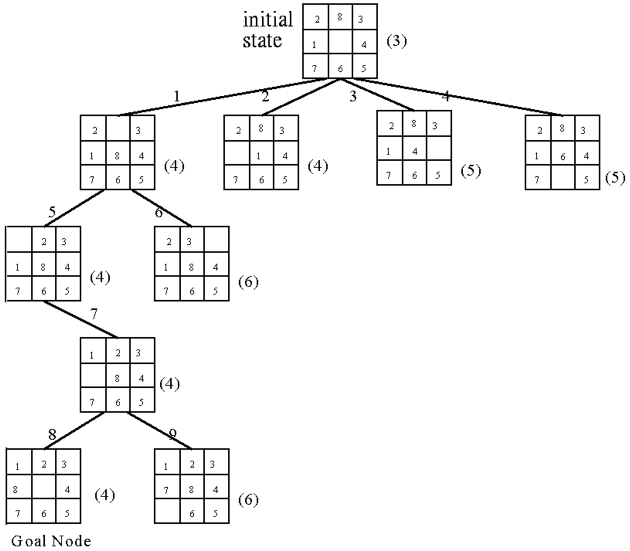
  The method selects the locally optimal node to expand.

- e.g. 8-puzzle problem

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

  where $d(n)$ is the depth of node n

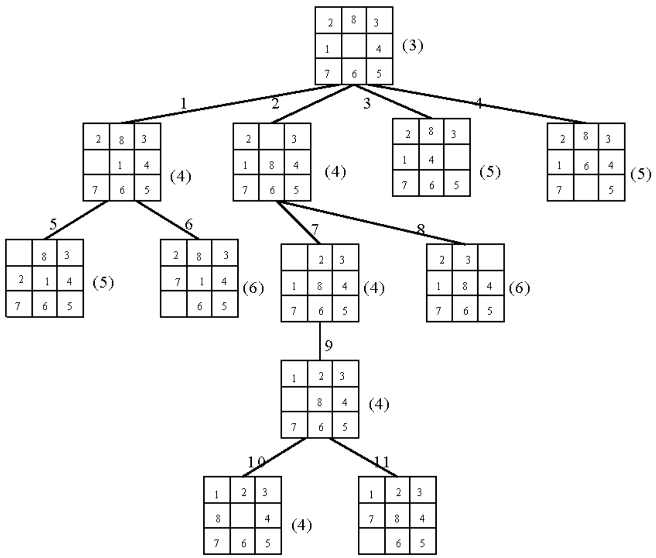
  $w(n)$ is # of misplaced tiles in node n.



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.



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.

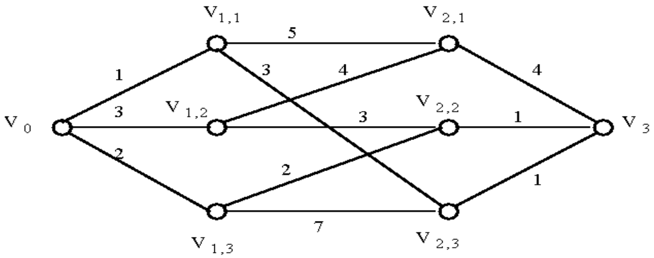
Step2: Remove the first element from the list. Expand the first element. If one of the descendants of the first element is a goal node, then stop; otherwise, add the descendants into the list.

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.

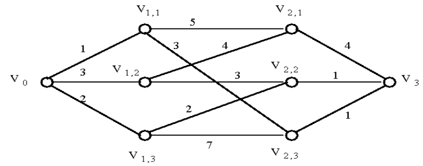
Branch-and-bound strategy

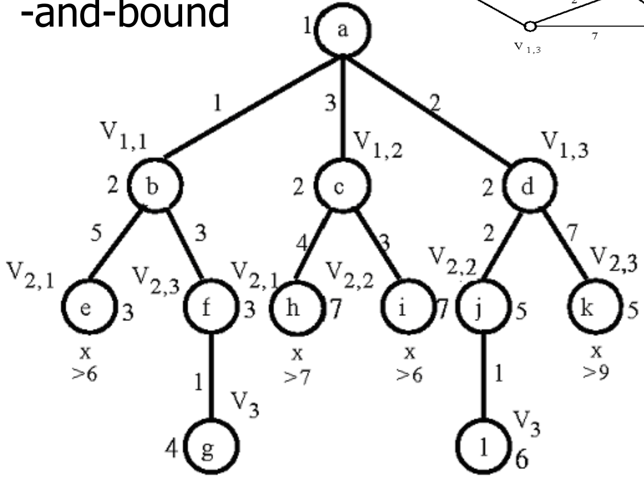
- This strategy can be used to efficiently solve optimization problems.
- e.g.



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

$$\begin{array}{ccc} J_1 & & J_2 \\ \downarrow & \searrow & \downarrow \\ J_3 & & J_4 \end{array}$$

- 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

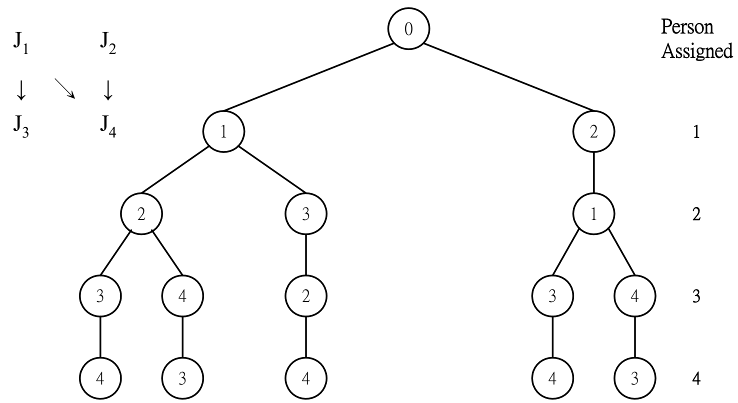
  J_1, J_3, J_2, J_4

  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.

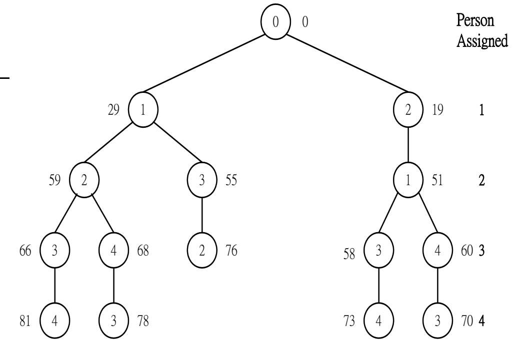


Cost matrix

- 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

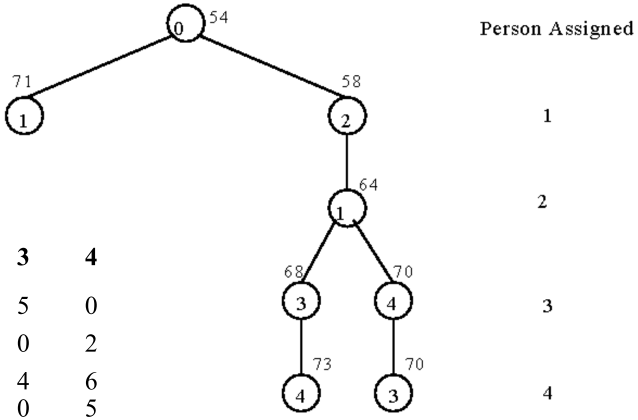
- Reduced cost matrix

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

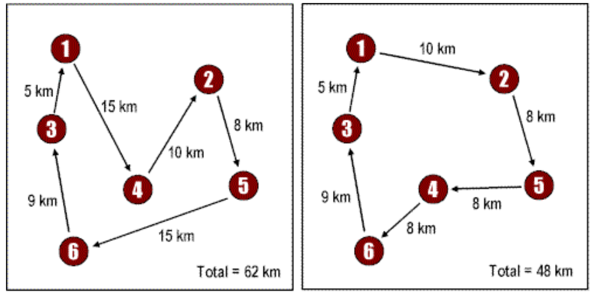
- Bounding of subsolutions:



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

- Given a set of points and their pairwise distances, the task is to find a shortest tour that visits each point exactly once.
- It is NP-complete.



The traveling salesperson optimization problem

- 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)

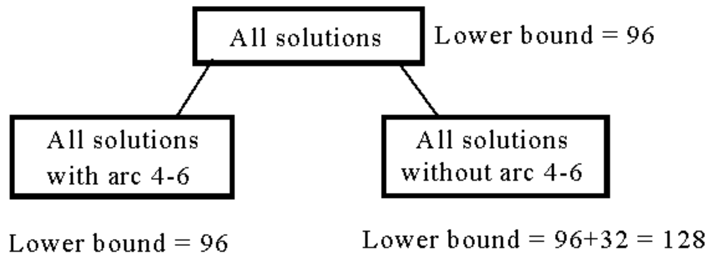
Reduced: 84

- 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.

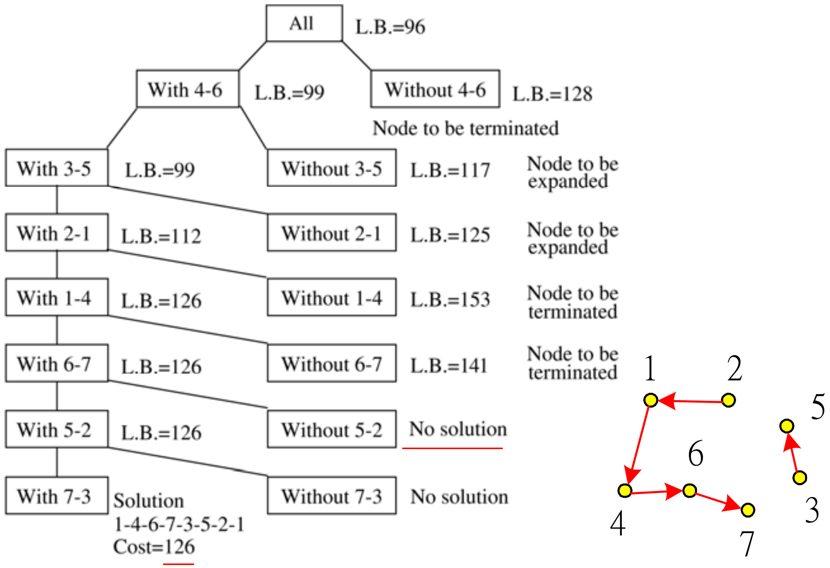
i \ j	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	3	21	56	7	∞	28
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

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)

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

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

The problem is modified:

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

- 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 $Y = -\sum_{i=1}^{n} P_i X_i$,

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

$$\Rightarrow Y' \le Y$$

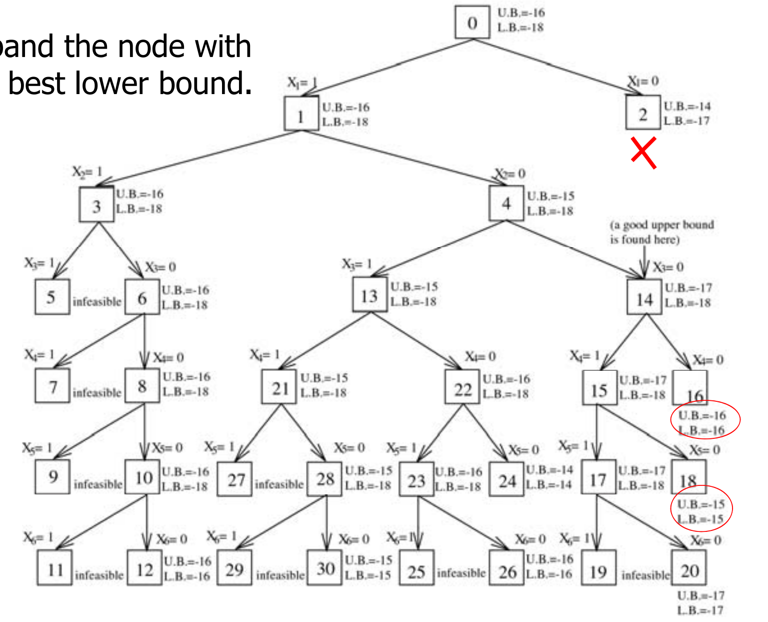
Upper bound and lower bound

- 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)

$\Rightarrow -18 \leq$ 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.



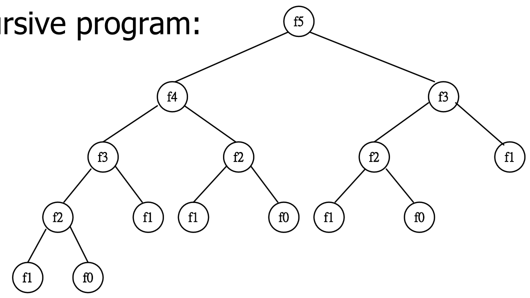
0/1 knapsack problem solved by branch-and-bound strategy.

Chapter 7

Dynamic Programming

Fibonacci sequence

- Fibonacci sequence: 0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , …
  $F_i = i$ if $i \leq 1$
  $F_i = F_{i-1} + F_{i-2}$ if $i \geq 2$
- Solved by a recursive program:



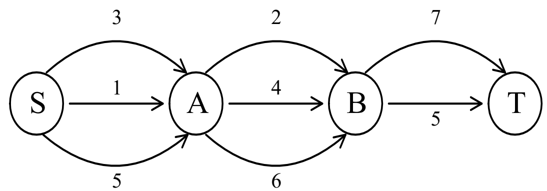
- Much replicated computation is done.
- It should be solved by a simple loop.

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

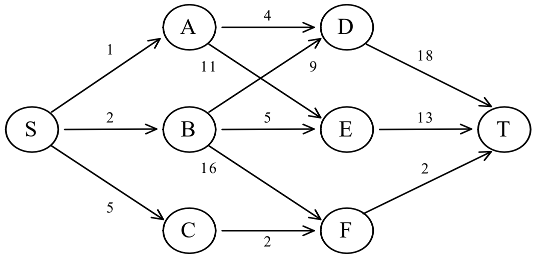
- To find a shortest path in a multi-stage graph



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

The shortest path in multistage graphs

- e.g.

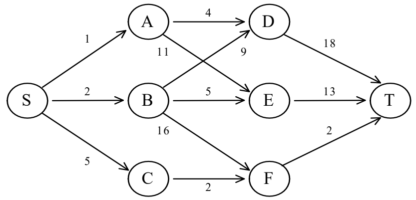


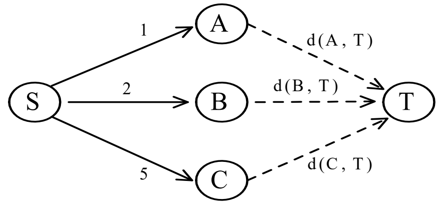
- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:

  (S, C, F, T) 5+2+2 = 9.

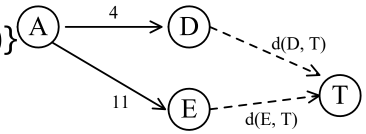
Dynamic programming approach

- Dynamic programming approach (forward approach):

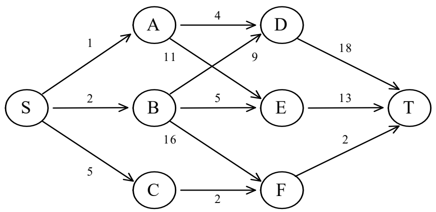


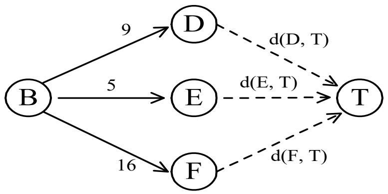


- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
- d(A,T) = min{4+d(D,T), 11+d(E,T)}
  = min{4+18, 11+13} = 22.



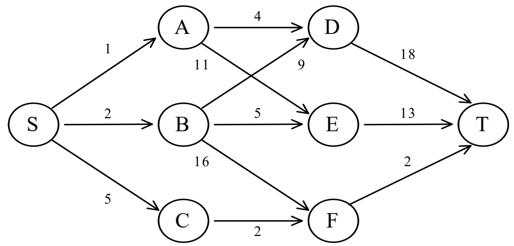
- d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
  = min{9+18, 5+13, 16+2} = 18.





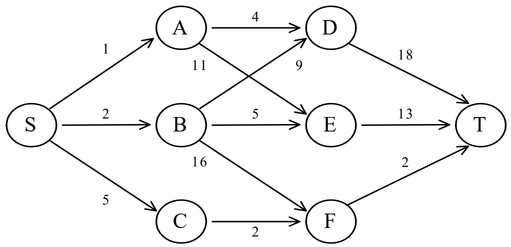
- d(C, T) = min{ 2+d(F, T) } = 2+2 = 4
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
  = min{1+22, 2+18, 5+4} = 9.
- The above way of reasoning is called backward reasoning.

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 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).

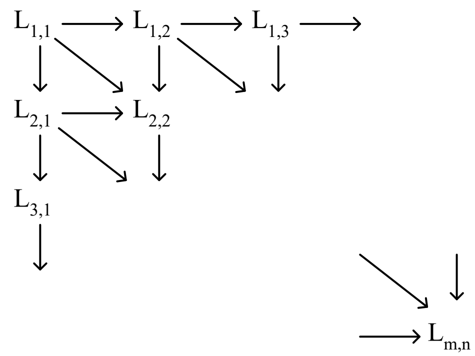
e.g. ad, ac, bac, acad, bacad, bcd.

- 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.

The edit distance problem

- 3 edit operations: insertion, deletion, replacement
- e.g string A='vintner', string B='writers'

```
v intner
wri t ers
RIMDMDMMI
```

M: match, I: insert, D:delete, R: replace

- The edit cost of each I, D, or R is 1.
- The edit distance between A and B: 5.

The edit distance algorithm

- Let A = $a_1\ a_2 \ldots a_m$ and B = $b_1\ b_2 \ldots b_n$
- Let $D_{i,j}$ denote the edit distance of $a_1\ a_2 \ldots a_i$ and $b_1\ b_2 \ldots b_j$.

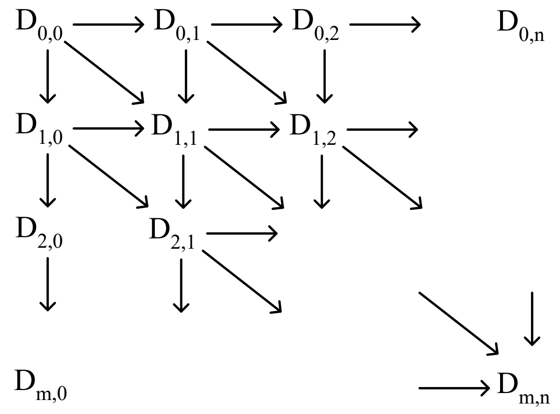
$D_{i,0} = i, \quad 0 \le i \le m$

$D_{0,j} = j, \quad 0 \le j \le n$

$D_{i,j} = \min\{D_{i-1,j} + 1, D_{i,j-1} + 1, D_{i-1,j-1} + t_{i,j}\}, 1 \le i \le m, 1 \le j \le n$

where $t_{i,j} = 0$ if $a_i = b_j$ and $t_{i,j} = 1$ if $a_i \ne b_j$.

- The dynamic programming approach for calculating the distance matrix:



- Time complexity: O(mn)

e.g. A='vintner', B='writers'

The 3 optimal alignments :

```
v-intner-     -:gap
wri-t-ers

-vintner-
wri-t-ers

vintner-
writ-ers
```

		w	r	i	t	e	r	s
	0	← 1	2	3	4	5	6	7
v	1	1	← 2	3	4	5	6	7
i	2	2	2	2	3	4	5	6
n	3	3	3	3	3	4	5	6
t	4	4	4	4	3	4	5	6
n	5	5	5	5	4	4	5	6
e	6	6	6	6	5	4	5	6
r	7	7	6	7	6	5	4	← 5

0/1 knapsack problem

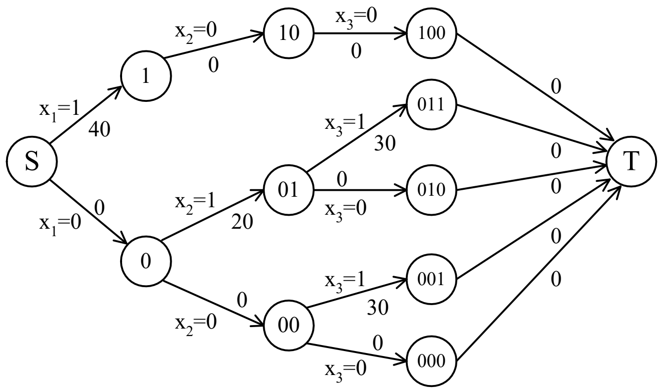
- n objects , weight $W_1, W_2, \ldots, W_n$
  profit $P_1, P_2, \ldots, P_n$
  capacity M
  maximize $\sum_{1 \le i \le n} P_i x_i$
  subject to $\sum_{1 \le i \le n} W_i x_i \le M$
  x_i = 0 or 1, $1 \le i \le n$
- e. g.

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

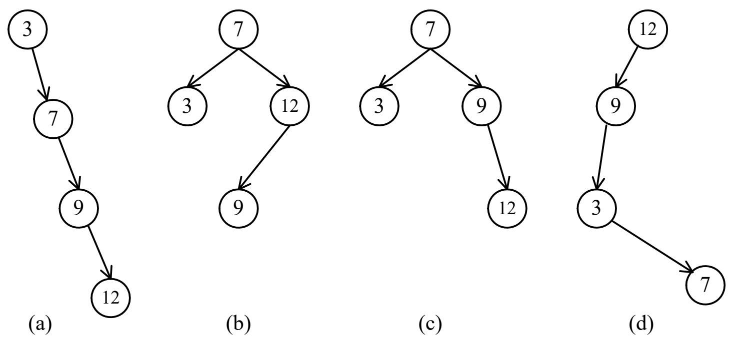
- The longest path represents the optimal solution:

  $x_1=0$, $x_2=1$, $x_3=1$

  $\sum P_i x_i = 20+30 = 50$
- 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;



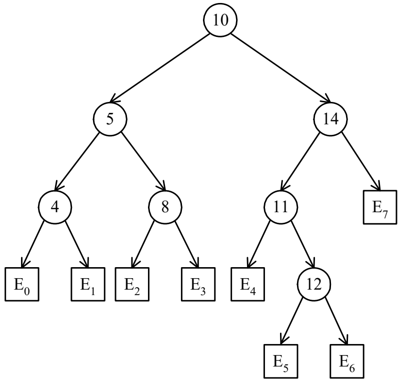
Optimal binary search trees

- 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:

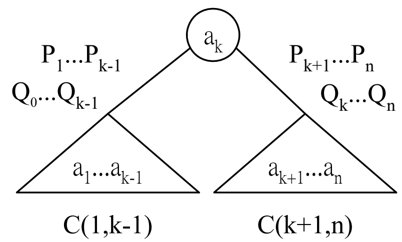
$$\sum_{i=1}^{n} P_i * \text{level}(a_i) + \sum_{i=0}^{n} Q_i * (\text{level}(E_i) - 1)$$

- The level of the root : 1

The dynamic programming approach

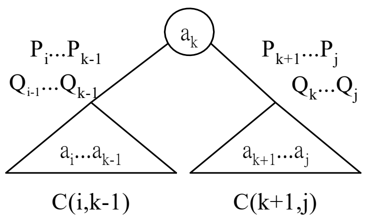
- Let C(i, j) denote the cost of an optimal binary search tree containing $a_i, \ldots, a_j$.
- The cost of the optimal binary search tree with a_k as its root :

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

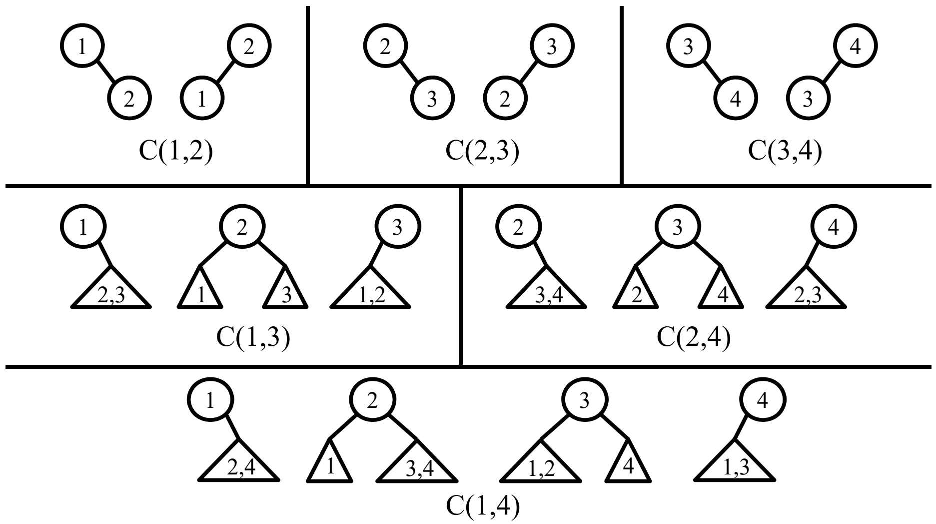


General formula

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

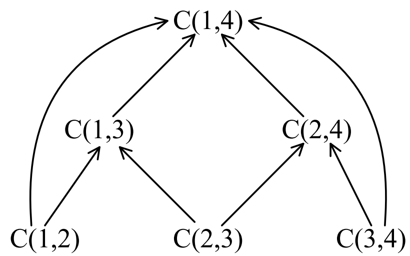


Computation for n=4



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.

$$O\left(\sum_{1\le m\le n} m(n-m)\right) = O(n^3)$$

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

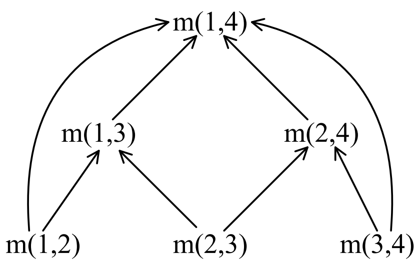
$(A_1 \times (A_2 \times A_3)) \times A_4$, # of scalar multiplications:
3 * 5 * 2 + 5 * 4 * 2 + 3 * 2 * 5 = 100

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

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

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

- Computation sequence :



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

Appendix

Permutations and Combinations

Permutations

- # of m-permutations of n items:

$$^{n}P_{m} = \frac{n!}{(n-m)!}$$

- 3-permutations of {1,2,3,4} in lexicographic order:

1 2 3, 1 2 4, 1 3 2, 1 3 4, 1 4 2, 1 4 3,
2 1 3, 2 1 4, 2 3 1, 2 3 4, 2 4 1, 2 4 3,
3 1 2, 3 1 4, 3 2 1, 3 2 4, 3 4 1, 3 4 2,
4 1 2, 4 1 3,4 2 1, 4 2 3, 4 3 1, 4 3 2.

Generating permutations lexicographically

- m-permutations of {1,2,...,n}:
  first permutation: 1 2 ... m
  last permutation: n n-1 ... n-m+1
- All permutations except the last permutation are updatable (can be used to generate the next permutation).
- The updatable condition for a permutation (P_1P_2 ... P_m):
  $\exists$ j, $\ni P_i < j \le n$ and j is not used at the left of P_i.

The updating algorithm

- Let P_i be the rightmost element and j be the smallest index satisfying the updatable condition.

$P_i \leftarrow j$
$P_{i+1} \leftarrow$ the first position which is not used.
:
:
$P_{i+k} \leftarrow$ the kth position which is not used.
:
:
$P_m \leftarrow$ the (m-i)th position which is not used.

Example of updating

- 5-permutations of {1,2,3,4,5,6,7,8,9}:

:
:
8 1 3 6 2 P_i=2, j=4
8 1 3 6 4 P_i=4, j=5
8 1 3 6 5 P_i=5, j=7
8 1 3 6 7 P_i=7, j=9
8 1 3 6 9 P_i=6, j=7
8 1 3 7 2
:
8 1 3 7 9 P_i=7, j=9
8 1 3 9 2
:
8 1 3 9 7 P_i=3, j=4
8 1 4 2 3
:

Ranking permutations

- Let N={1,2,3,…,n}
- The rank sequence $\{r_1, r_2, ..., r_m\}$ of permutation ($P_1P_2...P_m$):

  r_i is rank of P_i in N-{P_1, P_2,... , P_{i-1}},
  where the smallest rank is 0.
- $r_1r_2...r_m$ can be seen as a mixed radix integer:

  $0 \le r_m \le n\text{-}m$ (n-m+1 digits)
  $0 \le r_{m-1} \le n\text{-}m\text{+}1$ (n-m+2 digits)
  :
  $0 \le r_2 \le n\text{-}2$ (n-1 digits)
  $0 \le r_1 \le n\text{-}1$
- rankp($P_1P_2...P_m$) = $[\sum_{i=1}^{m-1} r_i \prod_{j=0}^{m-i-1}(n-i-j)] + r_m$

Example of ranking

- 3-permutations of {1,2,3,4}

$P_1P_2P_3$	$r_1r_2r_3$	rankp
1 2 3	0 0 0	0
1 2 4	0 0 1	1
1 3 2	0 1 0	2
1 3 4	0 1 1	3
1 4 2	0 2 0	4
1 4 3	0 2 1	5
2 1 3	1 0 0	6
2 1 4	1 0 1	7
2 3 1	1 1 0	8
2 3 4	1 1 1	9
2 4 1	1 2 0	10
2 4 3	1 2 1	11=1x3x2+2x2+1
:		

100
…
149
150
151
…
199
200
201

456=4*10^2+5*10+6

Unranking permutations

- Given d = rankp($P_1P_2...P_m$), how to obtain (P_1 P_2 ... P_m) from d?

$$r_i = \left\lfloor (d - \sum_{j=1}^{i-1} r_j \prod_{k=0}^{m-j-1}(n-j-k)) / \prod_{k=0}^{m-j-1}(n-j-k) \right\rfloor$$

for i =1, 2, ..., m

- d=11 in 3-permutations of {1,2,3,4}

2 | 11   1 --- r_3
3 | 5   2 --- r_2
1 ----- r_1

10 | 456   6
10 | 45   5
4

Unranking permutations

- $P_i = r_i + i - d_i$ where d_i is the smallest nonnegative integer such that

$$d_i = \sum_{j=1}^{i-1} [r_i + i - d_i < P_j], P_i \neq P_j, j < i$$

- d=11
  $P_1 = r_1 + 1 - d_1 = 1+1-0 = 2$
  $P_2 = r_2 + 2 - d_2 = 2+2-d_2 = 4$
  $P_3 = r_3 + 3 - d_3 = 1+3-d_3 = 1+3-1=3$

Combinations

- # of m-combinations of n items:

$$^{n}C_m = \frac{n!}{(n-m)!m!}$$

- 3-combinations of {1,2,3,4} in lexicographic order:
  1 2 3, 1 2 4, 1 3 4, 2 3 4

Generating combinations lexicographically

- m-combinations of {1, 2, ..., n}:
  - first combination:12...m
  - last combination:(n-m+1)(n-m+2)...n
  - The updatable condition for a combination $(c_1c_2...c_m)$:
    $\exists$ j, $\ni 1 \leq j \leq m$, $c_j < n - m + j$

The updating algorithm

- step1: Find the largest j satisfying the updatable condition.
- step2: $c_j \leftarrow c_j + 1$
- step3: $c_{j+1} \leftarrow c_j + 1$, $c_{j+2} \leftarrow c_{j+1} + 1$, ..., $c_m \leftarrow c_{m-1} + 1$
- Example: 3-combinations of {1, 2, 3, 4, 5, 6}
  ⋮
  1 3 4
  1 3 5
  1 3 6
  1 4 5

Ranking combinations

- 3-combinations of {1, 2, 3, 4, 5, 6}

c_1 c_2 c_3	rankc
1 2 3	0
1 2 4	1
1 2 5	2
1 2 6	3
1 3 4	4
1 3 5	5
1 3 6	6
1 4 5	7
1 4 6	8
1 5 6	9

	c_1 c_2 c_3	rankc
	2 3 4	10
$C_1^2=2$	2 3 5	11
	2 3 6	12
$C_2^3=3$	2 4 5	13
	2 4 6	14
	2 5 6	15
$C_3^4=4$	3 4 5	16
	3 4 6	17
	3 5 6	18
	4 5 6	19

$4+3+2=9$

Ranking rules

- 3-combinations of {1, 2, 3, 4, 5, 6}
- # of combinations greater than (2 3 4):
  - Fix nothing, 3-combinations of {3,4,5,6}=4
  - Fix (2), 2-combinations of {4,5,6}=3
  - Fix (2 3), 1-combinations of {5,6}=2
- Rankc(2 3 4)=19-(4+3+2)=10

Unranking combinations

- Given g = rankc(c_1, c_2, ..., c_m), how to obtain (c_1 c_2 ... c_m) from g?
- g=10 in 3-combinations of {1, 2, 3, 4, 5, 6}
  - $(C_3^6-1)-10=19-10=9$
  - $C_3^5=10>9>C_3^4=4 \text{ -----> } (2)$

    $9-4=5$

    $5>C_2^3=3 \text{ -----> } (2\ 3)$

    $5-3=2$

    $C_1^2=2 \text{ -----> } (2\ 3\ 4)$