1 UNIT 1 Introduction (AL Ch.1,2,3)

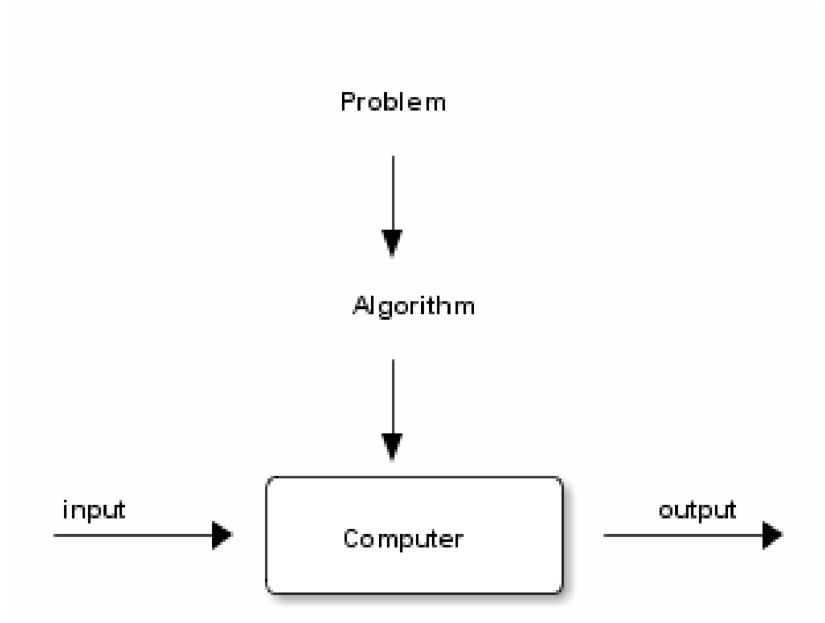
INTRODUCTION: Notion of Algorithm, Review of Asymptotic Notations, Mathematical Analysis of Non-Recursive and Recursive Algorithms Brute Force Approaches: Introduction, Selection Sort and Bubble Sort, Sequential Search and Brute Force String Matching.

2 Notion of Algorithm

2.1 Definition for Algorithm.

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

2.2 Notion of the algorithm



2.3 Characteristics

1. Algorithm has more than one representation.

   For example, gcd(m, n) is described below in the form of steps and pseudocode.

   Euclid's algorithm for computing gcd(m, n)

   (a) If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.
   (b) Divide m by n and assign the value of the remainder to r.
   (c) Assign the value of n to m and the value of r to n. Go to Step 1.

   OR

```
ALGORITHM Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n != 0 do
  r <- m mod n
  m <- n
  n <- r

return m
```

2. More than one algorithm for a given problem

   (a) Find the prime factors of m.
   (b) Find the prime factors of n.
   (c) Identify all the common factors in the two prime expansions found in Step 1 and Step 2.
   (d) Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

   Thus, for the numbers 60 and 24, we get 60 = 2 . 2 . 3 . 5 and 24 = 2 . 2 . 2 . 3. gcd(60, 24) = 2 . 2 . 3 = 12.

3. Each step of the algorihm must be unambiguous

4. Valid range of inputs must be specified

5. Different ideas behind the same algorithm can work differently.

3 Review of Asymptotic Notations

3.1 Efficiency or complexity

Analysis of an algorthms is an ivestigation of algorithm's efficiency or complexity with respect to the resources time and memory for large input size. The input size and the units for measuring efficiency are described below.

- Input size:
  - Length of the lists for searching and sorting algorithms.
  - The degree of polynomial for polynomial evaluations.
  - N or number of elements for matrix multiplications.
  - Number of bytes or words in text processing.
  - Numbers magnitude as number of bits for primality testing.
  - Review. $N = 2^n$, floor(Lg N) + 1 = n. Where n is the number of bits required for to represent N.
- Units
  - Time as seconds but it depends on the compiler or computer used to measure.
  - Number of operations but it is difficult to tackle all possble operations.
  - Number of basic operations as the most time consuming operation.

3.2 Orders of growth

T(n) is
$\approx$ Cost of basic operation x Count of operations
$\approx C_{op}C(n)$

The multiplier constant C_{op} can be dropped when comparing algorithms as it is constanct for both. The count of operations is called the the growth function of the algorithm denoted by f(n). Example of growth functions are : lg n, n, nlgn, n^k, 2^n, n! for input size n.

The rate of growth of the function is small for efficient algorithms. Algorithms that require 2^n and n! operations are exponential. Others are termed logarithmic, linear or polynomial.

The algorithms whose growth functions t(n) and g(n) are compared using the notations O, Θ and Ω. They generally mean the following:

- O means t(n) grows slower than g(n)
- Θ means t(n) grows as fast as g(n)
- Ω means t(n) grows faster than g(n)

3.3 O notation

$t(n) \in O(g(n)$

A function t(n) is said to be in O(g(n)), denoted $t(n) \in O(g(n))$, if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

$t(n) \leq cg(n)$ for all $n \geq n0$.

Example: $100n + 5 \in O(n2)$ when $n_0 = 101$, $c = 5$

3.4 Ω notation

$t(n) \in \Omega(n)$

A function t(n) is said to be in Ω(g(n)), denoted $t(n) \in \Omega(g(n))$, if t(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

$t(n) \geq cg(n)$ for all $n \geq n0$.

Example: $n^3 \in \Omega(n^2)$ for $n_0 = 0$, $c = 1$

3.5 Θ(g(n)) notation

A function t(n) is said to be in Θ(g(n)), denoted $t(n) \in \Theta(g(n))$, if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n0 such that c2g(n) $\leq$ t (n) $\leq$ c1g(n) for all n $\geq$ n0.

Example: 1/2n(n-1) $\in \Theta(n^2)$, $c_1 = 1/2$, c_2=1/4, $n_0 = 2$

3.6 Useful Property

Theorem: If t1(n) $\in$ O(g1(n)) and t2(n) $\in$ O(g2(n)), then t1(n) + t2(n) $\in$ O(max{g1(n), g2(n)}).

It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

Since t1(n) $\in$ O(g1(n)), there exist some positive constant c1 and some nonnegative integer n1 such that

t1(n) $\leq$ c1g1(n) for all n $\geq$ n1.

Similarly, since t2(n) $\in$ O(g2(n)),

t2(n) $\leq$ c2g2(n) for all n $\geq$ n2.

Let us denote c3 = max{c1, c2} and consider n max{n1, n2} so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned}
&t1(n) + t2(n)\\
&\quad \leq c_1\ g1(n) + c_2\ g2(n)\\
&\quad \leq c_3\ g1(n) + c_3\ g2(n)\\
&\quad \leq c_{3[g1}(n) + g2(n)]\\
&\quad \leq c_3\ 2\max\{g1(n), g2(n)\}.
\end{aligned}$$

Hence, t1(n) + t2(n) $\in$ O(max{g1(n), g2(n)}), with the constants c and n0 required by the O definition being 2c3 = 2 max{c1, c2} and max{n1, n2}, respectively.

3.7 Using limits comparing orders of growth

Comparing orders of growth using limits is an alternative to mathematical notations O, Θ and Ω.

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that t(n) is smaller order of growth than g(n)} \\ c & \text{implies that t (n) has the same order of growth as g(n)} \\ \infty & \text{implies that t(n) has a larger order of growth than g(n)} \end{cases}$$

The formual for n! $= \surd\{2\pi\ \mathrm{n}\}(\mathrm{n}/\mathrm{e})^{\mathrm{n}}$ is useful in evaluations.

3.8 Worst-Case, Best-Case and Average-Case efficiency

```
SequentialSearch(A[0..n  1], K)
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n  1] and a search key K
//Output: The index of the first element in A that matches K
// or 1 if there are no matching elements
i <- 0
while i < n and A[i] = K do
  i <- i + 1
if i < n return i
else return 1
```

- Best case is 1
- Worst case search for X in a list of size n is n.
- Average case is the time between the best case and worst case.

  Running time depends on the type of data for given input size.

  $$\begin{aligned} &\mathrm{Ta(n)} \\ &\quad = (\mathrm{X} \in \mathrm{A}).\mathrm{p} + (\mathrm{X} \notin \mathrm{A}).(1\text{-}\mathrm{p}) \\ &\quad = (1 + 2 + \ldots + \mathrm{n})/\mathrm{n.p} + \mathrm{n}.(1\text{-}\mathrm{p}) \\ &\quad = (\mathrm{n}+1)/2.\mathrm{p} + \mathrm{n}(1\text{-}\mathrm{p}) \end{aligned}$$

4 Non recursive algorithms

General plan for analyzing the time efficiency of nonrecursive algorithms

1. Identify input's size.
2. Identify the basic operation.

3. worst, best, average case
4. Set up the sumation
5. Solve the summation

4.1 maximum element in a list

```
MaxElement(A[0..n  1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n  1] of real numbers
//Output: The value of the largest element in A
maxval <- A[0]
for i <- 1 to n  1 do
   if A[i] > maxval
      maxval <- A[i]
return maxval
```

- Input size is n
- Basic peration is comparison
- Worst case
- Σ_1^n
- Solution is O(n)

4.2 Find if the list has unique elements

```
UniqueElements(A[0..n  1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n  1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for i <- 0 to n  2 do
   for j <- i + 1 to n  1 do
     if A[i] = A[j] return false
return true
```

- Input size n
- Basic operation is compare

- $\Sigma_{i=0}^{n-2}\Sigma_{j=i+1}^{n-1}$
- Solution is O(n^2)

T(n)
= 1 + T(n-1)
= 1 + 1 + T(n-2)
= 2 + T(n-2)
...
= i + T(n-i)
= n + T(n-n)
= n or O(n)

4.3 Matrix multiplication C = A X B

```
MatrixMultiplication(A[0..n  1, 0..n  1], B[0..n  1, 0..n  1])
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i <-  to n  1 do
   for j <- 0 to n  1 do
      C[i,j] <- 0.0
      for k <- 0 to n  1 do
         C[i,j] <- C[i,j ] + A[i,k] * B[k,j]
return C
```

- Input size n
- Basic operation is multiplication
- Worst case
- $\Sigma_{i=0}^{n-1}\Sigma_{j=0}^{n-1}\Sigma_{k=0}^{n-1}$
- Solution is O(n^3)

4.4 Binary digits in a number (loop count is lg n)

```
Binary(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n’s binary representation
count <- 1
```

```
while n > 1 do
  count <- count + 1
  n <- n/2
return count
```

- Input size n
- Basic operation is divide
- Σ_1^{lgn}
- Solution is O(lgn)

5 Recursive algorithms

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Identify input's size.
2. Identify the basic operation.
3. worst, best, average case
4. Set up a recurrence relation
5. Solve the recurrence.

5.1 n!

```
F(n)
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n  1)*n
```

- Input size is n
- Basic operation is multiplication
- No need of best and average case analysis
- Recurence relation is T(n) = T(n-1) + 1 and T(0) = 1

- Solution is O(n)

T(n)
= 1 + T(n-1)
= 1 + 1 + T(n-2)
= 2 + T(n-2)
...
= i + T(n-i)
= n + T(n-n)
= n or O(n)

5.2 Tower of Hanoi

```
Hanoi(disk, source, dest, spare):
if disk == 0
   move disk from source to dest
   return;
Hanoi(disk-1, source, spare, dest)
move disk from source to dest
Hanoi(disk-1, spare, dest, source)
```

- Input size n discs
- Basic operation is move from one tower to another
- no worst or best case
- T(n) = 2T(n-1) + 1 and T(1) = 1
- Solution $O(2^n)$

T(n)
= 1 + 2T(n-1)
= 1 + 2(1 + 2T(n-2))
= $1 + 2^1 + 2^2.T(n-2)$
...
= $1 + 2^1 + 2^2 + \ldots + 2^{i-1} + 2^i.T(n-i)$
= $(1 - 2^i)/(1-2) + 2^i.T(n-i)$
= $2^i - 1 + 2^i.T(n-i)$
= $2^i - 1 + 2^i.T(n-i)$
= $2^{n-1} - 1 + 2^{n-1}.T(1)$
= $2^{n-1} - 1 + 2^{n-1}.1$
= $2^n - 1 \in O(2^n)$

5.3 Binary digits

```
BinRec(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
else return BinRec(n/2) + 1
```

- Input size is n
- Basic operation is division in recursive call
- No difference between worst and best case
- $T(n) = T(n/2) + 1$ for $n > 1$, $T(1)=0$
- Solution is $O(\lg n)$

Choose 2^k as n,

$T(2^k)$
$= T(2^{k-1}) + 1$
$= 1 + 1 + T(2^{k-2})$
$= 2 + T(2^{k-2})$
$= i + T(2^{k-i})$
$= k + T(1)$
$= \lg n \in O(\lg n)$

6 Brute Force Approaches

6.1 Introduction

- Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- Applicable to a very wide variety of problems. More difficult to point out problems it cannot tackle.
- For some important problems—e.g., sorting, searching, matrix multiplication, string matching— the brute-force approach yields reasonable algorithms.

- The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
- Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.
- Can serve an important theoretical or educational purpose as a yardstick with which to judge more efficient alternatives for solving a problem.

6.2 Selection Sort

- A0 $\leq$ A1 $\leq$ A2 $\leq \ldots \leq$ Ai-1 | Ai,...,Amin,...An-1
- Select the smallest number and place it at A0
- Select next smallest and place it at A1
- $T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$
- $\Theta(\mathrm{n}^2)$

```
SelectionSort(A[0..n  1])
//Sorts a given array by selection sort
//Input: An array A[0..n  1] of orderable elements
//Output: Array A[0..n  1] sorted in nondecreasing order
for i <- 0 to n  2 do
  min <- i
  for j <- i + 1 to n  1 do
    if A[j] < A[min] min←j
  swap A[i] and A[min]
```

6.3 Bubble Sort

- A0,...,Aj,Aj+1,...,Ani1 | Ani $\leq \ldots \leq$ An1
- Push the largest element to the end
- Push the next largest to end.
- $T(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$
- $\Theta(\mathrm{n}^2)$

```
BubbleSort(A[0..n  1])
//Sorts a given array by bubble sort
//Input: An array A[0..n  1] of orderable elements
//Output: Array A[0..n  1] sorted in nondecreasing order
for i <- 0 to n2 do
  for j <- 0 to n2i do
    if A[j+1] < A[j] swap A[j] and A[j+1]
```

6.4 Sequential Search

- A[n] = K
- A[i] # K is the loop condition
- if i = n after the loop means key not found.
- $T(n) = \sum_{i=0}^{n-1} 1$
- Θ(n)

```
SequentialSearch2(A[0..n], K)
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n  1] whose value is
// equal to K or 1 if no such element is found
A[n] <- K
i <- 0
while A[i] = K do
  i <- i + 1
if i < n return i
else return 1
```

6.5 String matching

- Number positions to slide n - m + 1
- P[0..m] = T[i..i+m-1]
- Θ(n^2)

```
BruteForceStringMatch(T[0..n  1], P[0..m  1])
//Implements brute-force string matching
//Input: An array T [0..n  1] of n characters representing a text and
```

```
// an array P[0..m  1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
// matching substring or 1 if the search is unsuccessful
for i <- 0 to n  m do
  j <- 0
  while j < m and P[j] = T [i+j] do
     j <- j + 1
  if j = m return i
return 1
```

7 UNIT 2 Divide and Conquer (HS Ch.3)

DIVIDE AND CONQUER: Divide and Conquer: General Method, Defective Chess Board, Binary Search, Merge Sort, Quick Sort and its performance.

8 General Plan

Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

Typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size n/2. More generally, an instance of size n can be divided into b instances of size n/b, with 'a' of them needing to be solved. (Here, a and b are constants; a $\geq$ 1 and b $>$ 1.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time T (n):

T (n) = aT(n/b) + f(n)

where f(n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions.

8.1 Master Theorem

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d\,, \\ \Theta(n^d logn) & \text{if } a = b^d\,, \\ \Theta(n^{log_b a} & \text{if } a = b^d\,. \end{cases}$$

For, Merge sort or Quick sort

T(n) = 2. T(n/2) + n

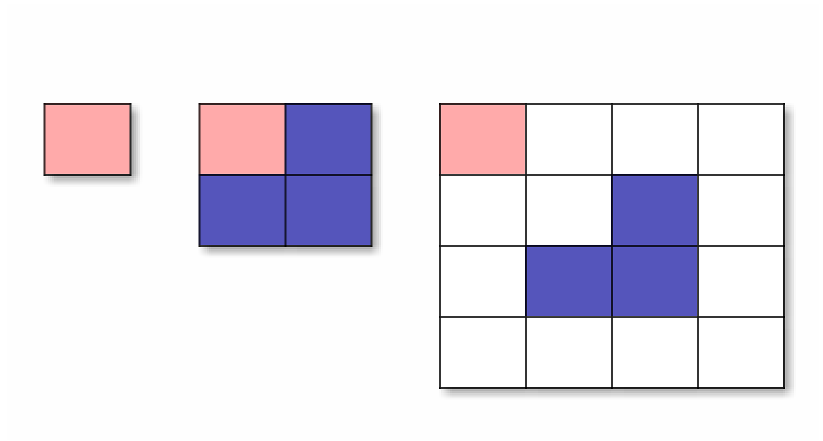
a = b = 2, d = 1, T(n) = Θ(nlgn)

8.2 Substitution Method

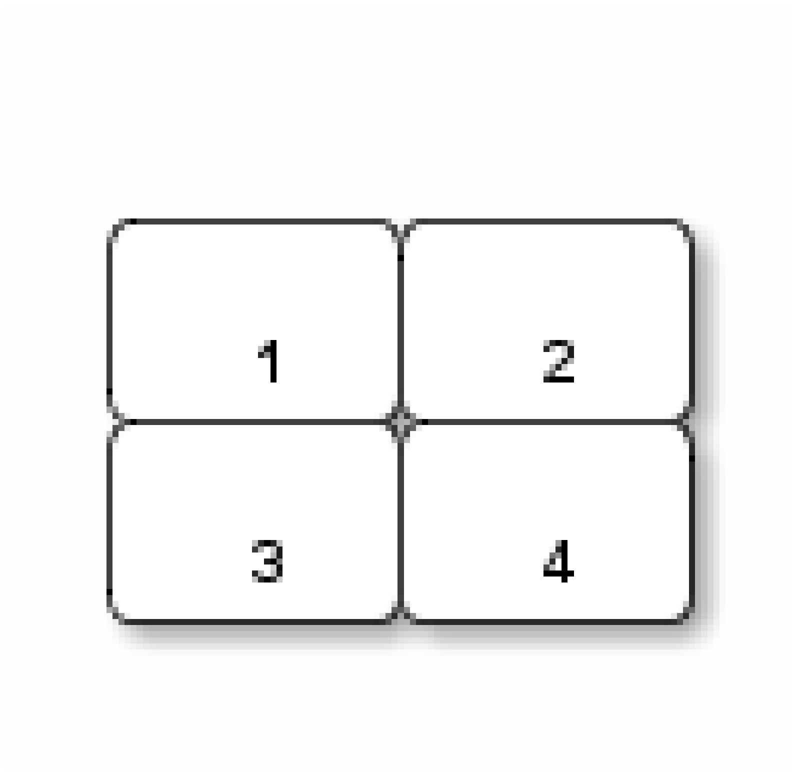
T(n)
= n + 2.T(n/2)
= n + n + 4.T(n/4) + ...
= n + n + n + 8.T(n/8) + ...
= ...
= i.n + 2^i.T(n/2^i)
= nln + n.T(1) for 2^i = n => i lgn
= nlogn

9 Defective Chess Board

9.1 Illustration



9.2 Design



The board corner (tr,tc) and size size is divided into 4 boards is given below along with square for creating the defect.

board	top-left corner	defective square
1	(tr,tc)	(tr+s-1, tc+s-1)
2	(tr, tc+s)	(tr+s-1, tc+s)
3	(tr+s, tc)	(tr+s, tc+s-1)
4	(t+s, tc+s)	(tr+s, tc+s)

9.3 Algorithm

```
ChessBoard(tr, tc, dr, dc, size)
// [tr,tc] identifies the top left corner of the board
// [dr,dc] identifies the defective position
// size is the board size 2^k
if size = 0 return;
s = zize/2
tile <- tile +1

if dr < tr+s && dc < tc+s  // defect in 1
   ChessBoard(tr, tc, dr, dc, s);          //1
   ChessBoard(tr, tc+s, tr+s-1, tc+s, s);  //2
   ChessBoard(tr+s, tc, tr+s, tc+s-1, s);  //3
   ChessBoard(tr+s, tc+s, tr+s, tc+s, s);  //4

// repeat for other three boards
tile < tile-1
Board[dr][dc] = tile
```

9.4 Analysis

T(n) = 4T(n-1) + 1 for n >0 and
T(0) = 1 for n=0

T(n)
= 1 + 4T(n-1)
= 1 + 4[1 + 4T(n-2)]
= 1 + 4 + 4^2 T(n-2)
= 1 + 4 + ... + 4^{i-1} + 4^i T(n-i)
= (1-4^i)/(1-4) + 4^i T(n-i)
= 1/3.(4^i -1) + 4^i T(n-i)
= 1/3.(4^n - 1) + 4^n T(0)
= 4^n(1/3 +1) - 1/3
= 1/3(4^n - 1)
$\in$ O(4^n)

9.5 C Program

```
#include <stdio.h>
#define K 4
int Board[8][8], tile=0;
void ChessBoard(int tr, int tc, int dr, int dc, int size)
{
  if(size==0) return;
  int t=tile++, s=size/2;
  // Use (d) type domino to cover the junction of the chessboards
  if(dr < tr+s && dc < tc+s)  // defect in 1
    {
      ChessBoard(tr, tc, dr, dc, s);          //1
      ChessBoard(tr, tc+s, tr+s-1, tc+s, s);  //2
      ChessBoard(tr+s, tc, tr+s, tc+s-1, s);  //3
      ChessBoard(tr+s, tc+s, tr+s, tc+s, s);  //4
    }

  // Use (c) type domino to cover the junction of the chessboards
  if(dr < tr+s && dc >= tc+s) // defect in 2
    {
      ChessBoard(tr, tc+s, dr, dc, s);       // 2
      ChessBoard(tr, tc, tr+s-1, tc+s-1, s); // 1
      ChessBoard(tr+s, tc, tr+s, tc+s-1, s); // 3
      ChessBoard(tr+s, tc+s, tr+s, tc+s, s); // 4
    }

  // Use (b) type domino to cover the junction of the chessboards
  if(dr >= tr+s && dc < tc+s) // defect in 3
    {
      ChessBoard(tr+s, tc, dr, dc, s);       // 3
      ChessBoard(tr, tc, tr+s-1, tc+s-1, s); // 1
      ChessBoard(tr, tc+s, tr+s-1, tc+s, s); // 2
      ChessBoard(tr+s, tc+s, tr+s, tc+s, s); // 3
    }


  // Use (a) type domino to cover the junction of the chessboards
  if(dr >= tr+s && dc >= tc+s) // defect in 4
    {
      ChessBoard(tr+s, tc+s, dr, dc, s);     // 4
      ChessBoard(tr, tc, tr+s-1, tc+s-1, s); // 1
      ChessBoard(tr, tc+s, tr+s-1, tc+s, s); // 2
      ChessBoard(tr+s, tc, tr+s, tc+s-1, s); // 3
    }
  tile--;
  Board[dr][dc]=tile;
}

print()
{
  int i, j;

  printf("-------------------\n");
  for (i=0; i<K; i++)
```

```
    {
      for (j=0; j<K; j++)
        {
          printf("%2d", Board[i][j]);
        }
      printf("\n");
    }
}

void main()
{
  int i, j;

  ChessBoard(0, 0, 1, 1, K);
  print();
}
```

10 Binary Search

- Problem P = (n, ai,. . . , al, x)
- Is used to find if an element x is present
- If present find j such that $a_j = x$
- If not present set j = 0
- Base case when n = 1. Then search is if ai = x present otherwise absent.
- Pick an index q in the range i to l and compare x with aq.
  - x = aq; P is solved
  - x < aq; search in ai, ai+1, . . . , aq-1
  - x > aq; search in aq+q, . . . , al
- division take $\Theta(1)$
- There is no need for any combining.

10.1 Recursive Search

```
BinSearch(a,i,l,x)
```

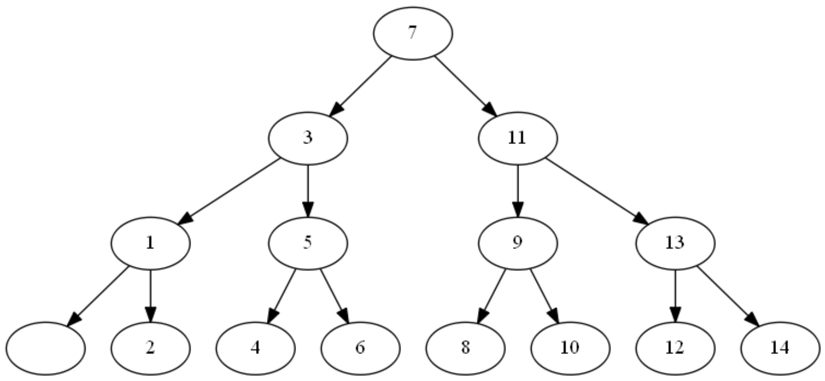
```
// Input : a[i:l] of non decreasing elements, where 1 <= i <= l
// Output: j such that x = a[j] otherwise 0

if i = l
   if x = a[i]
      return i
   else
      return 0

mid = (i+l)/2
if x = a[mid]
   return mid;

if x < a[mid]
   BinSearch(a, i, mid-1, x)
else
   BinSearch(a, mid+1, l, x)
```

-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

i) x = 151, present ii) x -14, not present

low	high	mid	low	high	mid
1	14	7	1	14	7
8	14	11	1	6	3
12	14	13	1	2	1
14	14	14	2	2	2

10.2 Decision tree



10.3 Theorem

If n is in the range $[2^{k-1}, 2^{k}]$ then BinSearch makes at most k element comparisons for a successful search and either k-1 or k comparisons for an unsuccessful search. In otherwords the time for a successful search is O(logn) and for an unsuccessful search is Θ(logn).

Proof: Consider the binary decsion tree describing the action of BinSearch on n elements (and k levels). All successful searches end at a circular node whereas all unsuccessful searches end at a square node.

The number of element comparisons to terminate at a circular node on level i is i. Whereas the number of element comparisons needed to terminate at a square node at level i is only i-1.

Internal nodes are between levels 1 and k and therefores successful search takes atmost k comparisons.

External nodes lies at the level k or k+1. Hence the unsuccessful search takes either k-1 or k comparisons.

10.4 Analysis

The following properties of binary tree are used in the analysis.

1. Level = 1 + height of the tree.
2. Number of internal nodes = 1 + number of external nodes
3. E is the sum of all external distances to external nodes
4. I is the sum of all distances to internal nodes.

$A_s(n)$
= The average number of comparisons in a successful search
= at most k comparison, where k is the level of internal nodes
= 1 + average height of internal nodes
= 1 + I/n

$A_u(n)$
= The average number of comparisons in an unsuccessful search
= k-1 comparison, where k is the level of external nodes

$= 1 +$ average height external nodes - 1
$= E/(n+1)$
$= n\log n/(n+1)$ (height of the tree is logn)
$\propto \log n$

The relationship between E and I is,

$E = I + 2n$

Therefore

$A_s(n)$
$= 1 + I/n$
$= 1 + 1/n(E-2n)$
$= 1 + 1/n[(n+1)A_u(n) - 2n]$
$= (1 + 1/n)A_u(n) - 1$

i.e $A_s(n)$ and $A_u(n)$ are directly related to each other. That means, $A_s(n) \propto$ logn.

$A_s(n) = A_u(n) \propto \log n$

11 Merge Sort

- Divide the list by 2
- Sort(A[0..n/2-1])
- Sort(A[n/2..n-1])
- Merge sorted lists A[0..n/2-1], A[n/2..n-1] into A
- $T(n) = 2T(n/2) + n-1 = \Theta(n\lg n)$ by master theorm
- Is a stable sort algorithm
- Needs additional memory for sorting.
- Not efficien for small n (time is spent in recursions)

11.1 Example

```
        1,2,2,3,4,5,6,6


    2,4,5,6            1,2,3,6


  2,5      4,6       1,3       2,6


5    2    4    6    1    3    2    6
```

11.2 Algorithm

```
//a[low..high] is a global array to be sorted.
//small(P) is true if there is only one element to sort.
//In this case the list is already sorted.

MergerSort(low, mid, high)
if low < high
   mid <- (low + high)/2
   MergeSort(low, mid)
   MergeSort(mid+1, high)
   Merge(low, mid, high

// a[low..high] is a global array containing two sorted sub lists in
// a[low..mid] and a[mid+1..high]. The goal is to merge these two sets
// into a single set residing in a[low..high].
// b[] is an auxilary global array.
Merge(low, mid, high)
i <- low; j <- mid+1; k <- low;
while i <= mid and j <= high
   if a[i] <= a[j]
      b[k] <- a[i]; i <- i+1
   else
      b[k] <- a[j]; j <- j+1
   k <- k+1

while i <= mid
   b[k] = a[i];
   i <- i+1; k <- k+1;
```

```
while j <= high
   b[k] = a[j];
   j <- j+1; k <- k+1;

for k = low to high
   a[k] = b[k]
```

12 Quick Sort

- Partition the list w.r.t A[1]
- A(0), A(1), . . . , A(s-1) all are <= A(s}
- A(s+1), A(s+2), . . . , A(n-1) all are >= A(s)
- A(s) is in its final position
- Sort A(0..s-1)
- Sort A(s+1..r), r is n-1

12.1 Example

Increment i, if A[i] is bigger than P (pivot) Decrement J, if A[j] is smaller than P (pivot)

65	70	75	80	85	60	55	50	45	
p,i									j
65	i							j	
65	45	i					j	70	
65	45	50	i			j	75	70	
65	45	50	55	i	j	80	75	70	
65	45	50	55	60,j	85,i	80	75	70	
60	45	50	55	65	85,i	80	75	70	

12.2 Algorithm

```
Partition(A[l..r])
```

[1]DEFINITION NOT FOUND.

```
// Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//
// Input: Subarray of array A[0..n  1], defined by its left and right
// indices l and r (l<r)
//
// Output: Partition of A[l..r], with the split position returned as
// this function's value

p <- A[l]
i <- l; j <- r + 1
do
  do i <- i + 1 while A[i] < p
  do j <- j  1 while A[j] > p
  swap(A[i], A[j])
while i < j
swap(A[i], A[j ]) //undo last swap when i  j
swap(A[l], A[j ])
return j

Quicksort(A[l..r])

// Sorts a subarray by quicksort
//
// Input: Subarray of array A[0..n  1], defined by its left and right
// indices l and r
//
// Output: Subarray A[l..r] sorted in nondecreasing order

if l < r
   s <- Partition(A[l..r]) //s is a split position
   Quicksort(A[l..s  1])
   Quicksort(A[s + 1..r])
```

12.3 Best case

- Yields two sub-blocks of approximately equal size and the pivot element in the "middle" position
- Takes n+1 data comparisons.
- Recurrence equation

  T(n)

= n + 2T(n/2)
= n + n + ...
= n x h, where h is the height of the tree h=n+1
= n log n
$\in \Theta$(nlogn)

12.4 Worst case

- If the list is already ordered. One of the sublists is empty.
- The length of successive sublists, including the original, are n, n-1, n-2, ...
- The worst case comparisons are n+1 when the list size is n.

T(n)
= n + T(n-1)
= n + n-1 + ... + 3 + 2 + 1
= n(n+2)/2
$\in \Theta(n^2)$

12.5 Average case

T(n) = (n+1) + 1/n $\sum_{k=1}^{n}[T(k-1) + T(n-k)]$, for n >= 2
= 0, otherwise

(n+1) comparisons with the pivot (which includes two extra where ponters cross)

T(n)
= n+1 + 1/n $\sum_{k=1}^{n}[T(k-1) + T(n-k)]$
= n+1 + 2/n \$$\sum_{k=1}^{n}$T(k-1)

n.T(n) = n(n+1) + 2 \$$\sum_{k=1}^{n}$T(k-1)
(n-1).T(n-1) = (n-1)n + 2 \$$\sum_{k=1}^{n-1}$T(k-1)

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

T(n)/(n+1)
= T(n-1)/n + 2/(n+1)
= T(n-2)/(n-1) + 2/n + 2/(n+1)

...
= T(1)/2 + $\sum_{k=3}^{n} 2/(k+1)$

< 2 $\sum_{k=1}^{n} 1/k$
< 2 $\int_{k=1}^{n} 1/x.dx$

T(n)
< 2 (n+1)log n
< c.n log n
$\in$ Θ(nlogn)

12.6 Advantages/Disadvantages

disadvantages

- Worst case O(n^2) is not good
- Not a stable sort

Advantages

- Sorting in place.
- Inner for loop is more efficient than other sorts

13 UNIT 3 Greedy Method (HS Ch.4)

THE GREEDY METHOD: The General Method, Knapsack Problem, Job Sequencing with Deadlines, Minimum-Cost Spanning Trees: Prim's Algorithm, Kruskal's Algorithm; Single Source Shortest Paths.

14 The General Method

- Applicable to optimization problems only
- Obtains an optimal solution by making sequence of choices
- For each step, that seems to be the best choice is chosen
- The choice made must be : Feasible, Locally optimal, Irrecovable

14.1 Advantages

- Greedy algorithm work fast when they work
- Simple and easy to implement

14.2 Disadvantages

- Does not always produce an optimal solution to every problem
- Correctness of greedy algorithm is hard to prove

15 Knapsack Problem

- A thief robbing a store finds n items; the ith item is worth Pi dollars & weighs Wi kilograms, where Pi & Wi are integers.
- He wants to take as many valuable items as possible, but he can carry at most M kilograms in his knapsack (M is integer).
- Instead of making a binary (0-1) choice for each item as in the 0/1 knapsack problem, the thief can take fractions xi of items

The solution to the knapsack problem is to maximise profit $\sum_{1 \le i \le n} p_i x_i$ and limit the weights by $\sum_{1 \le x_i \le n} w_i x_i \le m$.

15.1 Greedy Solution

1. Compute the value per weight Pi/Wi for each item.
2. Take the item with the greatest value per weight as much as possible.
3. If the supply of that item is exhausted and the knapsack still can carry more, take as much as possible of the item with the next greatest value per weight.
4. Repeat until the knapsack is full.

15.2 Example 1

P = [60,100,120]
W = [10, 20, 30]
M = 50
n = 3

Compute P/W ration

P/W = [6, 5, 4]

Ordered (P, W) pairs,
(P,W) = [(60,10), (100, 20), (120, 30)]

0. cap=50, price=0
1. Take (60, 10): cap=40, price=60
2. Take (100,20): cap=20, price=160
3. Take (120x20/30,20/30): cap=0, prices=240

15.3 Knapsack algorithm

```
GreedyKnapsack(oat m, int n)
// p[1..n] and w[1..n] contain the prots and weights
// respectively of the n objects ordered such that
// p[i]/w[i]  p[i+1]/w[i+1]. m is the knapsack
// capacity and x[1..n] is the solution vector.
for i <- 1 to n do
```

```
  x[i] <- 0.0; // initialize x

U <- m;        // Available capacity
for i <- 1 to n do
  if w[i] > U break;
  x[i] := 1.0; // put the whole object in
  U <- U - w[i]

if i \le n
  x[i] <- U/w[i]; // the last object to be put in
```

15.4 Analysis

T(n)
= Sorting of items based on p/w takes + selection of items
= O(nlgn) + n
= O(nlogn)

15.5 Example 2

P=[10,7,12,13,6,20]
W=[3,2,4,3,13,8]
M=15
n=6

Compute P/W ratio
P/W = [3.3, 3.5, 3.0, 4.3, 0.5, 2.5]

Order (P,W) pairs in increasing order of P/W.
(P,W)=[(13,3), (7,2), (10,3), (12,4), (20,8), (6,13)]

0. Cap = 15, Price=0
1. Take (13, 3): cap = 12, price=13
2. Take (7 ,2): cap = 10, price=20
3. Take (10, 3): cap = 7, price=30
4. Take (12, 4): cap = 3, price=42
5. Take (20*3/8 = 7.5, 3/8): cap = 0, prices=49.5

15.6 Example 3

P = [25,24,15]

W = [18,15,10]
M = 20
n = 3

P/W = (1.389, 1.6, 1.5)

Ordered (P, W) = [(24,15), (15, 10), (25, 18)]

0. cap=20, price=0
1. Take (24,15): cap=5, price=24
2. Take (15*1/2,5/10): cap=0, price=31.5

16 Job Sequencing with deadlines.

Job Sequencing is a well known optimization problem.

- N jobs to execute, each of which takes unit time.
- At any time t = 1, 2, 3 . . . we can execute exactly one job.
- Job i earns us a profit pi > 0 iff it is executed no later than time di.

That is, we have N jobs, Profits Pi and deadlines di for each job.

Feasibility of solution:

A set of jobs is feasible if there exists at least one sequence that allows all the jobs to be executed no later than their deadlines.

16.1 Greedy solution

1. Construct the schedule step by step.
2. Add the job with highest value of pi if feasible.

```
GreedyJob(d,J,n)
// J is a set of jobs that can be cmpleted by their deadlines
J <- {1}
for i <- 2 to n
    if jobs J ∪ {i} can be completed by their deadlines
          J <- J ∪ {i}
```

16.2 Example 1

J	1	2	3	4
pi	50	10	15	30
di	2	1	2	1

Jobs ordered by pi

J	1	4	3	2
pi	50	30	15	10
di	2	1	2	1

Solution

i	job,deadline	ts-1	ts-2	ts-3	ts-4	
1	1,2	1				
2	4,1	4	1			
3	3,2					reject
4	2,1					reject

J={1,4}, s=[4,1]; abreviation for time slot - ts

step 1:

- Our job set is {1}, since p1 > pi for all i 1.
- We may run job 1 before its deadline, so our set is feasible.

step 2:

- We will select job 4 and try to schedule it. That is we have job set {1, 4} now.
- By running in the order of 4, 1; we can keep the set feasible. So job 4 is added to our set.

step 3:

- We will select job 3 and try to schedule it. That is we have job set {1, 3, 4} now.

- But there is no way to keep the set {1, 3, 4} feasible. So scheduling job 3 is rejected.

step 4:

- we will select job 2 and try to schedule it. That is we have job set {1, 2, 4} now.
- But there is no way to keep the set {1, 2, 4} feasible. So scheduling job 2 is rejected.
- So as a result; our solution set is {1, 4} and the order is 4, 1.

16.3 Algorithm

```
JS(d, J, n)
// d[1..n] - deadlines of n jobs
// p[1..n] - p[1] >= p[2] ... >= p[n]
// J[1..k] - feasible solution. The k jobs in the array j are in order of increasing
//           deadline.

d[0] <- 0; J[0] <- 0;
J[1] <- 1 // inclde job 1 as feasible
k <- 1 // the number of jobs in the solution

for i <- 2 to n
  r <- k

  //job i can be inserted into J at the appropriate place
  //without pushing some job already in j past its deadline.

  while d[J[r]] > d[i] and d[J[r] \ne r do r <- r-1

  //Accept job i and insert into J at r+1 if deadline of J[r]
  //is smaller and deadline of i is more than r

  if d[J[r] \le d[i] and d[i] > r
    for m = k step -1 to r+1
      j[m+1] <- j[m]
    j[r+1] <-  i;
    k <- k+1
return k, J[1..k]
```

16.4 Analysis

For this algorithm, sorting jobs into order of decreasing profit takes a time in Θ(nlogn).

The worst case occurs, when the procedure turns out to also to sort the jobs by order of decreasing deadline, and when they can all fit into the schedule.

So, when job i is considered, algorithm looks at each of the k = i-1 jobs already in the schedule. That is; there are,

n-1 k trips round the while loop k=1

and

n-1 m trips round the inner for loop. m=1

So the algorithm takes time in $\Omega(n^2)$.

17 Prim's algorithm for MST

A spanning tree for an undirected graph G=(V,E) is a subgraph of G that is a tree and contains all the vertices of G. The MST is a spanning tree, where the total cost of all the edges in the tree is minimised.

Prim's algorithm work good for dense graphs and kruskal's for sparse graphs.

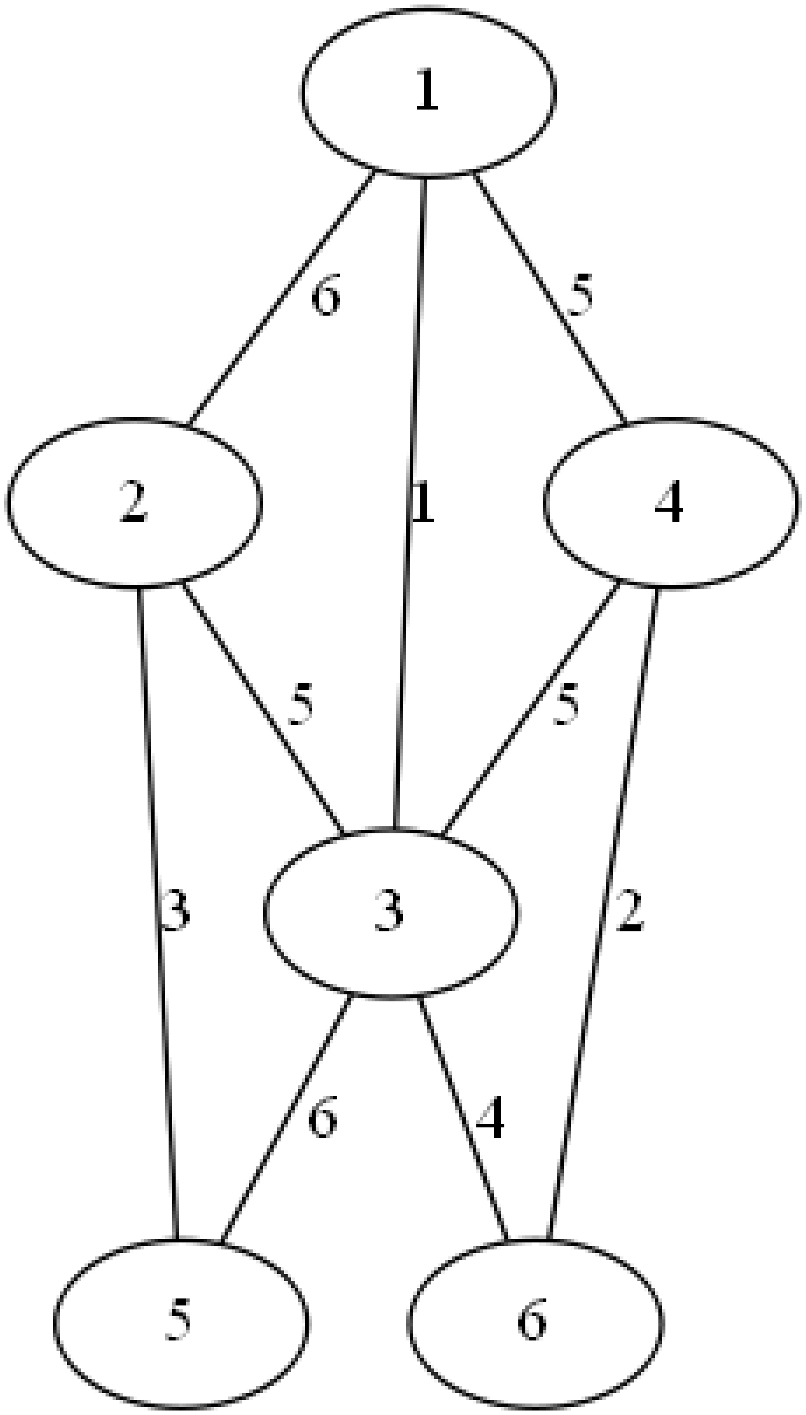
It works as follows:

- Starts from an arbitrary "root": U = {a}
- At each step:
  - Find a light edge spanning (U, V - U)
  - Add this edge to T
- Repeat until the tree spans all vertices

17.1 Example

Use two arrays, closest and lowcost.

- For i $\in$ V-U, closest[i] gives the vertex in U that is closest to i
- For i $\in$ V-U, lowcost[i] gives the cost of the edge (i, closest(i))


1
6
1
5
2
4
5
5
3
3
2
6
4
5
6

U	V-U	low cost edge from U	T
{1}	{2,3,4,5,6}	(1,3).1	(1,3)
{1,3}	{2,4,5,6}	(1,4).5, (3,6):4	(3,6)
{1,3,6}	{2,4,5}	(1,4}.4, (3,2):5, (3,4):5, (6,4):2	(6,4)
{1,3,4,6}	{2,5}	(1,2).6, (3,2):5	(3,2)
{1,2,3,4,6}	{5}	(5,3).3	(5,3)
{1,2,3,4,5,6}	{}		

(u,v).d notation means edge (u,v) with weight d.

- Step 1

  U = {1} V-U = {2,3,4,5,6}

V-U	U	lowcost
2	1	6
3	1	1
4	1	5
5	1	∞
6	1	∞

  Select vertex 3 to include in U 1 -> 3

- Step 2

  U = {1,3}, V-U = {2,4,5,6}

V-U	U	lowcost
2	3	5
4	1	5
5	3	6
6	3	4

  Now select vertex 6, 1 -> 3, 6 -> 3

- Step 3

  U = {1,3,6}, V-U = {2,4,5}

V-U	U	lowcost
2	3	5
4	6	2
5	3	6

  Now select vertex 4, 1 -> 3, 6 -> 3, 4 -> 6

- Step 4

  U = {1,3,6,4}, V-U = {2,5}

V-U	U	lowcost
2	3	5
5	3	6

Now select vertex 2, 1 -> 3, 6 -> 3, 4 -> 2, 2 -> 3

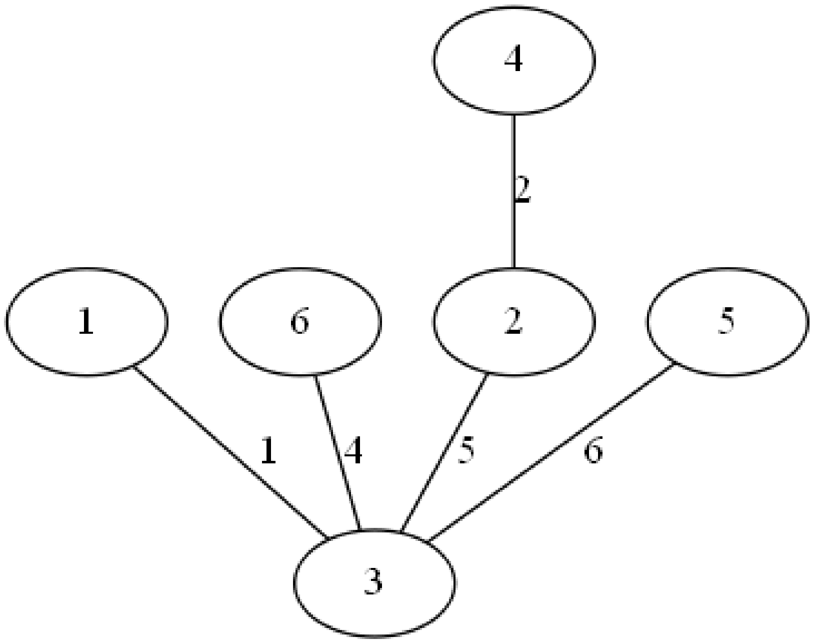
- Step 5

  U = {1,2,3,6,4}, V-U = {5}

V-U	U	lowcost
5	3	6

Now select vertex 5, 1 -> 3, 6 -> 3, 4 -> 2, 2 -> 3, 5 -> 3

- MST



17.2 Algorithm

Prim(G)
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = (V, E)

```
//Output: T , the set of edges composing a minimum spanning tree
   of G
U <- {1} //the set of tree vertices can be initialized with any vertex
T <-
for i <- 1 to |V|  1 do
    find a minimum-weight edge e* = (v*-u*) among all the edges
   (v, u)
    such that v is in V and u is in U
    U <- U ∪ {u*}
    T <- T ∪ {e*}
return T
```

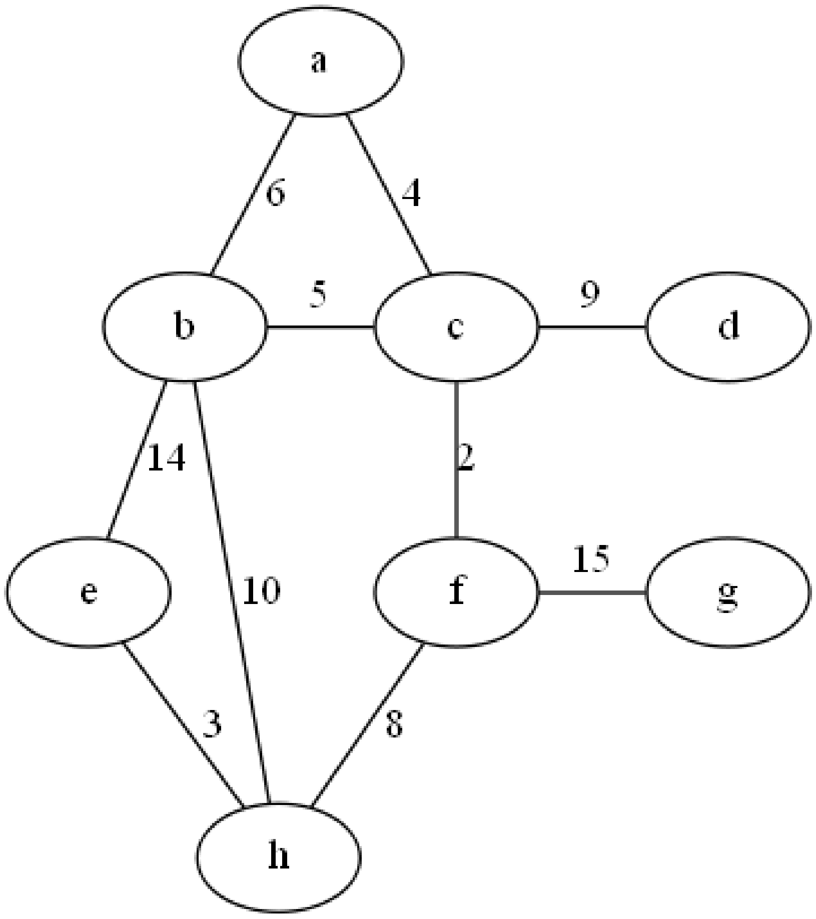
17.3 Analysis

T(n) = O(VlgV)

18 Kruskal's algorithm for MST

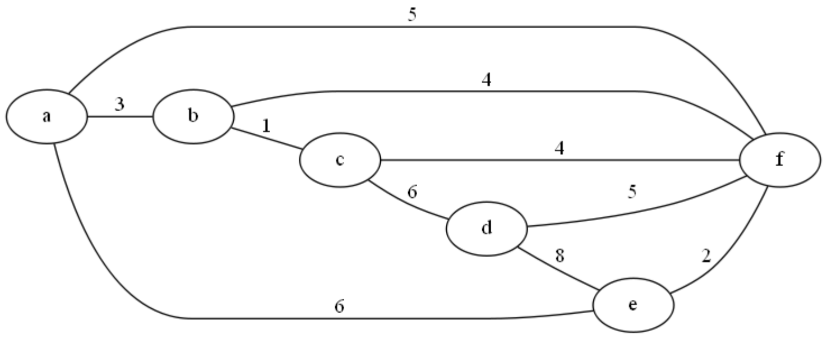
- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the light edge that connects them.
- Continue until the forest 'merge to' a single tree

18.1 Example



edge	components	remarks
{}	{a} {b} {c} {d} {e} {f} {g} {h}	
(c,f)	{a} {b} {c f} {d} {e} {g} {h}	
(e,h)	{a} {b} {c f} {d} {e h} {g}	
(a,c)	{b} {a c f} {d} {e h} {g}	
(b,c)	{a b c f} {d} {e h} {g}	
(b,a)		don't add
(h,f)	{a b c f e h} {d}{g}	
(c,d)	{a b c d e f h} {g}	
(b,h)		don't add
(b,e)		don't add
(f,g)	{a b c d e f g h}	

18.2 Example 2 AL book



edge	components	remarks
{}	{a} {b} {c} {d} {e} {f}	
(b,c)	{a} {b,c} {d} {e} {f}	
(e,f)	{a} {b,c} {d} {e,f}	
(a,b)	{a,b,c} {d} {e,f}	
(b,f)	{a,b,c,e,f} {d}	
(f,d)	{a,b,c,d,e,f}	

Minimum cost = 15

18.3 Disjoint set data structure

1. Make-Set
   - Make-set(a) = {a}
2. Find-Set
   - S = {r, s, t, u}
   - FIND-SET(u) = r
   - FIND-SET(s) = r
3. Union
   - S_u = {r, s, t, u}, S_v = {v, x, y}
   - UNION (u, v) = {r, s, t, u, v, x, y}

18.4 Algorithm

```
krusjal(G,w)
/ Graph G(V,E) with weights w
/ T set of edges in the tree
T <-
for each vertex v $\in$ V
        Make-Set(v)
sort edges of E by increasing weight
for each edge (u,v) $\in$ E in order
    if Find-Set(u) $\neq$ Find-Set(v)
        T <- T $\cup$ {(u,v)}
        Union(u,v)
return T
```

18.5 Analysis

T(n) = O(ElogV)

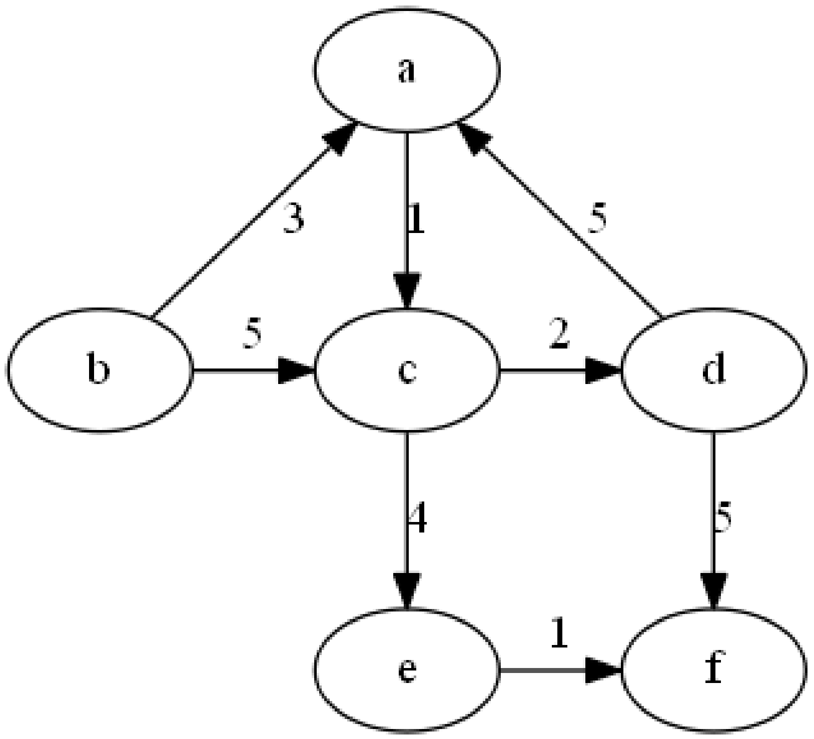
19 Single source shortest paths

- Graphs are digraphs to allow for one way paths.
- Path length is sum of the weights of the edges on the path.
- Source vertex is the starting vertex of the path
- Destination is the last vertex on the path.

The problem is to find the shortest paths from source vertex, v0, to all other vertices.

1. Find vertex u $\in$ V-S such that u.d is minimum
2. Add this vertex, u into S.
3. Recompute v.d for all vertices in V-S

19.1 Example



a	b	c	d	e	f	finalized
∞,-	0,-	∞,-	∞,-	∞,-	∞,-	
3,b	0,-	5,b	∞,-	∞,-	∞,-	b
3,b	0,-	4,a	∞,-	∞,-	∞,-	a
3,b	0,-	4,a	6,c	8,c	∞	c
3,b	0,-	4,a	6,c	8,c	11,d	d
3,b	0,-	4,a	6,c	8,c	9,e	e

19.2 Algorithm

```
ShortestPaths(G, w)
S <- {s}; s.d <- 0

for all v $\in$ V - S
        v.d <- w(s, v)
        v.$\pi$ <- null //predecessor
```

```
while S $\neq$ V do
    chose u from VS such that u.d is minimum
    S <- S $\cup$ {u}
    for all v in V-S
        v.d <- minimum(v.d, u.d + w(u,v))
        v.$\pi$ <- u
```

19.3 Analysis

T(n) = O(E+VlogV)

20 UNIT 4 Dynamic Programming (HS Ch.5)

DYNAMIC PROGRAMMING: The General Method, Warshall's Algorithm, Floyd's Algorithm for the All-Pairs Shortest Paths Problem, Single-Source Shortest Paths: General Weights, 0/1 Knapsack, The Traveling Salesperson problem.

21 The general method

- n!

  n*(n-1)!

- F(n) Fibonacci numbers (Recursive version)

  ```
  F[0] = 0
  F[1] = 1
  F(n) = F(n-1) + F(n-2)

  T(n) = T(n-1) + T(n-2) + 1
       = O(2^n)
  ```

- F(n) Fibonacci numbers (Dynamic version)

  ```
  F[0] <- 0
  F[1] <- 1
  for i <- 2 to n
    F[i] <- F[i-1] + F[i-2]
  return F[n]

  T(n) = O(n)
  ```

- Properties

  1. Optimal substructure

     An optimal solution to a problem (instance) contains optimal solution to subproblems.

  2. Overlapping subproblems

     A recursive solution contains a "small" number of distinct subproblems repeated many times.

  3. Memo(r)ization

     After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

  4. Bottom Up execution

22 Warshall's Algorithm

The transitive closure of a directed graph with n vertices can be defined as the n × n boolean matrix T={t_{ij}}, where t_{ij} = 1 if there is path between i and j.

Warshalls algorithm constructs the transitive closure through a series of boolean matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$. Where $R^{(k)}$ is computed from $R^{(k-1)}$

$r_{ij} = r_{ij} \text{ v } r_{ik} \text{ ˆ } r_{kj}$

if $r_{ik} = 0$, $r_{ij} = r_{ij} \text{ v } 0 \text{ ˆ } r_{kj} = \mathbf{r_{ij}}$

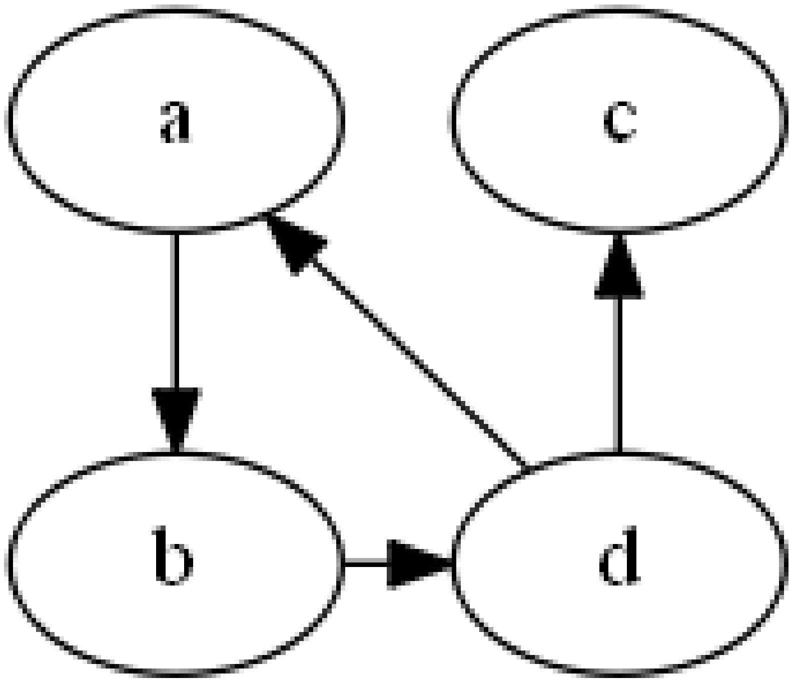
if $r_{ik} = 1$, $r_{ij} = r_{ij} \text{ v } 1 \text{ ˆ } r_{kj} = \mathbf{r_{ij} \text{ v } r_{kj}}$

That is add k-th row to other rows where r_{ik}=1

```
Warshall(A[1..n, 1..n])
//ImplementsWarshall's algorithm for computing the transitive clo-
   sure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R(0) <- A
for k <- 1 to n do
    for i <- 1 to n do
        for j <- 1 to n do
            $R^{(k)}$[i,j] <- $R^{(k1)}$[i,j] or ($R^{(k1)}$[i,k] and $R^{(k1)}$[k,j])
return R(n)
```

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

22.1 Example 1



R0	a b c d
a	0 1 0 0
b	0 0 0 1
c	0 0 0 0
d	1 0 1 0

k=1, add row 1 to row d because r_{ik}=1

R1	a b c d
a	0 1 0 0
b	0 0 0 1
c	0 0 0 0
d	1 1 1 0

k=2, add row 2 to row a and d because r_{ik}=1

R2	a b c d
a	0 1 0 1
b	0 0 0 1
c	0 0 0 0
d	1 1 1 1

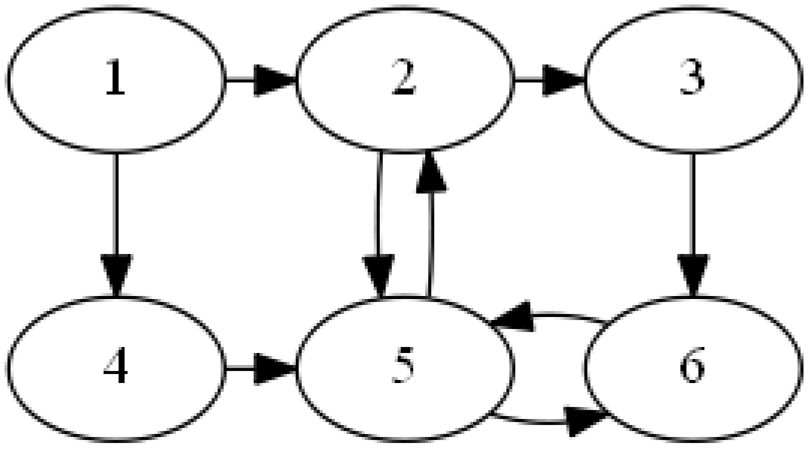
k=3, add row 3 to row d because r_{ik}=1

R3	a b c d
a	0 1 0 1
b	0 0 0 1
c	0 0 0 0
d	1 1 1 1

k=4, add row 4 to row a,b,d because r_{ik}=1

R4	a b c d
a	1 1 1 1
b	1 1 1 1
c	0 0 0 0
d	1 1 1 1

22.2 Example 2



R0	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	1	0	1	0
3	0	0	0	0	0	1
4	0	0	0	0	1	0
5	0	1	0	0	0	1
6	0	0	0	0	1	0

r6	1	2	3	4	5	6
1	0	1	1	1	1	1
2	0	1	1	0	1	1
3	0	1	1	0	1	1
4	0	1	1	0	1	1
5	0	1	1	0	1	1
6	0	1	1	0	1	1

1. R1 = R0 because none of the rows in column 1 has 1
2. R2 we add row 2 to rows 1, 5
3. R3 we add row 3 to rows 1, 2, 5
4. R4 we add row 4 to row 1 = R3
5. R5 we add row 5 to rows 1, 2, 4, 5, 6
6. R6 we add row 6 to every row

23 Floyd's algorithm for the all-pairs SP problems

Consider intermediate vertices of a path:

```
i --> O --> O --> ... --> O --> j
      1     2             k-1
```

d_{ij}(k-1) is the length of the shortest path from i to j whose intermediate vertices are 1,2,..,k-1

Can we extend k-1 to k? In otherwords, we want d_{ij}(k). Can we use d_{ij}(k-1)?

```
                             k

i --> O --> O --> ... --> O --> j
      1     2             k-1
```

Two possibilities:

1. Going through the vertex k does not help - the path through vertices 1..k-1 is still the shortest.
2. There is a shorter path consisting of two subpaths, one from i to k and from k to j. Each subpath passes only through vertices 1..k-1

Therefore,

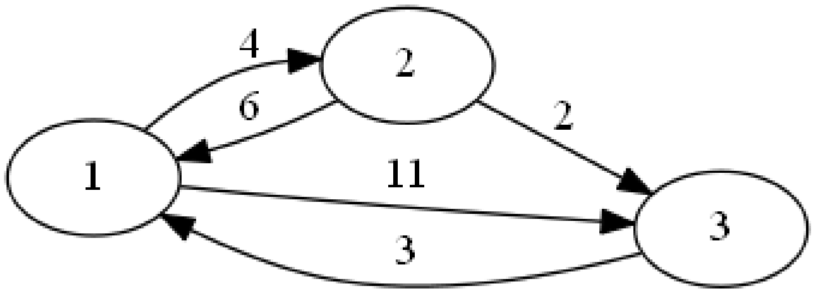
If i to j is the shortest path having intermediate vertices in {1,2,. . . ,k} then i to k and k to j are shortest paths havng intermediate vertices in {1,2,..,k-1}.

$$d_{ij}(k) = \begin{cases} w_{ij} \\ \text{, k=0 or no intermediate vertices} \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) \\ \text{, k} \geq 1 \end{cases}$$

That means solving by the stages $d_{ij}(0)$, $d_{ij}(1)$, $d_{ij}(2)$, . . . , $d_{ij}(n)$.

Compute and store solutions to sub-problems. Combine those solutions to solve larger sub-problems.

23.1 Example 1



A0	1	2	3	A1	1	2	3	A2	1	2	3	A3	1	2	3
1	0	4	11	1	0	4	11	1	0	4	6	1	0	4	6
2	6	0	2	2	6	0	2	2	6	0	2	2	5	0	2
3	3	-	0	3	3	7	0	3	3	7	0	3	3	7	0

23.2 Example 2

A0	1 2 3
1	0 8 5
2	3 0 -
3	- 2 0

A1	1 2 3
1	0 8 5
2	3 0 8
3	- 2 0

A2	1 2 3
1	0 8 5
2	3 0 8
3	5 2 0

A3	1 2 3
1	0 7 5
2	3 0 8
3	5 2 0

23.3 Algorithm

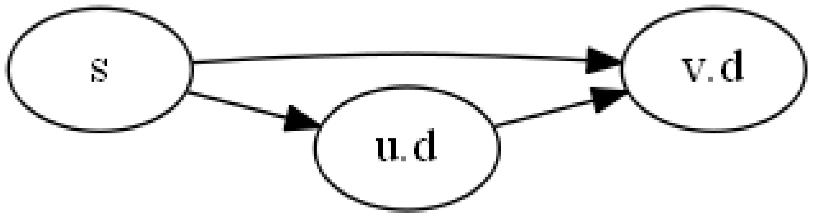
```
AllPaths(Cost, A, n)
/ Cost[0..n-1, 0..n-1] is the cost adjacency matrix of a graph
/ with n vertices. A[i,j] is the cost of a shortest path from
// vertex i to j

for i <- 0 to n-1 do
    for j <- 0 to n-1 do
        A[i,j] <- cost[i,j] // copy cost into A

for k <- 1 to n do
    for i <- 0 to n-1 do
        for j <- 0 to n-1 do
            A[i,j] <- min(A[i,j], A[i,k] + A[k,j]);
```

T(n) = O(n^3)

24 Bellman-Ford algorithm

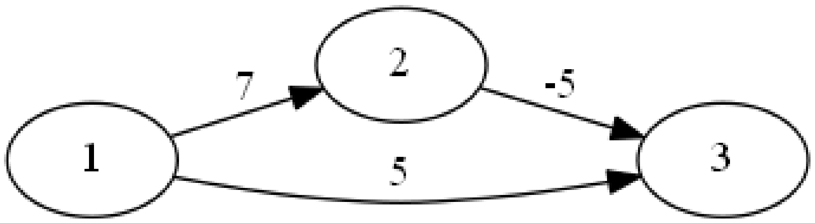
24.1 Idea

Bellman-ord algorithm is a single source shortest path with general weights. Shortest path between s and v is denoted by δ(s,v).



v.d = min(v.d, u.d + w(u,v) for all u ∈ V)

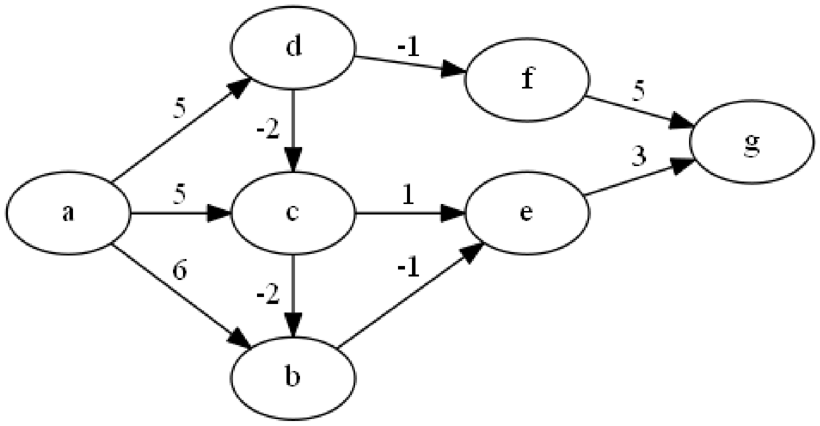
24.2 Example 1



24.3 Algorithm

```
BELLMAN-FORD(G, w, s)

// Initialization
for each vertex v ∈ G.V
        v.d = $\infty$
s.d = 0

// Relaxation
for i = 1 to |V|-1
        for each edge(u,v) ∈ E
                if v.d > u.d + w(u,v)
                        v.d = u.d + w(u,v)
```

24.4 Example 2



E = {(a,b).5,(a,c).5,(a,d).6,(b,e).-1,(c,b).-2,(c,e).1,(d,c).-2,(d,f).-1,(e,g).3,(f,g).5}

v	a	b	c	d	e	f	g
δ(s,v)	0	1	3	5	0	4	3

25 0/1 knapsack

- Number of items is n.
- Knapsack capacity is m.

After considering i items,

- Value of knapsack is denoted by f(i,j)
- j is the remainig knapsack capacity.

The solution in terms of i-1 items is given by the following recurrence.

$$f(i,j) = \begin{cases} 0 \\ \text{for i=0 or m=0} \\ f(i-1,j) \\ \text{if wi} > \text{m} \\ max(f(i-1,j), f(i-1,j-wi)+v_i \end{cases}$$

1. m = 0 (Knapsack capacity is 0) or n = 0 (There are no items)
2. Exclude item i because its weight exceeds the remaining knapsack capacity.
3. Otherwise, maximize the value by excluding or including item i.

25.1 Algorithm

```
dpknapsack(n, m, W[0..n], V[0..n], f[0..n, 0..m])
//Input n items, capacity m, weights w and values v of n items.
//Input  Output is the table f[i,j] to store
//       value of items included when the capacity is j.

for j = 0 to w do
    f(0, j) <- 0

for i = 0 to n do
    f(i, 0) <- 0

for i = 0 to n do
  for j = 0 to m
    if j < w[i] then
       f[i,j] <- f[i-1,j]
    else
       f[i,j] <- maximum {f[i-1,j], v[i] + f[i-1,j-w[i]]}

return f[n, m]
```

25.2 Example

N=4, M=5

i	Wi	Pi
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	10	22	22	22
3	0	10	10	22	22	22
4	0	10	15	25	37	37

26 UNIT 5 (AL Ch.5)

DECREASE-AND-CONQUER APPROACHES, SPACE-TIME TRADEOFFS: Decrease-and-Conquer Approaches: Introduction, Insertion Sort, Depth First Search and Breadth First Search, Topological Sorting Space-Time Tradeoffs: Introduction, Sorting by Counting, Input Enhancement in String Matching.

27 Decrease-and-Conquer approaches

Compute solution to the problem of size n in terms of the solution to the problem of smaller size.

- Decrease by 1
  $a^n = a^{(n-1)}.a$
- Decrease by constant
  $a^n = (a^{n/2})^2$ if n is even otherwie $(a^{(n-1)/2})^2.a$
- Decrease by variable size
  gcd(m,n) = gcd(n, m mod n)

28 Insertion Sort

Iteration of insertion sort: A[i] is inserted in its proper position among the preceding elements previously sorted.

$A0 \le \ldots \le A[j] < A[j+1] \le \ldots \le A[i-1] \mid A[i] \ldots A[n-1]$

28.1 Example

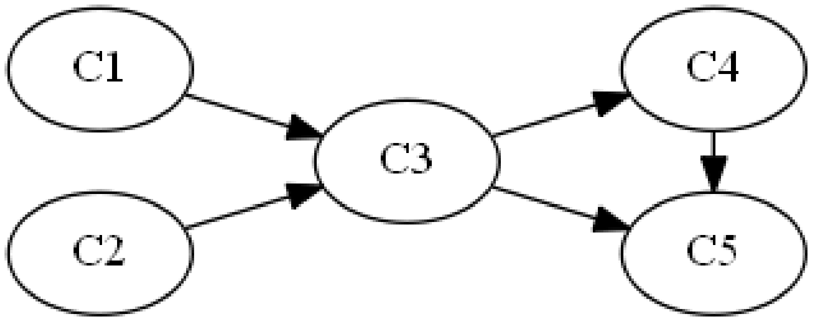
```
89 <- 45 68 90 29 34 17
45 89 <- 68 90 29 34 17
45 68 89 <- 90 29 34 17
45 68 89 90 <- 29 34 17
29 34 45 68 89 <- 34 17
29 34 45 68 89 90 <- 17
17 29 34 45 68 89 90 <-
```

28.2 Algorithm

```
InsertionSort(A[0..n  1])
//Sorts a given array by insertion sort
//Input: An array A[0..n  1] of n orderable elements
//Output: Array A[0..n  1] sorted in nondecreasing order
for i <- 1 to n  1 do
    v <- A[i]
    j <- i  1

    while j $\geq$ 0 and A[j] > v do
        A[j+1] <- A[j] // move to the right
        j <- j1
    A[j+1] <- v
```

T(n)
$= \Sigma_{i=1}^{n-1}\Sigma_{j=0}^{i-1}$
$= \Sigma_{i=1}^{n-1} i$
$= \frac{(n-1).n}{2}$
$=$ O(n^2)

29 Topological Sorting



The solution obtained is C1, C2, C3, C4, C5

- Applicable for Directed Acyclic Graph (DAG)
- Courses modelled as nodes
- Pre-requisites as edges

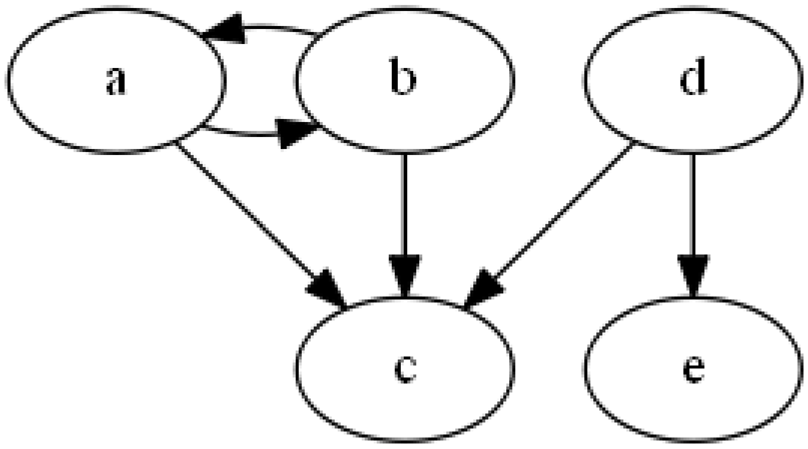
List the vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

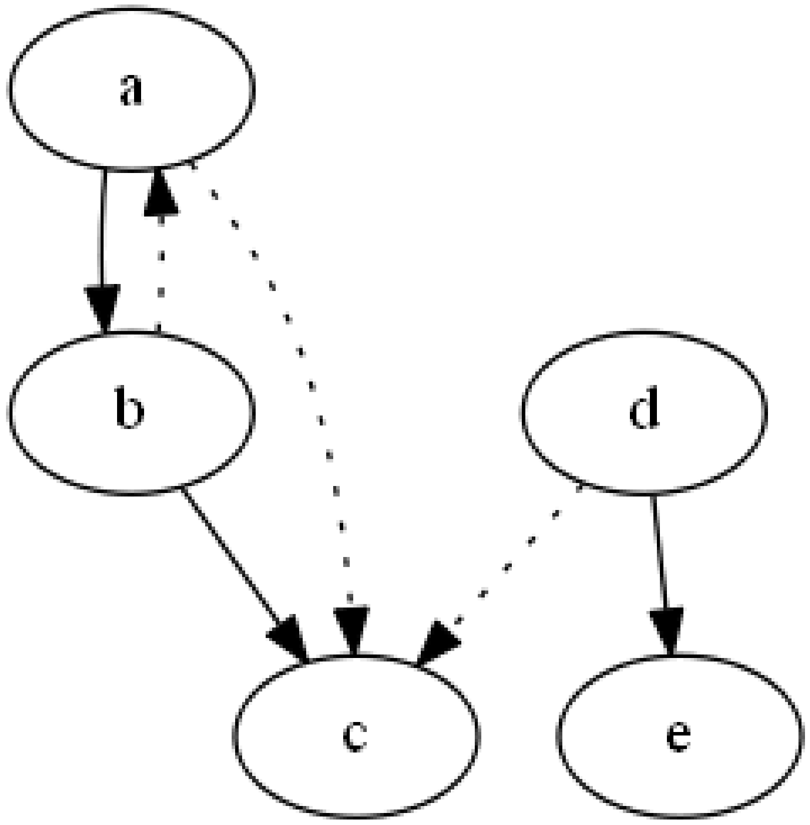
Source Removal method:

Repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.

The order in which the vertices are deleted yields a solution to the topological sorting problem.

30 DFS and BFS





- DFS and BFS are principal traversal algorithms for traversing digraphs.
- DFS forest will have,
  - Tree edges (ab, bc, de)
  - Back edges from vertices to their ancestors (ba)
  - Forward edges from vertices to descendants (ac)
  - Cross edges other than the above (dc)
- The presence of a back edge indicates that the digraph has a directed cycle.
- The absence of a back edge indicates that the digraph is a DAG

30.1 BFS Algorthm

```
BFS(G)
```

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G(V, E)
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are visited by the BFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”
count <- 0
for each vertex v in V do
  if v is marked with 0
     bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable count

count <- count + 1; mark v with count and initialize a queue with v
while the queue is not empty do
   for each vertex w in V adjacent to the front vertex do
      if w is marked with 0
         count <- count + 1; mark w with count
         add w to the queue
   remove the front vertex from the queue
```

30.2 DFS Algorithm

```
ALGORITHM DFS(G)
//Implements a depth-first search traversal of a given graph
//Input: Graph G(V,E)
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are first encountered by the DFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”
count <- 0
for each vertex v in V do
   if v is marked with 0
      dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count
count <- count + 1; mark v with count
for each vertex w in V adjacent to v do
```

```
   if w is marked with 0
      dfs(w)
```

31 Space Time tradeoffs

Often, we can solve a problem faster by using additional space.

- Input enhancement

  is based on preprocessing the instance to obtain additional information that can be used to solve the instance in less time.

- Prestructuring

  is based on using extra space to facilitate faster and/or more exible access to the data.

- Dynamic programming

  is based on recording solutions to smaller instances so that each smaller instance is solved only once.

31.1 Sorting by counting

31.1.1 Comparison counting sorting

- Count, for each element 'i' of a list to be sorted, the total number of elements smaller than this element
- These numbers will indicate the positions of the elements 'i' in the sorted list

```
Array        A = 62,31,84,96,19,47
Counts       C = 3, 1, 4, 5, 0, 2
Sorted Array S = 10,31,47,62,84,96
```

Actual steps are:

```
Array        A = 62 31 84 96 19 47
Counts       C =  0  0  0  0  0  0
Counts       C =  3  0  1  1  0  0 62
Counts       C =  3  1  2  2  0  1 31
Counts       C =  3  1  4  3  0  1 84
Counts       C =  3  1  4  5  0  1 96
Counts       C =  3  1  4  5  0  2 19
Sorted Array S = 10,31,47,62,84,96
```

```
ComparisonCountingSort(A[0..n1])
//Sorts an array by comparison counting
//Input: An array A[0..n  1] of orderable elements
//Output: Array S[0..n  1] of A's elements sorted in nondecreasing order

for i <- 0 to n  1 do
  Count[i] <- 0

for i <- 0 to n  2 do
  for j <- i + 1 to n  1 do
    if A[i] < A[j]
       Count[j] <- Count[j]+1
    else
       Count[i] <- Count[i]+1

for i <- 0 to n  1 do
  S[Count[i]] <- A[i]

return S
```

T(n)
$= \Sigma_{i=0}^{n-1} \Sigma_{j=i+1}^{n-1}$
$= \Sigma_{i=0}^{n-1} (n-1-i)$
= n.(n-1)/2
= O(n^2)

31.1.2 Distribution counting sorting

The idea to count how many times each integer appears in an array. This information can be used to sort the array.

For example, if we knew there were three 0s (and 0 is the minimum), then the sorted array will have 0s in positions 0, 1, and 2.

This requires an additional array to store the counts.

```
Array        A=13,11,12,13,12,12
Values       V=11,12,13 (between l=11 and u=13)
Frequency    F=1, 3, 2
Distribution D=1, 4, 6
Sorted Array S=11,12,12,12,13,13 (j=12-11=1, D[1]-1 = 4-1 = 3, S[3]=12,...]
```

```
DistributionCountingSort(A[0..n  1], m)
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n  1] of integers between l and u (l  u)
//Output: Array S[0..n  1] of A's elements sorted in nondecreasing order
for j <- 0 to u  l do
  D[j]<-0 //initialize frequencies

for i <- 0 to u  l do
  D[A[i]l] <- D[A[i]l]+ 1 //compute frequencies

for j <- 1 to u  l do
  D[j] <- D[j1]+ D[j] //reuse for distribution

for i <- n1 downto 0 do
  j <- A[i]l       //ell
  S[D[j]1] <- A[i]
  D[j] <- D[j]  1

return S
```

T(n) = O(n), better than Quicksort but at the cost of space.

32 String Matching

- Text T[1..n]
- Pattern P[1..m]
- T[s+1..s+m] = P[1..m]

Horspool's algorithm

Matches P[m..1] with Text T[s+m..s+1] i.e from right to left. On mismatch the pattern is shifted to the right by 's' based on the first character in text.

1. P does not have 'c', shift over the pattern if 'c' is not in P

```
T = ...            S          ...
P       B A R B E R
                      B A R B E R
```

2. P has more than one 'c' in P, align the first occurence of 'c' in P

```
T = ...            B          ...
P       B A R B E R
            B A R B E R
```

3. P has 'c' as the last character in P, shift over P

```
T = ...        M E R           ...
P       L E A D E R
                      L E A D E R
```

4. P has 'c' as the last character and more in p, align the first occurence of 'c' in P

```
T = ...            A R           ...
P       R E O R D E R
              R E O R D E R
```

32.1 Shift table

```
ShiftTable(P [0..m  1])

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m  1] and an alphabet of possible characters
//Output: Table[0..size  1] indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)

for i <- 0 to size  1 do
  Table[i] <- m

for j <- 0 to m-2 do
  Table[P[j ]] <- m  1 j
```

i =	0	1	2	3	4	5
P =	B	A	R	B	E	R
j	0	1	2	3	4	-

$m = 6$

char	B	A	R	E
shift	2	4	3	1

32.2 Algorithm

- Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table.
- Step 2 Align the pattern against the beginning of the text.
- Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text.

  If matching is not found shift the pattern by the length given by the shift table.

```
HorspoolMatching(P [0..m  1], T [0..n  1])

//Implements Horspool's algorithm for string matching
//Input: Pattern P[0...m  1] and text T [0..n  1]
//Output: The index of the left end of the first matching substring
// or -1 if there are no matches

ShiftTable(P [0..m  1]) //generate Table of shifts

i <- m - 1 //position of the pattern's right end
while i <= n  1 do

   k <- 0 //number of matched characters
   while k <= m  1 and P[m1-k]= T[ik] do
     k <- k + 1

   if k = m
      return i  m + 1
   else i <- i + Table[T[i]]

return -1
```

32.3 Example

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R         B A R B E R             shift(A)=4  shift(B)=2
        B A R B E R     B A R B E R         shift(E)=1  shift(R)=3
```

```
B A R B E R          B A R B E R       shift(-)=6  match
```

33 UNIT 6 Limitations (AL Ch.11)

LIMITATIONS OF ALGORITHMIC POWER AND COPING WITH THEM: Lower-Bound Arguments, Decision Trees, P, NP, and NPComplete Problems, Challenges of Numerical Algorithms.

34 Lower bounds

Lower bounds, which are estimates on a minimum amount of work needed to solve a problem. In general, obtaining a nontrivial lower bound even for a simple-sounding problem is a very difficult task.

- worst case lower bound
- average case lower bound

We can use the lower bound to measure,

1. The difficulty of a problem.
2. Can answer if the algorithm for a problem is the best?

The lower bound for a problem is not unique. e.g. Ω(1), Ω(n), Ω(n log n) are all lower bounds for sorting.

At present, if the highest lower bound of a problem is Ω(n log n) and the time complexity of the best algorithm is O(n2).

- 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 Ω(n log n) for a problem and there is an algorithm with time complexity O(n log n), then the algorithm is optimal.

34.1 Trivial lower bounds

Count the number of inputs to be processed and the number outputs to be produced.

Examples:

- Generate all permutations of distinct objects $\Omega(\mathrm{n}!)$
- Polynomial evaluation of degreen n is $\Omega(\mathrm{n})$
- Matrix multiplication is $\Omega(\mathrm{n}^2)$

34.2 Information theoretic arguments

Deducing a positive integer between 1 and n selected by somebody by asking that person questions with yes/no answers.

34.3 Adversary arguments

Playing a guessing game between you and your friend.

- You are to pick up a date, and the friend will try to guess the date by asking YES/NO questions.
- Your purpose is forcing your friend to ask as many questions as possible.

Idea:

- You did not pick up a date in advance at all, but Construct a date according to the questions.
- The requirement is that the finally constructed date should be consistent to all your answers to the questions.
- Looks like cheating, but it is a good way to find the lower bound.

34.4 Problem reduction

To show that problem P is at least as hard as another problem Q with a known lower bound, we need to reduce Q to P (not P to Q!).

In other words, we should show that an arbitrary instance of problem Q can be transformed (in a reasonably efficient fashion) to an instance of problem P, so any algorithm solving P would solve Q as well.

35 Decision Trees

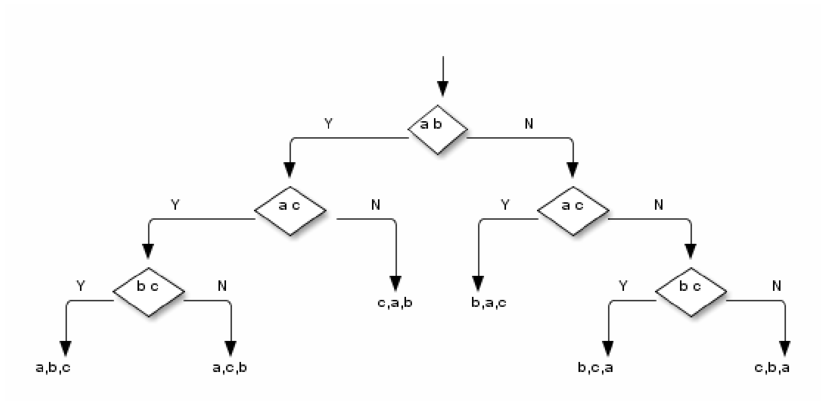
Assumptions:

- All numbers are distinct (so no use for ai = aj)
- All comparisons have form ai < aj

Decision tree model:

- Full binary tree
- Internal node represents a comparison.
- Each leaf represents one possible result.
- The height (i.e., longest path) is the lower bound.

Example: decission tree for comparison sort of a,b,c



Theorem: Any comparison sort algorithm requires Ω(nlg n) comparisons in the worst case

Proof:

- Suppose height of a decision tree is h, and number of paths (i,e,, permutations) is n!.
- Since a binary tree of height h has at most 2h leaves,

$$\begin{aligned} h &\geq \log n \\ &\geq \log n! \\ &\geq \log (\sqrt{}\{2\pi\ n\}(n/e)^{n}) \\ &\geq n\log n \\ &= \Omega(n\log n) \end{aligned}$$

That is to say: any comparison sort in the worst case needs at least nlg n comparisons.

36 Class P and NP problems

36.1 P (Polynomial)

Class P is a class of decision problems that can be solved in polynomial time by deterministic algorithm.

Examples of P:

1. GCD of two numbers
2. Sorting a list
3. Searching for a key
4. FInding MST
5. Finding shortest paths

Such problems are also known as tractable.

36.2 NP (Non-deterministic Polynomial)

Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial.

Examples of NP:

1. Hamiltonian Circuit Problem
2. Travelling Salesperson Problem
3. Knapsack problem
4. Graph coloring problem

NP is the set of problems whose solutions can be verified in polynomial time. But as far as anyone can tell, many of those problems take exponential time to solve.

Such problems are also known as intractable.

36.3 NP Complete

A decision problem D is said to be NP-complete if

1. It belongs to class NP
2. Every problem in NP is polynomially reducible to D

Interest in NP complete problems is if one finds an algorithm to solve NP complete in polynomial time that other NP problems can be solved in polynomial time.

36.4 P=NP

The equivalency of P and NP. Roughly speaking,

- P is a set of relatively easy problems

- NP is a set of what seem to be very, very hard problems

So P = NP would imply that the apparently hard problems actually have relatively easy solutions.

37 Challenges of Numerical algorithms

Numerical analysis is about algorithms for solving mathematical problems.

Examples:

1. Solving equations and systems of equations,
2. Evaluating functions such as sin x, ln x
3. computing integrals, etc.

Challenges:

1. Truncation errors:
   e^x using Taylor series
2. Round off errors
   Caused by the limited accuracy with which we can represent real numbers in a digital computer.

38 UNIT 7 Coping with Limitations (AL Ch.12)

COPING WITH LIMITATIONS OF ALGORITHMIC POWER: Backtracking: n - Queens problem, Hamiltonian Circuit Problem, Subset – Sum Problem. Branch-and-Bound: Assignment Problem, Knapsack Problem, Traveling Salesperson Problem. Approximation Algorithms for NP-Hard Problems – Traveling Salesperson Problem, Knapsack Problem

39 Introduction

There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems) i) Exact solution strategies ii) Approximation strategy.

Exact solution strategies guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time.

- Exhaustive search (brute force)
  - Useful only for small instances dynamic programming
  - Applicable to some problems (e.g., the knapsack problem)
- Backtracking
  - Eliminates some unnecessary cases from consideration.
  - Yields solutions in reasonable time for many instances
  - But worst case is still exponential.
- Branch-and-bound
  - Further refines the backtracking idea for optimization problems.

Approximation strategy is used to get a solution in polynomial time that is not necessarily optimal.

- Accuracy ratio of an approximate solution sa,
  - r(sa) = f(sa) / f(s*) for minimization problems
  - r(sa) = f(s*) / f(sa) for maximization problems

  where f(sa) and f(s*) are values of the objective function f for the approximate solution sa and actual optimal solution s*

- Performance ratio of the algorithm A,
  - The lowest upper bound of r(sa) on all instances

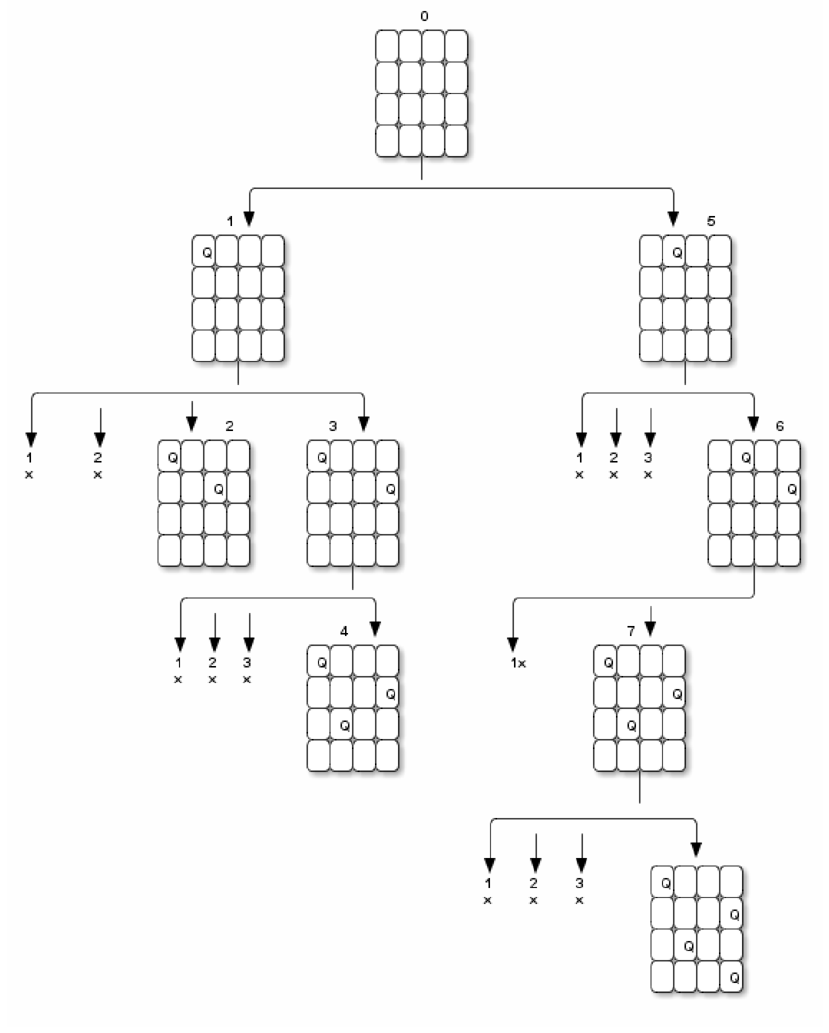
40 Backtracking

- Construct the state-space tree.
  - nodes: partial solutions
  - edges: choices in extending partial solutions
- Explore the state space tree
  - Using depth-first search.
  - "Prune" nonpromising nodes.
  - Stop exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search.

40.1 n-Queens Problem.

Place n queens on an n-by-n chess board so that no two of them are in the same row, column, or diagonal

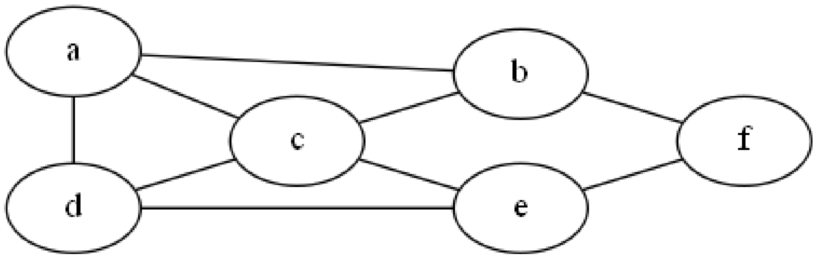
State-Space Tree of the 4-Queens Problem.

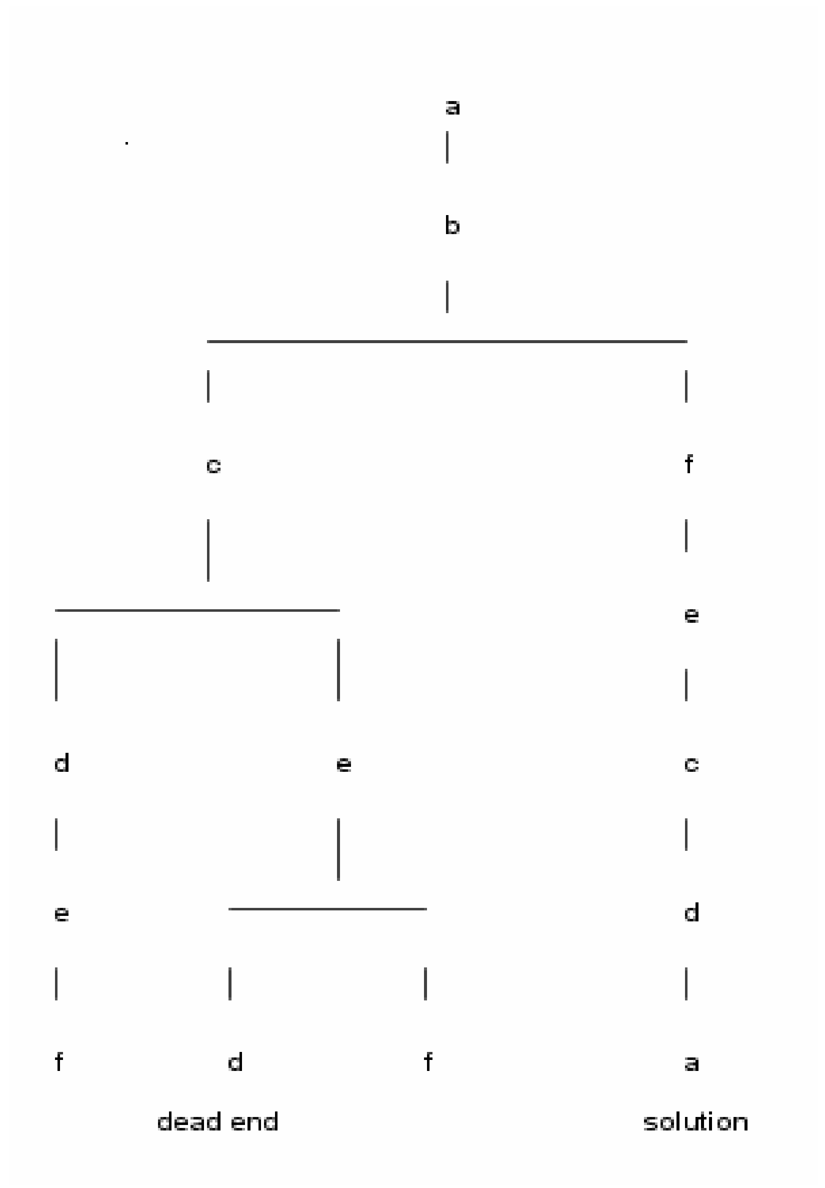


State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

For 8x8 board, the number of distinct solutions are 92.

40.2 Hamiltonian Circuit Problem

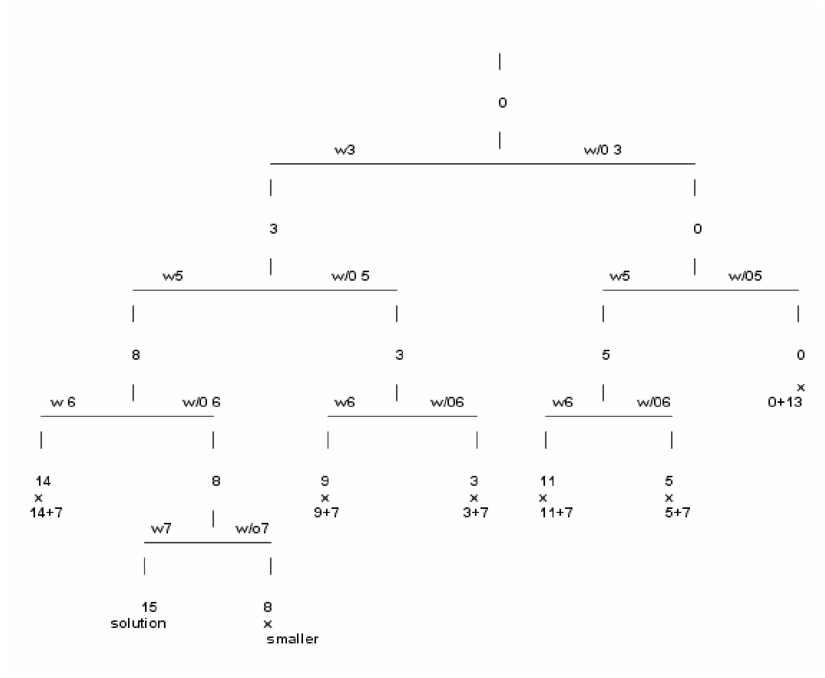




40.3 Subset Problem

Find a subset of a given set A = {a1,. . . , an} of n positive integers whose sum is equal to a given positive integer d.

State space tree for A={3,5,6,7} and d=15



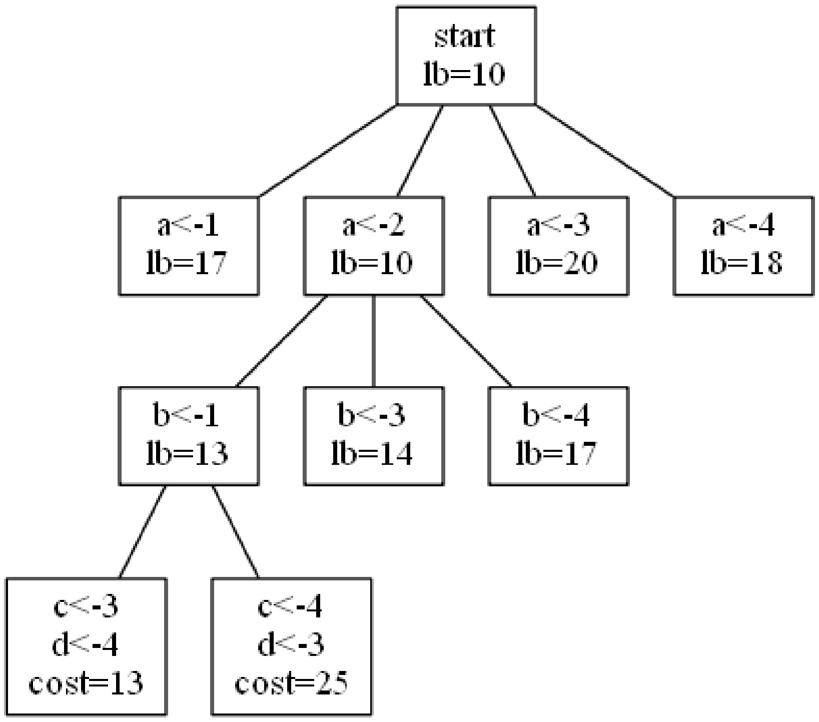
41 Branch-and-Bound

- Is an enhancement of backtracking for solving optimization problems
- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)
- Uses the bound for ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far.

41.1 Assignment Problem

Select one element in each row of the cost matrix C so that, no two selected elements are in the same column the total cost of assignment is minimized.

	Job 1	Job 2	Job 3	Job 4
Person A	9	2	7	8
Person B	6	4	3	7
Person C	5	8	1	8
Person D	7	6	9	4



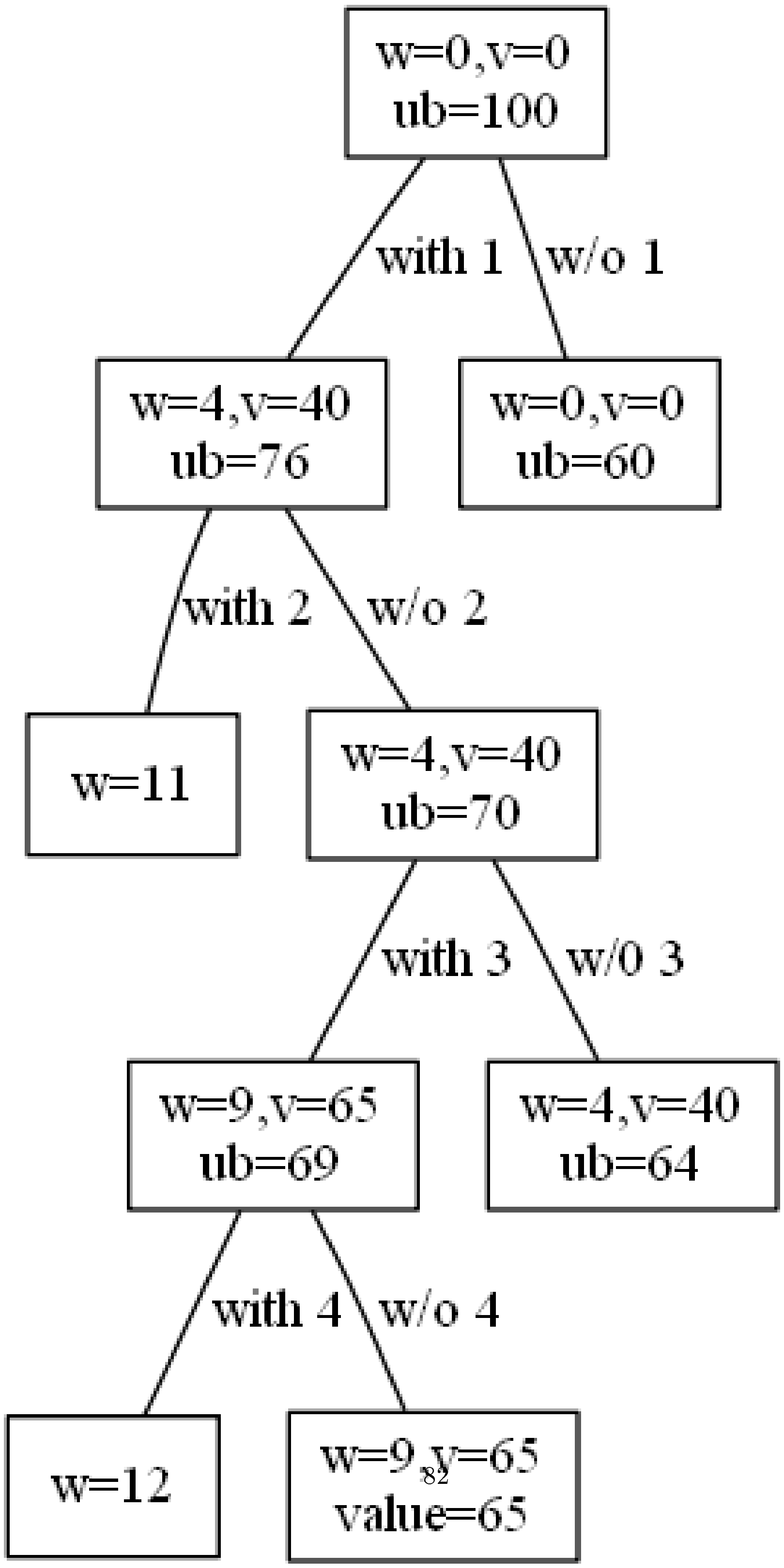
1. start/lb=2+3+1+4=10
2. Level 1, jobs 1,2,3,4 to person A
   (a) a <- 1, lb=9 + 3 + 1 + 4=17 x
   (b) a <- 2, lb=2 + 3 + 1 + 4=10 optimal solution
   (c) a <- 3, lb=7 + 4 + 5 + 4=20 x
   (d) a <- 4, lb=8 + 3 + 1 + 6=18 x
3. Level 2, jobs 1,3,4 to person B
   (a) b <- 1, lb=2+6+1+4=13 optimal solution
   (b) b <- 3, lb=2+3+5+4=14 x
   (c) b <- 4, lb=2+7+1+7=17 x

4. Level 3, jobs 3, 4 to person C and D
    (a) c <- 3, d <- 4, lb=2+6+1+4=13 optimal solution
    (b) c <- 4, d <- 3, lb=2+6+8+9=25 x

41.2 Knapsack problem

The knapsack capacity W=10

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

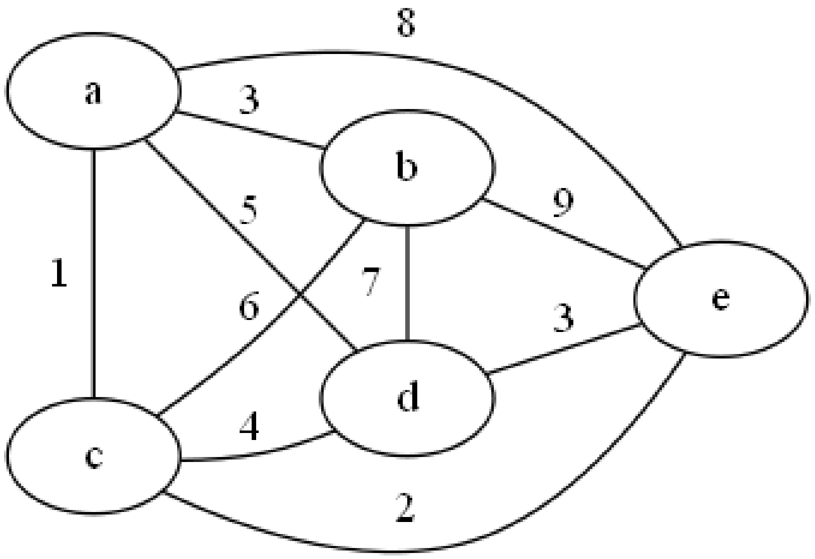

w=0,v=0
ub=100
with 1
w/o 1
w=4,v=40
ub=76
w=0,v=0
ub=60
with 2
w/o 2
w=11
w=4,v=40
ub=70
with 3
w/0 3
w=9,v=65
ub=69
w=4,v=40
ub=64
with 4
w/o 4
w=12
w=9,v=65
value=65

$ub = v + (W \quad w)(v_{i+1}/w_{i+1})$.

Evaluate solutions at each level based on upper bound. Keep the optimal solution and discard others.

1. Level 1
   (a) w=0,v=0,ub=0+(10-0)x40/4=100
2. Level 2
   (a) with i=1, w=4,v=40,ub=40+(10-4)x6=76
   (b) w/o i=1, w=0,v=0, ub=0+(10-0)x6=60 x
3. Level 3
   (a) with i=2, w=11, x
   (b) w/o i=2, w=4,v=40,ub=40+(10-4)x5=30
4. Level 4
   (a) with i=3, w=9, v=65, ub=65+(10-9)x4=69
   (b) w/0 i=3, w=4, v=40, ub=40+(10-4)x4=64 x
5. Level 5
   (a) with i=4, w=12, x
   (b) w/o i=4, w=9, v=65, ub=65+(10-9)x0=65 optimal solution

41.3 Traveling Salesman Problem Approximation Approach



Assumptions:

1. We can consider only tours that start at a.
2. Because our graph is undirected, we can generate only tours in which b is visited before c.
3. After visiting n1 = 4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one.

Lowerbound:

Compute a lower bound on the length l of any tour as follows. For each city i, 1 i n, find the sum si of the distances from city i to the two nearest cities; compute the sum s of these n numbers, divide the result by 2.

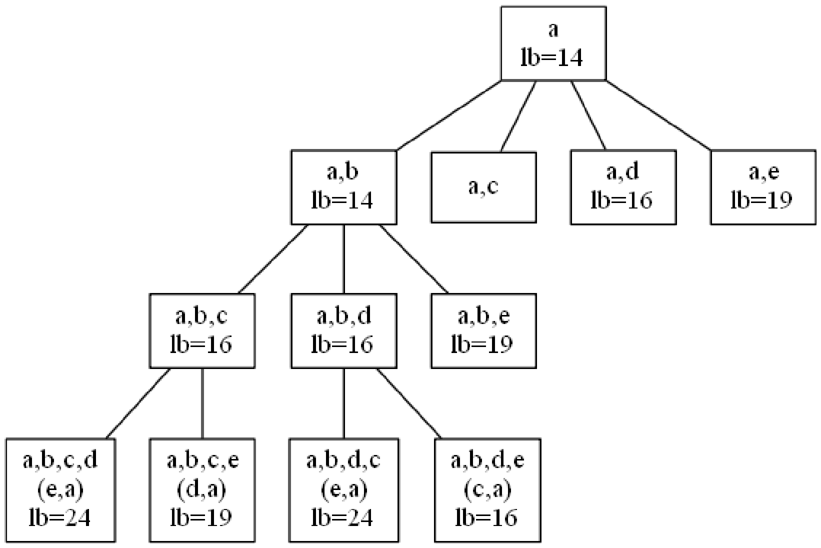
lb = s/2

For example, for the instance in Figure 12.9a, formula (12.2) yields,

lb = [(1+ 3) + (3 + 6) + (1+ 2) + (3 + 4) + (2 + 3)]/2 = 14.

Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound accordingly. For example, for all the Hamiltonian circuits of the graph that must include edge (a, d), we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges (a, d) and (d, a):

$[(1+ 5) + (3 + 6) + (1+ 2) + (3 + 5) + (2 + 3)]/2 = 16.$



42 Approximation Algorithms for NP hard problems

Solve NP hard problems approximately by a fast algorithm. This approach is particularly appealing for applications where a good but not necessarily optimal solution will suffice. Besides, in real-life applications, we often have to operate with inaccurate data to begin with. Under such circumstances, going for an approximate solution can be a particularly sensible choice.

most of them are based on some problem-specific heuristic. A heuristic is a common-sense rule drawn from experience rather than from a mathematically proved assertion.

Accuracy measures:

accuracy ratio of an approximate solution sa,

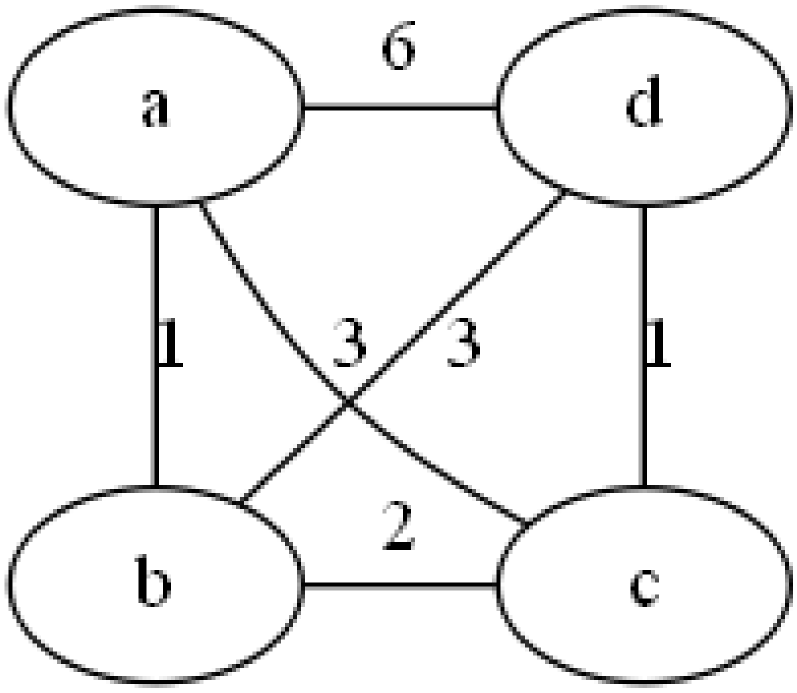
1. r(sa) = f(sa) / f(s*) for minimization problems
2. r(sa) = f(s*) / f(sa) for maximization problems

where f(sa) and f(s*) are values of the objective function f for the approximate solution sa and actual optimal solution s*

42.1 TSP

42.1.1 Nearest-Neighbor Algorithm for TSP

Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one.



Approximate tour is (a,b).1 -> (b,c).2 -> (c,d).1 -> (d,a).6. Length of the tour is 10.

Optimal tour is (a,c).3 -> (c,d).1 -> (d,b).3 -> (b,a)1. Length of the tour is 8.

Accuracy: RA = (unbounded above) – make the length of AD arbitrarily large in the above example.

r(sa) = f(sa) / f(s*) = 10/8

42.1.2 Multifragment-heuristic algorithm

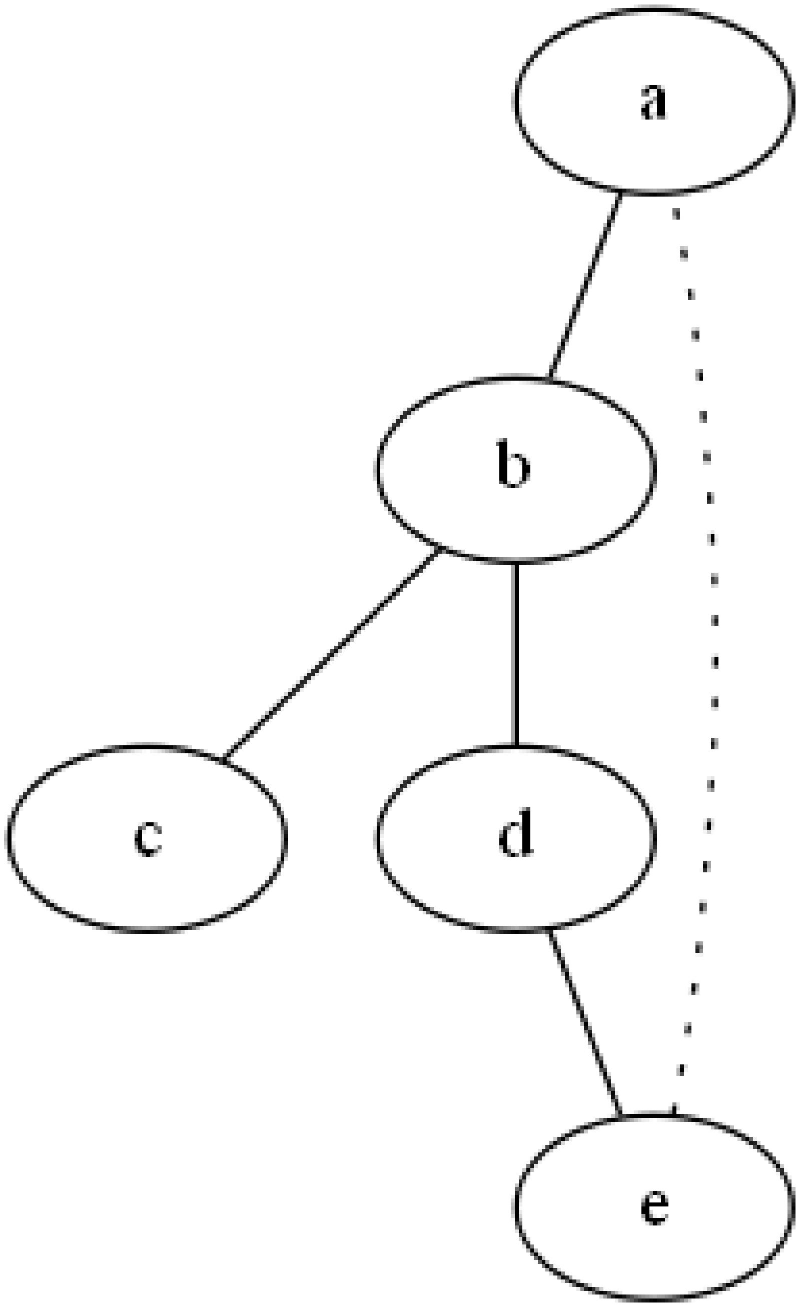
- Step 1 Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.
- Step 2 Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.
- Step 3 Return the set of tour edges.

Example: (Same G used in nearest neighbor)

Edges	Sorted edges	Edges considered
(a,b).1	(a,b).1	(a,b).1
(a,d).6	(c,d).1	(c,d).1
(a,c).3	(b,c).2	(b,c).2
(b,c).2	(a,c).3	-
(b,d).3	(b,d).3	-
(c,d).1	(a,d).6	(a,d).6

42.1.3 Twice-around-the-tree algorithm

- Step 1 Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.
- Step 2 Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (This can be done by a DFS traversal.)
- Step 3 Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (This step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.



a, b, c, b, d, e, d, b, a

a, b, c, d, e, a

42.1.4 Christofides Algorithm

- Stage 1: Construct a minimum spanning tree of the graph
- Stage 2: Add edges of a minimum-weight matching of all the odd vertices in the minimum spanning tree
- Stage 3: Find an Eulerian circuit of the multigraph obtained in Stage 2
- Stage 4: Create a tour from the path constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

It has four odd-degree vertices: a, b, c, and e. The minimum-weight matching of these four vertices consists of edges (a, b) and (c, e). (For this tiny instance, it can be found easily by comparing the total weights of just three alternatives: (a, b) and (c, e), (a, c) and (b, e), (a, e) and (b, c).) The traversal of the multigraph, starting at vertex a, produces the Eulerian circuit a b c e d b a, which, after one shortcut, yields the tour a b c e d a of length 37.

42.2 Knapsack

42.2.1 Greedy algorithm for the discrete knapsack problem

- Step 1 Compute the value-to-weight ratios ri = vi/wi, i = 1, . . . , n, for the items given.
- Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)
- Step 3 Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.

42.2.2 Greedy algorithm for the continuous knapsack problem

- Step 1 Compute the value-to-weight ratios vi/wi, i = 1, . . . , n, for the items given.
- Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

- Step 3 Repeat the following operation until the knapsack is filled to its full capacity or no item is left in the sorted list: if the current item on the list fits into the knapsack in its entirety, take it and proceed to the next item; otherwise, take its largest fraction to fill the knapsack to its full capacity and stop.

43 UNIT 8 PRAM (HS Ch.13)

PRAM ALGORITHMS: Introduction, Computational Model, Parallel Algorithms for Prefix Computation, List Ranking, and Graph Problems,

44 Introduction

The algorithm is executed sequentially on a single processor. The run time of the algorithm can be speeded up if we employ more processors.

$\mathrm{T_s(n)}$ is the time complexity of sequential algorihm $\mathrm{T_p(n)}$ is the time complexity of the parallel algorithm

Speedup = $\mathrm{T_s(n)/T_p(n)}$

The speedup is called linear speedup if Ts(n)/Tp(n) = p. It is called super linear speedup if Ts(n)/Tp(n) < p.

Amdhal's Law

$\frac{1}{f+\frac{1-f}{p}}$

Where, f is the fraction of the algorithm that cannot be parallelized.

The Amdhal's law says that the speedup is inversely proportional to the fraction of the algorithm that cannot be parallelized.

p	f	Speedup	Remarks
10	1	1	No improvement wit p processors
10	.5	2	slightly better
10	.1	5	
10	.01	9	

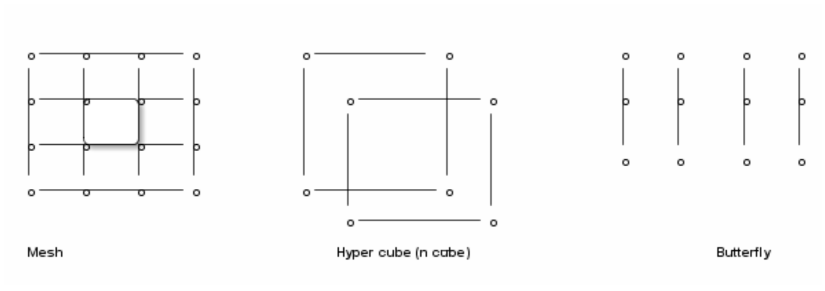
45 Computational Model

RAM model

The Random Access Machine Model (RAM model) of serial computers:

- Memory is a sequence of words.
- Each memory access takes one unit of time.
- Basic operations (add, multiply, compare) take one unit time.
- Instructions are not modifiable.
- Read-only input tape, write-only output tape
- Fixed connection modeil
- Shared memory model (PRAM)

45.1 Fixed connection model



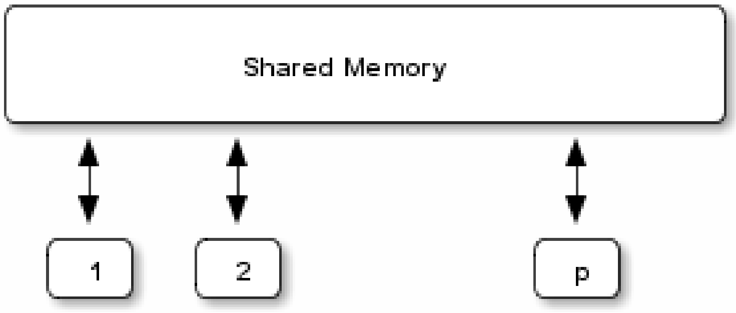
Processors are connected through communication links. The commnication time depends on the path length.

45.2 PRAM

The Parallel Random Access Machine (PRAM model) of paralel computers:

- P processors, each with its own unmodifiable program.
- A single shared memory composed of a sequence of words.
- a read-only input tape, a write-only output tape.

PRAM model is a synchronous, MIMD, shared address space parallel computer.



45.3 Read/Write Conflicts

The global memory in PRAM model is shared by all the processors. Simultaneous modification of global memory by two or more processors lead to inconsistent data. The various models to rsolve read-write conflicts are: EREW, CREW, ERCW and CRCW.

	Exclusive Read	Concurrent Read
Exclusive Write	EREW	CREW
	Single processor reads and writes	Multiple processors read, single one writes.
Concurrent Write	ERCW	CRCW
	Not interesting.	Multiple processors read and write.

CW is allowed if the processors have the message to write.

45.4 Work-Time paradigm

For sequential programs, performance or time complexity T(n) is measured as the number of operations.

For parrallel programs, performance or time complexity T(n) is measured as the number of time steps. And work complexity or efficiency is measured as the number of operations.

Example: vector addition

	u1	u2	u3	u4	...	un
+	v1	v2	v3	v4	...	vn
	u1+v1	u2+v2	u3+v3	u4+v4		un+vn

1. Work = number of operations O(n)
2. Time = bumber of time steps O(1)

A parallel algorithm is work optimal if W(n) = Ts(n).

46 Prefix sum

- We are given an ordered set A of n elements and a binary associative operator $\oplus$.

  A = {a0, a1, a2, ..., a_{n-1}}

- We have to compute the ordered set

  A = {a0,(a0 $\oplus$ a1),...,(a0 $\oplus$ a1 $\oplus$,...,a_{n-1})}

Example: if $\oplus$ is + and the input is ordered set

{5, 3, -6, 2, 7, 10, -2, 8}

Then the output is,

{5, 8, 2, 4, 11, 21, 19, 27}

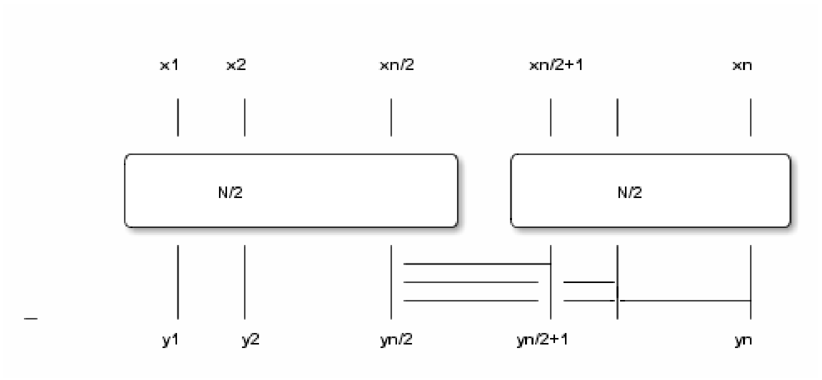
46.1 Sequential algorithm

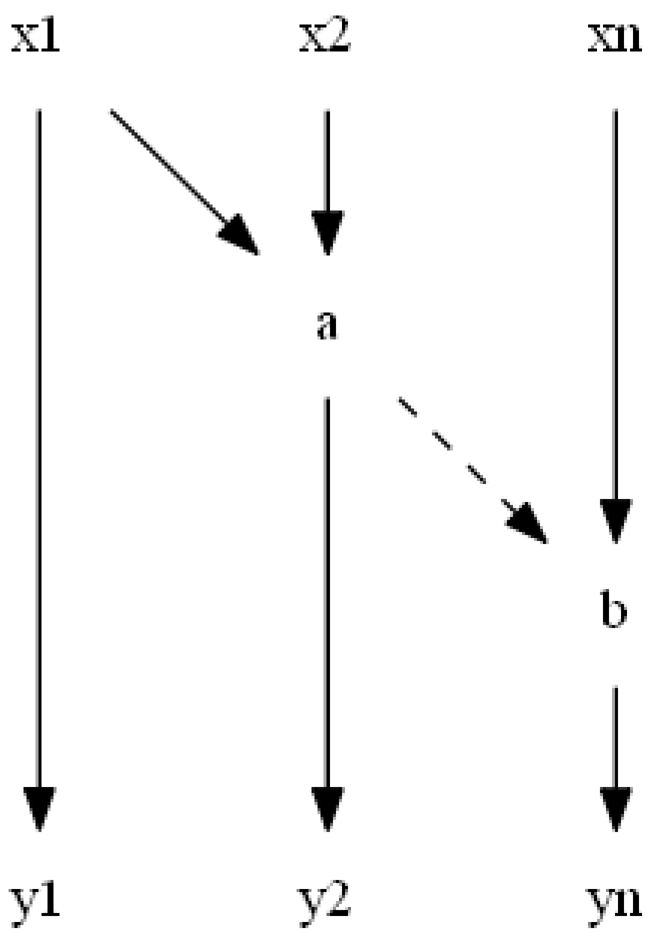
```
int prfix[n];

for(i=2;i<=n;i++)
  prefix[i] = prefix[i-1] + prefix[i];
```

T(n) = O(n)

46.2 Parallel algoritm with n processors

- Divides the elements into lower half upper half recursively. (upper/lower algorithm)
- Prefix sum is independently computed on the lower and upper halves.
- Output from the highest element of the lower half is added to each output of the upper half.





Let the input be x1, x2, . . . , xn. Employ divide-and-conquer strategy.

Step 0. If n = 1, one processor outputs x1

Step 1. Let the first n/2 processors recursively compute the prefixes x1, x2, xn/2 and let y1, y2, . . . , yn/2 be the result. At the same time let the rest of the processors recursively compute the prefixes of xn/2+1, xn/2+2, . . . , xn and let yn/2+1, yn/2+2, . . . , yn be the output.

Step 2. Note that the first half of the final answer is the same as y1, y2, . . . , yn/2. The second half of the final answer is yn/2 ⊕ yn/2+1, yn/2 ⊕ yn.

Let the second half of the processors read yn/2 concurrently from the global memory and update theire answers. This step takes O(1) time.

```
T(n) = T(n/2) + O(1)
     = O(logn)
```

Is it work optimal?

Sequential algorithm	$T_s(n)$	O(n)
Parallel algorithm	$T_p(n)$	O(logn)
	$W_p(n)$	n processors x O(logn)

The algorithm is not work optimal if $T_s(n) \neq W_p(n)$.

If we have n/logn processors and the $T_p(n) = O(\log n)$ then the algorithm becomes work ptimal.

46.3 Work optimal parallel algoritm with n/logn processors

```
5,12,8,6      3,9,11,12    1,5,6,7    10,4,3,5

   |             |            |           |
   v             v            v           v
5,17,25,31    3,12,23,35   1,6,12,19  10,14,17,22

                   |
                   v
              31, 35, 19, 22

                   | prefix sum
                   v
              31, 66, 85,107

                   |
                   v
5,17,25,31    31 + (3,12,23,35)   66 + (1,6,12,19)   85 + (10,14,17,22)
```

Step 1. Processor i (i=1,,2,. . . ,n/logn) in parallel computes the prefix of its logn elements sequentially. This takes O(logn).

Step 2. A total of n/logn processors employ DaC algorithm to compute the prefixes of elements, one from each group having n/logn elements. This takes O(log(n/logn)) = o(logn)

Step 3. Each processor updates the prefixes computed in step 1 by the prefixes from step 2. This takes O(logn)

T(n) = O(logn) is the combined time complexity of steps 1,2 and 3

47 List Ranking

Given a singly linked list L with n objects, for each node, compute the distance to the end of the list.

A[1..n]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Neighbor	5	4	2	0	3	1

A[6] -> A[1] -> A[5] -> A[3] -> A[2] -> A[4] -> null

47.1 Sequential algorithm

1. Determine the list head by examining A[1..n] to identify i, $1 \leq i \leq n$, such that A[j] $\neq$ i, $1 \leq j \leq n$

   In the above example 6 is the head. .

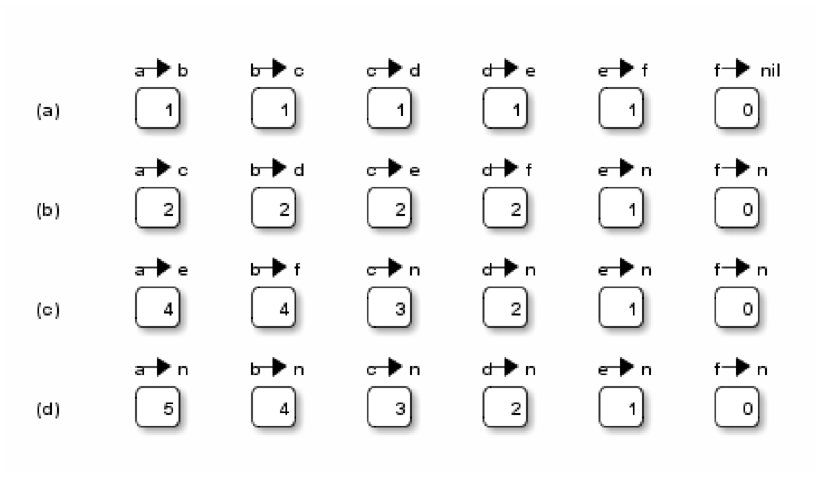
2. Scan the list left to right and assign ranks n-1, n-2, . . . , 0 in this order.

T(n) = O(n)

47.2 Parallel algorithm

```
for q <- 1 to logn do
        processor i (in parallel for $1 \leq i \leq n$) does:
              if Neighbor[i] $\neq$ 0
                    Rank[i] <- Rank[i] + Rank[Neignbor[i]
                    Neighbor[i] <- Neighbor[Neighbor[i]]
```

T(n) = O(logn) when we employ n processors.



(a)
a→b 1
b→c 1
c→d 1
d→e 1
e→f 1
f→nil 0
(b)
a→c 2
b→d 2
c→e 2
d→f 2
e→n 1
f→n 0
(c)
a→e 4
b→f 4
c→n 3
d→n 2
e→n 1
f→n 0
(d)
a→n 5
b→n 4
c→n 3
d→n 2
e→n 1
f→n 0