SEMESTER- I

Core Course-I-DESIGN AND ANALYSIS OF ALGORITHMS

UNIT - I

Introduction: Algorithm Definition – Algorithm Specification – Performance Analysis - Asymptotic Notations. Elementary Data Structures: Stacks and Queues – Trees – Dictionaries – Priority Queues–Sets and Disjoint Set Union – Graphs .

Algorithm Definition:

• An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

Created with OfficeSuite

- Input. Zero or more quantities are externally supplied.

- Output. At least one quantity is produced.

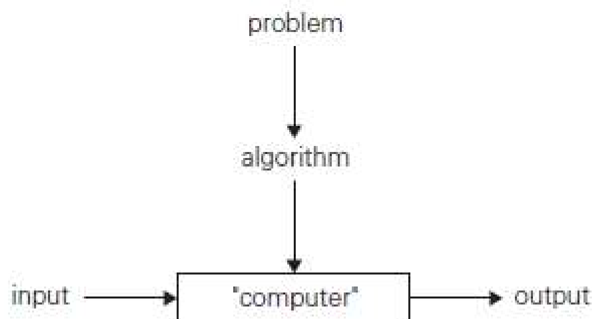
- Definiteness. Each instruction is clear and unambiguous.

- Finiteness. The algorithm terminates after a finite number of steps.

- Effectiveness. Every instruction must be very basic enough and must be feasible.

- 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.

Created with OfficeSuite



Algorithm Specification

1. **Name of algorithm:** Every algorithm is given an identifying name written in capital letters.
2. **Introductory comment:** The algorithm name is followed by a brief description of the tasks the algorithm performs and any assumptions that have been made. The description gives the name and types of the variables used in the algorithm. Comment begins with // and continues until end of line. Comments specify no action and are included only for clarity.
3. **Steps:** Each algorithm is made of a sequence of numbered steps and each step has an ordered sequence of statements, which describe the tasks to be performed. The statements in each step are executed in a left-to-right order.
4. **Data type:** Data types are assumed simple such as integer, real, char, boolean and other data structures such as array, pointer, structure are used. Array i^{th} element can be described as A[i] and (i, j) element can be described as A[i, j]. Structure data type can be formed as

```
node =record
{
        data type-1        identifier 1;
        ...
        data type-n        identifier n;
        node     *link;
}
```

link is a pointer to the record type node.

Efficiency of Algorithms:

- The performances of algorithms can be measured on the scales of time and space.

Created with OfficeSuite

- The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental.

- In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity: The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

- The time complexity of an algorithm can be computed either by an empirical or theoretical approach.

Created with OfficeSuite

- The empirical or posteriori testing approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem.

- The time taken by the execution of the programs for various instances of the problem are noted and compared.

- The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

Analysis of Algorithms:

- Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program.
- That gives us a measure that will work for different operating systems, compilers and CPUs.
- The asymptotic complexity is written using big-O notation.

Rules for using big-O:

The most important property is that big-O gives an

Created with OfficeSuite

upper bound only. If an algorithm is O(n2), it doesn’t have to take n2 steps (or a constant multiple of n2).

Created with OfficeSuite

take more than n2. So any algorithm that is O(n), is also an O(n2) algorithm.

Created with OfficeSuite

- If this seems confusing, think of big-O as being like "<". Any number that is < n is also < n2.

1. Ignoring constant factors: O(c f(n)) = O(f(n)), where c is a constant; e.g. O(20 n3) = O(n3)

2. Ignoring smaller terms: If a<b then O(a+b) = O(b), for example O(n2+n)

= O(n2)

3. Upper bound only: If a<b then an O(a) algorithm is also an O(b) algorithm. For example, an O(n) algorithm is also an O(n2) algorithm (but not vice versa).

4. n and log n are "bigger" than any constant, from an asymptotic view (that means for large enough n). So if k is a constant, an O(n + k) algorithm is also O(n), by ignoring smaller terms. Similarly, an O(log n + k) algorithm is also O(log n).

5. Another consequence of the last item is that an O(n log n + n) algorithm, which is O(n(log n + 1)), can b

Created with OfficeSuite

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity f(n) of M increases as n increases.

It is usually the rate of increase of f(n) with some standard functions. The mostcommon computing times are O(1), O(log2 n), O(n), O(n log2 n), O(n2), O(n3), O(2n).

Asymptotic Notations:

- It is often used to describe how the size of the input data affects an algorithm's usage of computational resources.

- Running time of an algorithm is described as a function of input size n for large n.

Big oh(O): Definition: f(n) = O(g(n)) (read as f of n is big oh of g of n) if there exist a positive integer n0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n0$. Here g(n) is the upper bound of the function f(n).

Created with OfficeSuite

Omega(Ω): Definition: $f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), if there exists a positive integer n0 and a positive number c such that $|f(n)| \geq c\,|g(n)|$ for all $n \geq n0$. Here g(n) is the lower bound of the function f(n).

Theta(Θ): Definition: $f(n) = \Theta(g(n))$ (read as f of n is theta of g of n), if there exists a positive integer n0 and two positive constants c1 and c2 such that $c1\,|g(n)| \leq |f(n)| \leq c2\,|g(n)|$ for all $n \geq n0$. The function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, $n \geq n0$.

Data Structure:

- ***Linear data structure***

- ***Non Linear Data Structure***

Stack

- Stack is an Abstract data structure works on the principle Last In First Out Last In.

Created with OfficeSuite

- The last element add to the stack is the first element to be delete.

- Insertion and deletion can be takes place at one end called TOP.

- It looks like one side closed tube.

Stack operation:

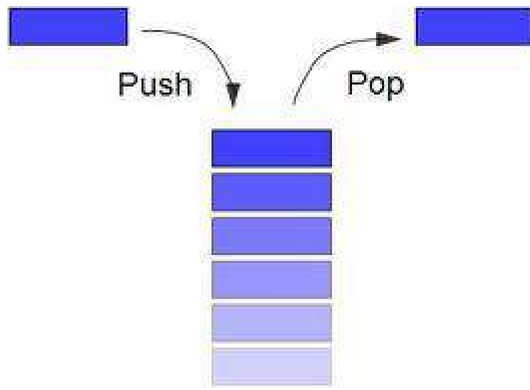
Push( )

Add operation of the stack is called push operation.

Pop( )

Delete operation is called as pop operation.

Created with OfficeSuite

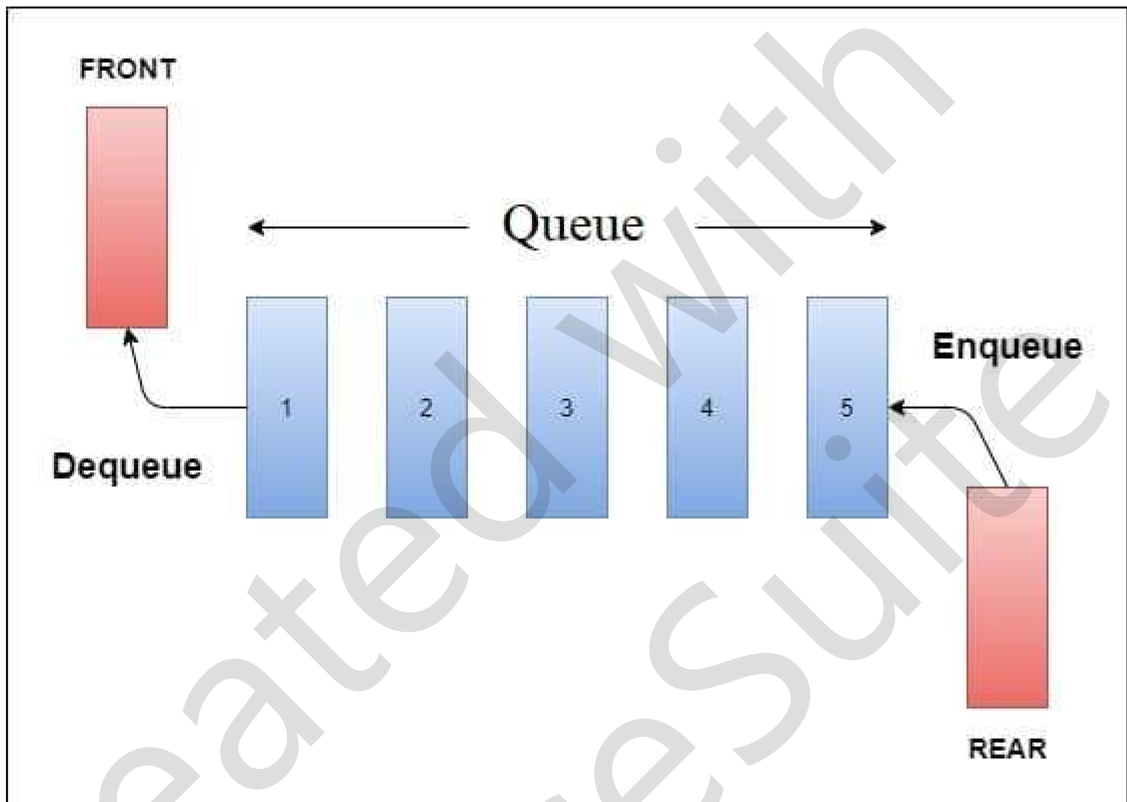


Queues:

- A queue is a data structure that is best described as "first in, first out".
- A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front.
- A real world example of a queue is people waiting in line at the bank.

Created with OfficeSuite

- As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the linear.



Created with OfficeSuite

Queue operations using array:

- In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*.

- The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue front = rear if and only if there are no elements in the queue.

- The initial condition then is front = rear = 0.

- The various queue operations to perform creation, deletion and display the elements in a queue are as

follows:

1. **insert Q( ):** inserts an element at the end of queue Q.

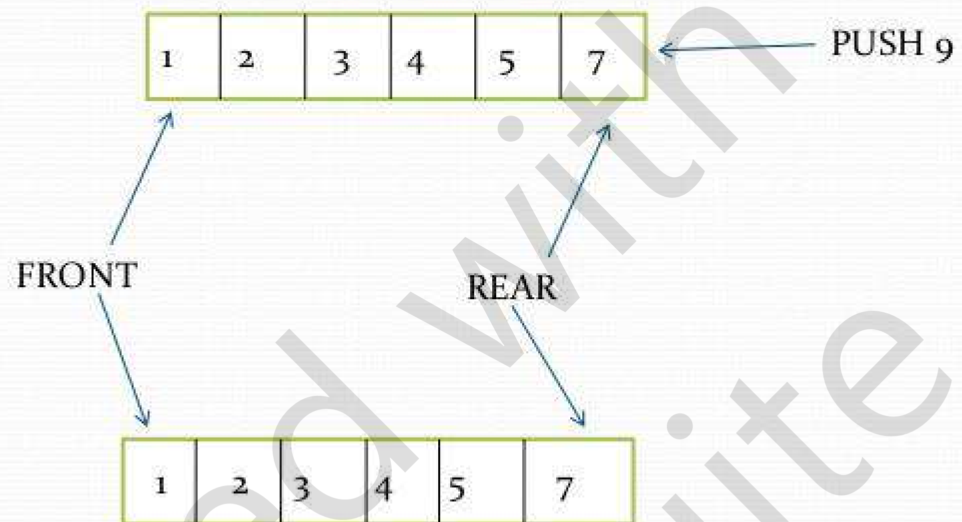
2. **delete Q( ):** deletes the first element of Q.

3. **display Q( ):** displays the elements in the queue.

Created with OfficeSuite

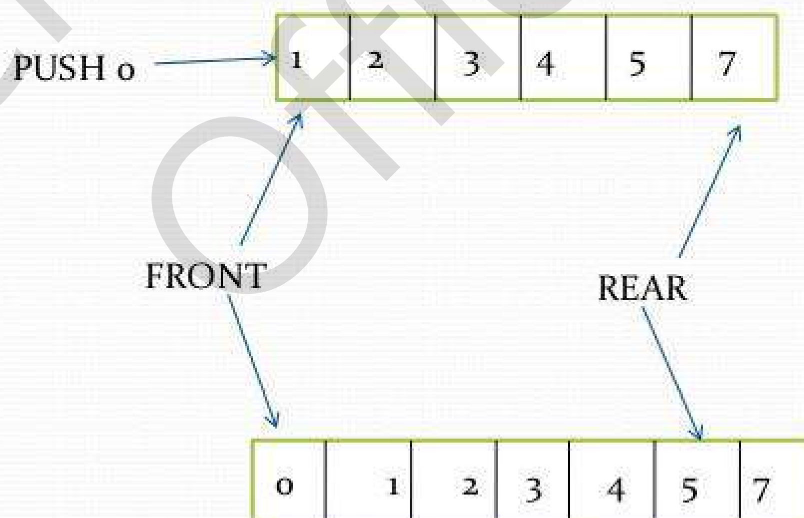
Insert_back

- insert_back() is a operation used to push an element at the back of a *Deque*.



Insert_front

- insert_front() is a operation used to push an element into the front of the *Deque*.



Created with Office Suite

Priority queue:

- ✓Priority Queue is more specialized data structure than Queue.
- ✓Like ordinary queue, priority queue has same method but with a major difference.
- ✓In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value.
- ✓Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- ✓insert / enqueue − add an item to the rear of the queue.
- ✓remove / dequeue − remove an item from the front of the queue.

Created with OfficeSuite

Priority Queue Representation

- ✓ We're going to implement Queue using array in this article.

- ✓ There is few more operations supported by queue which are following.

- ➢ Peek − get the element at front of the queue.

- ➢ isFull − check if queue is full.

- ➢ isEmpty − check if queue is empty.

Insert / Enqueue Operation

- Whenever an element is inserted into queue, priority queue inserts the item according to its order.

- Here we're assuming that data with high value has low priority.

Remove / Dequeue Operation

Created with OfficeSuite

- Whenever an element is to be removed from queue, queue get the element using item count.
- Once element is removed. Item count is reduced by one.

Non Linear Data Structures:

- The data structure where data items are not organized sequentially is called non linear data structure.
- A data elements of the non linear data structure could be connected to more than one elements to reflect a special relationship among them.
- All the data elements in non linear data structure can not be traversed intraversed in single run.

Examples of non linear data structures
are Trees and Graphs.

Tree

Created with Office Suite

- A tree is a hierarchical data structure composed of nodes.

- **Root**: the top-most node (unlike real trees, trees in computer science grow downward!). Every (non-empty) tree has one.

- **Parent**: the node connected directly above the current one   Everychild(except for the root) has one.

- **Child**: a node connected below the current one. Each node can have 0 or more.

- **Leaf**: a node that has no children.

- **Depth/Level**: the length of the path (edges) from the root to a node(depth/level of the root is 0).

- Tree Height: the maximum depth from of any node in the tree.

Created with OfficeSuite

- A tree commonly used in computing is a binary tree.

- A binary tree consists of nodes that have at most 2 children.

GRAPH

- ✓ A graph is a data structure that contains of a set of vertices and a set of edges which connect pairs of the vertices.

- ✓ A vertex (or node) can be connected to any number of other vertices using edges.

- ✓ An edge may be bidirectional or directed (one-way).

- ✓ An edge may have a weight on it that indicates a cost for traveling over that edge in the graph.

- ✓ Unlike trees, graphs can contain *cycles*

- ✓ In fact, a tree is an *acyclic graph*

Created with OfficeSuite

Graph Implementation

- ✓ We usually represent graphs using a *table* (2d list) where each column and row is associated with a specific vertex.
- ✓ This is called an adjacency matrix.
- ✓ A separate list of vertices shows which vertex name (city,person, etc.) is associated with each index.
- ✓ The values of the 2d list are the weights of the edges between the row vertices and column vertices.
- ✓ If there is not an edge between the two vertices, we use infinity, or None.

Disjoint Sets Data Structure

- ✓ A disjoint-set is a collection ©={S_1, S_2,…, S_k} of distinct dynamic sets.
- ✓ Each set is identified by a member of the set, called *representative*.

Created with OfficeSuite

Disjoint set operations:

- ✓ **MAKE-SET(*x*)**: create a new set with only *x*. assume *x* is not already in some other set.

- ✓ **UNION(*x*,*y*)**: combine the two sets containing *x* and *y* into one new set. A new representative is selected.

- ✓ **FIND-SET(*x*):** return the representative of the set containing *x*.

An Application of Disjoint-Set

- ✓ Determine the connected components of an undirected graph.

CONNECTED-COMPONENTS(G)

for each vertex $v \in$ V[G]

do MAKE-SET(v)

for each edge $(u,v) \in$ E[G]

Created with OfficeSuite

do if FIND-SET(u) ≠ FIND-SET(v)

then UNION(u,v)

SAME-COMPONENT(u,v)

if FIND-SET(u)=FIND-SET(v)

then return TRUE

UNIT - II

Divide and Conquer: The General Method – Defective Chessboard – Binary Search – Finding the Maximum and Minimum – Merge Sort – Quick Sort – Selection - Stassen's Matrix Multiplication.

General Method:

In divide and conquer method, a given problem is,

Created with OfficeSuite

i) Divided into smaller subproblems.
ii) These subproblems are solved independently.
iii) Combining all the solutions of subproblems into a solution of the whole.

If the subproblems are large enough then divide and conquer is reapplied. The generated subproblems are usually of some type as the original problem.

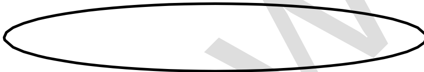
Hence recurssive algorithms are used in divide and conquer strategy.

Created with OfficeSuite

Problem of size N

Subprogram of size

Subprogram of size



Pseudo code Representation of Divide and conquer rule for problem "P"

Solution to

Solution to

Solution to the original problem of

Created with OfficeSuite

```
Algorithm DAndC(P)

{

if small(P) then return S(P)
else{

divide P into smaller instances P1,P2,P3…Pk;

apply DAndC to each of these subprograms; // means DAndC(P1), DAndC(P2)…..
DAndC(Pk)

return combine(DAndC(P1), DAndC(P2)….. DAndC(Pk));
```

$T(n) = T(1)$ if $n=1$

$aT(n/b)+f(n)$ if $n>1$

Time Complexity of DAndC algorithm:

a,b□ contants.

This is called the **general divide and-conquer recurrence.**

Example for GENERAL METHOD:

As an example, let us consider the problem of computing the sum of n numbers $a_0, \dots a_{n-1}$.

If n > 1, we can divide the problem into two instances of the same problem.
They are sum of the first | n/2|numbers

Compute the sum of the 1st [n/2] numbers, and then compute the sum of another n/2 numbers. Combine the

Created with OfficeSuite

answers of two n/2 numbers sum.

i.e.,

$a_0 + . . . + a_{n-1} =(a_0 + ...+ a_{n/2}) + (a_{n/2} + \quad + a_{n-1})$

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)

This is called the general **divide and-conquer recurrence**.

f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. (For the summation example, a = b = 2 and f (n) = 1.

Created with OfficeSuite

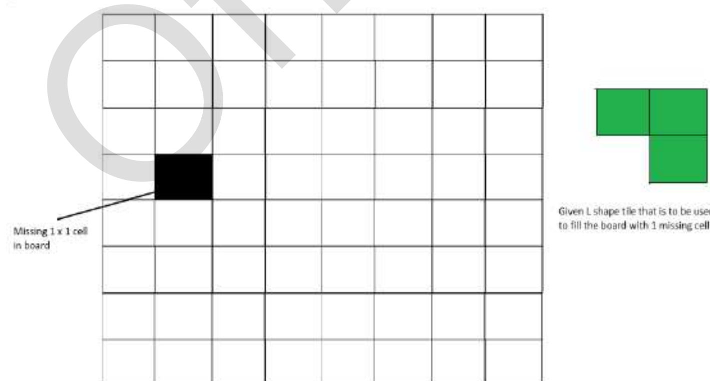
Advantages of DAndC:

The time spent on executing the problem using DAndC is smaller than other method. This technique is ideally suited for parallel computation.

This approach provides an efficient algorithm in computer science.

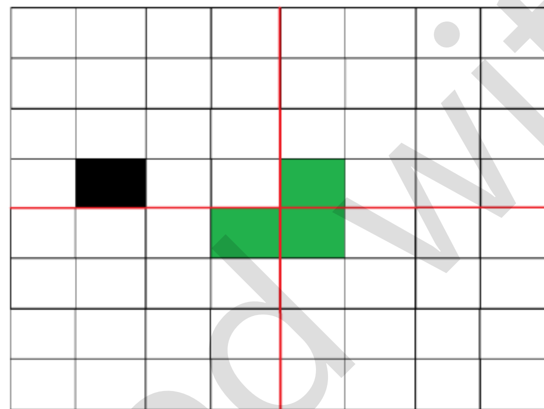
Defective Chess Board Problem using Divide and Conquer algorithm:

Given a n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.



Created with OfficeSuite

The below diagrams show working of above algorithm



After placing first tile

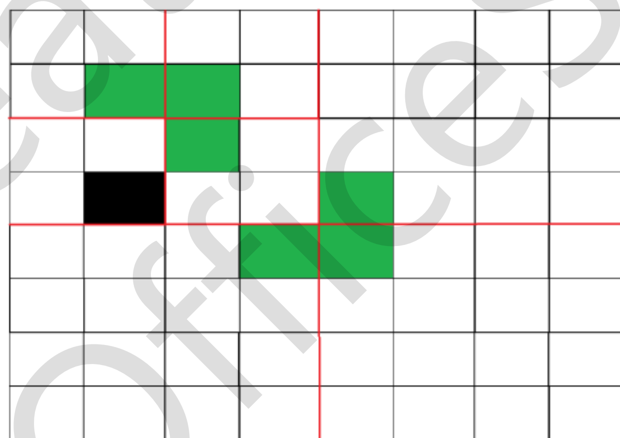
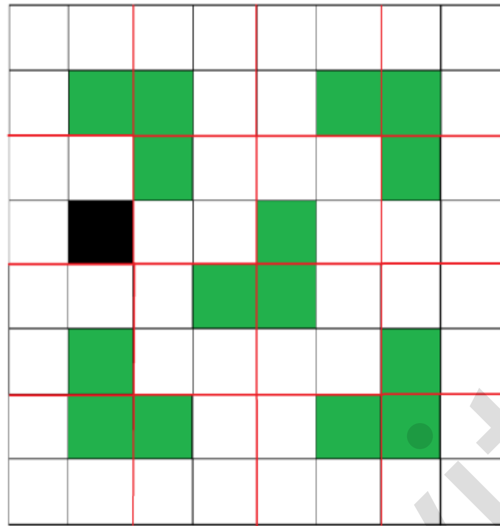


Figure 3: Recurring for first subsquare.

Created with OfficeSuite



Shows first step in all four sub
squares:

Examples:

Input : size = 2 and mark coordinates = (0, 0)

Output :

-1 1

1 1

Coordinate (0, 0) is marked. So, no tile is there. In the remaining three positions,

a tile is placed with its number as 1.

Input : size = 4 and mark coordinates = (0, 0)

Created with OfficeSuite

Output :

-1 3 2 2

3 3 1 2

4 1 1 5

4 4 5 5

Binary Search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :

Created with OfficeSuite

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

Finding the maximum and minimum:

METHOD 1 (Simple Linear Search):

Created with OfficeSuite

Initialize values of min and max as minimum and maximum of the first two elements respectively. Starting from 3rd, compare each element with max and min, and change max and min accordingly (i.e., if the element is smaller than min then change min, else if the element is greater than max then change max, else ignore the element).

Time Complexity: O(n)
In this method, the total number of comparisons is 1 + 2(n-2) in the worst case and 1 + n – 2 in the best case.
In the above implementation, the worst case occurs when elements are sorted in descending order and the best case occurs when elements are sorted in ascending order.

METHOD 2 (Tournament Method):
Divide the array into two parts and compare the maximums and minimums of the two parts to get the maximum and the minimum of the whole array.

METHOD 3(Compare in Pairs):

If n is odd then initialize min and max

Created with OfficeSuite

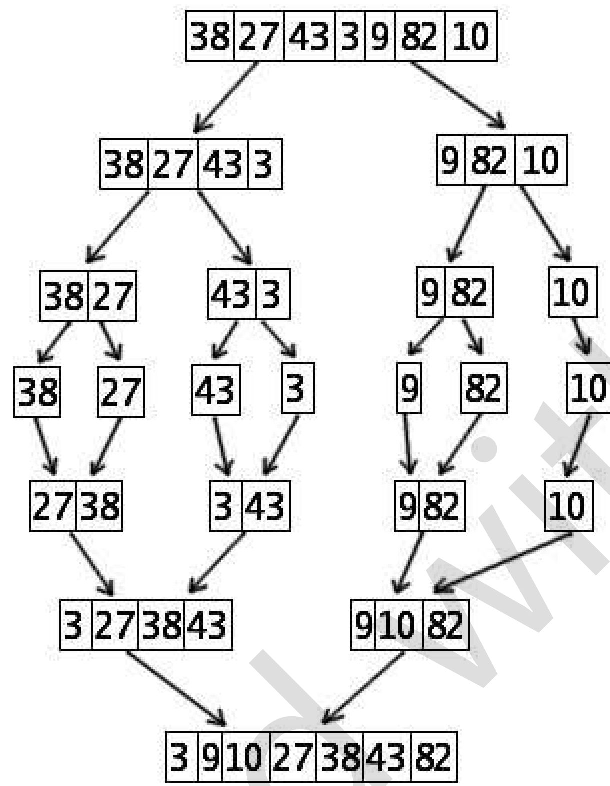
as first element.
If n is even then initialize min and max as minimum and maximum of the first two elements respectively.
For rest of the elements, pick them in pairs and compare their maximum and minimum with max and min respectively.

Created with OfficeSuite

Merge Sort:

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of $O(n \log n)$. Elementary implementations of the merge sort make use of two arrays - one for each half of the data set. The following image depicts the complete procedure of merge sort.

Created with OfficeSuite



Advantages of Merge Sort:

1. Marginally faster than the heap sort for larger sets
2. Merge Sort always does lesser number of comparisons than Quick Sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.
3. Merge sort is often the best choice for sorting a linked list because the slow random- access performance of a linked list makes some other

Created with OfficeSuite

algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

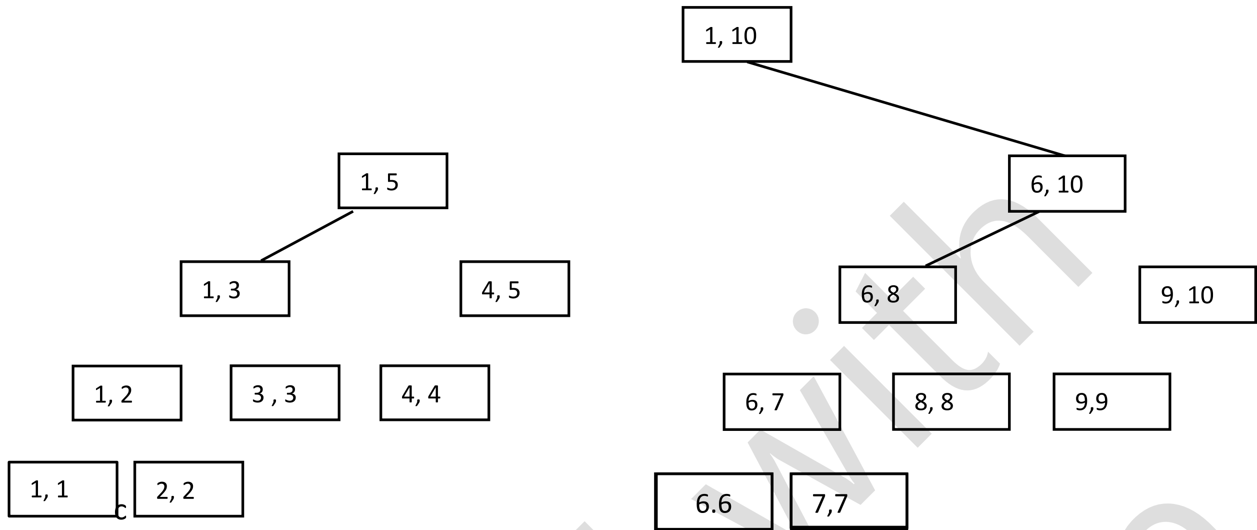
Tree call of Merge sort

Consider a example: (From text book) A[1:10]={310,285,179,652,351,423,861,254,450,520}

6, 7

1, 1 c 2, 2

Created with OfficeSuite



Created with OfficeSuite

Computing Time for Merge sort:

The time for the merging operation in proportional to n, then computing time for merge sort is described by using

recurrence relation.

$$T(n)= \begin{cases} a & \text{if } n=1; \\ 2T(n/2)+ cn & \text{if } n>1 \end{cases}$$

Quick Sort

Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is O (n log n).

- Auxiliary space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly, especially when it comes to large data sets.

Created with OfficeSuite

- Its worst case has a time complexity of O (n) which can prove very fatal for large data sets. Competitive sorting algorithms.

Name	Time Complexity			Space Complexity
	Best case	**Average Case**	**Worst Case**	
Bubble	$O(n)$	-	$O(n^2)$	$O(n)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$
Quick	$O(\log n)$	$O(n \log n)$	$O(n2)$	$O(n + \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(2n)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Created with OfficeSuite

Comparison between Merge and Quick Sort:

- Both follows Divide and Conquer rule.
- Statistically both merge sort and quick sort have the same average

case time i.e., O(n log n).

- Merge Sort Requires additional memory. The pros of merge sort are: it is a stable sort, and there is no worst case (means average case and worst case time complexity is same).
- Quick sort is often implemented in place thus saving the performance and memory by not creating extra storage space.
- But in Quick sort, the performance falls on already sorted/almost sorted list if the pivot is not randomized. Thus why the worst case time is $O(n^2)$.

Randomized Sorting Algorithm: (Random quick sort)

- While sorting the array a[p:q] instead of picking a[m], pick a random element (from among a[p], a[p+1], a[p+2]---a[q]) as the partition elements.
- The resultant randomized algorithm works on any input and

Created with OfficeSuite

runs in an expected $O(n \log n)$ times.

Created with OfficeSuite

Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Strassen´s Matrix Multiplication:

Let A and B be two n×n Matrices. The product matrix C=AB is also a n×n matrix whose i, j^{th} element is formed by taking elements in the ith row of A and j^{th} column of B and multiplying them to get

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

Here $1\leq i$ & $j \leq n$ means i and j are in between 1 and n.

To compute C(i, j) using this formula, we need n

multiplications.

The divide and conquer strategy suggests another way to compute the product of two $n \times n$ matrices.

For Simplicity assume n is a power of 2 that is $n=2^k$ Here k□ any nonnegative integer.

If n is not power of two then enough rows and columns of zeros can be added to both A and B, so that resulting dimensions are a power of two.

Let A and B be two $n \times n$ Matrices. Imagine that A & B are each partitioned into four square sub matrices. Each sub matrix having dimensions $n/2 \times n/2$.

The product of AB can be computed by using

Created with OfficeSuite

previous formula. If AB is product of 2×2 matrices then

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

$$T(n) = \begin{cases} b & \text{if } n \le 2; \\ 7T(n/2) + cn^2 & \text{if } n > 2 \end{cases}$$

Created with OfficeSuite

UNIT-3

UNIT - III

The Greedy Method: General Method - Container Loading - Knapsack Problem - Tree Vertex Splitting – Job Sequencing With Deadlines - Minimum Cost Spanning Trees – Optimal Storage on Tapes–Optimal Merge Patterns-Single Source Shortest Paths.

GREEDY METHOD

Greedy Method:

The greedy method is perhaps (maybe or possible) the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

Feasible solution:-

Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

Optimal solution:

To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one

Created with OfficeSuite

input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Greedy algorithms neither postpone nor revise the decisions (ie., no back tracking).
Example: Kruskal's minimal spanning tree. Select an edge from a sorted list, check, decide, and never visit it again.

Application of Greedy Method:

- Job sequencing with deadline
- 0/1 knapsack problem
- Minimum cost spanning trees
- Single source shortest path problem

.

Algorithm for Greedy method

Algorithm Greedy(a,n)
//a[1:n] contains the n inputs.
{
Soluti
on

Created with OfficeSuite

```
:=0;
For
i=1 to
n do
{
X:=select(a);
If
Feasible(soluti
on, x) then
Solution
:=Union(soluti
on,x);
}
Return solution;
}
```

Selection Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.

Feasible Boolean-valued function that determines whether x can be included into the solution vector.

Created with OfficeSuite

Union □ function that combines x with solution and updates the objective function.

Container Loading:

Large ship to be loaded with cargo . • All containers are of the same size but may be of different weights. • Container i has weight wi . • The capacity of the ship is C. • Load the ship with maximum number of containers without exceeding the cargo weight capacity. • Find values $x_i \in \{0, 1\}$ such that • And the optimum function is maximized. • Every set of xi 's that satisfy the constraints is a feasible solution. • Every feasible solution that maximizes is an optimal solution.

• Ship may be loaded in stages. • Greedy criterion: From the remaining containers, select the one with least weight.

Example:

n = 8 [w1 , w2 , w3 , w4 , w5 , w6 , w7 , w8] = [100, 200, 50, 90, 150, 50, 20, 80] C = 400 [x1 , x2 , x3 , x4 , x5 , x6 , x7 , x8] = [1, 0, 1, 1, 0, 1, 1, 1]

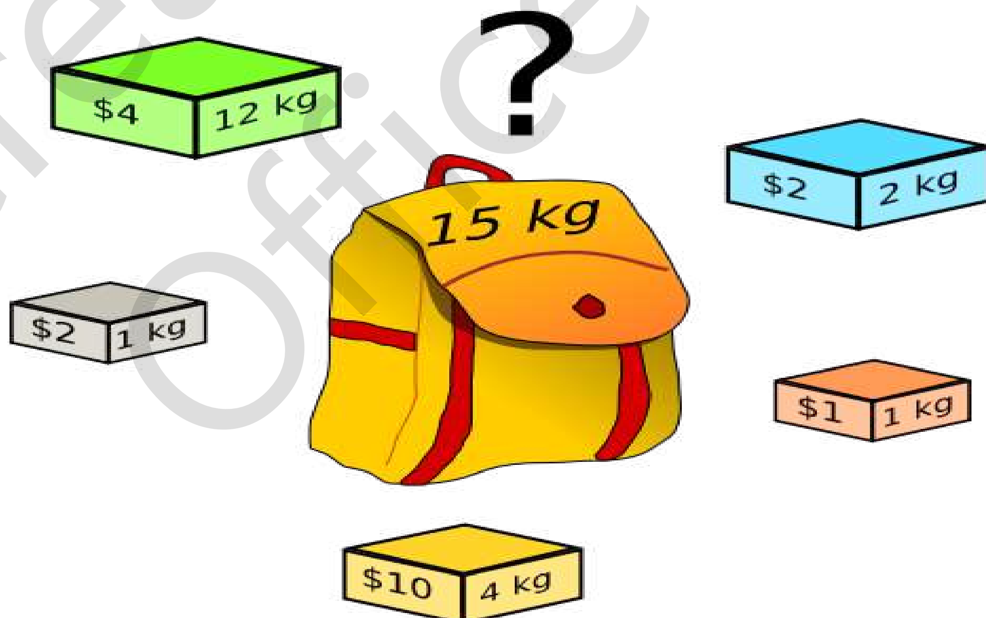
Created with OfficeSuite

Algorithm

ContainerLoading(c, capacity, numberofContainers, x) // set x[i] = 1 if and only if container c[i], i ≥ 1 is loaded. { // sort into increasing order of weights. Sort(C, numberofContainers); n = numberofContainers; for i = 1 to n do x[i] = 0; i = 1; while ((i ≤ n) && (c[i].weight ≤ capacity)) { x[c[i].id] = 1; capacity = capacity – c[i].weight; i++; } }

Knapsack Problem:

• Given a set of items, each with a weight and a profit, determine the number of each item to include in a collection so that the total weight is less than or



equal to a given limit and the total profit is as large as possible. • Items are divisible; you can take any

Created with OfficeSuite

fraction of an item. • And it is solved using greedy method.

Created with OfficeSuite

• Given n objects and a knapsack or bag. • wi → weight of object i. • m → knapsack capacity. • If a fraction xi , 0 ≤ xi ≤1 of object i is placed into the knapsack, then a profit of pixi is earned. • Objective is to fill the knapsack that maximizes the total profit earned. • Problem can be stated as • A feasible solution is any set (x1 …. xn) satisfying equations ②and ③. • An optimal solution is a feasible solution for which equation ① is maximized. 6 subject to wixi ≤ m − − − − − − ② 1≤i ≤n 0 ≤ xi ≤ 1, 1 ≤ i ≤ n − − − − −③ maximize pi 1 ≤i ≤n xi − − − − − − ①.

<u>Example:</u>

n = 3, m = 20 (x1 , x2 , x3) Σwixi Σpixi 1. (1/2, 1/3, 1/4) 16.5 24.25 2. (1, 2/15, 0) 20 28.2 3. (0, 2/3, 1) 20 31 4. (0, 1, 1/2) 20 31.5 5. (2/3, 8/15, 0) 20 29.5 6. (5/6, 1/3, 0) 20 28.8 Among all the feasible solutions yields the maximum profit Weight wi 18 15 10 Profits pi 25 24 15

<u>The greedy algorithm:</u>

Step 1: Sort pi /wi into nonincreasing order.

Step 2: Put the objects into the knapsack according

Created with OfficeSuite

to the sorted sequence as possible as we can.

e. g. n = 3, M = 20 (w1 , w2 , w3) = (18, 15, 10) (p1 , p2 , p3) = (25, 24, 15) Sol: p1 /w1 = 25/18 = 1.39 p2 /w2 = 24/15 = 1.6 p3 /w3 = 15/10 = 1.5 Optimal solution: x1 = 0, x2 = 1, x3 = 1/2 Weight wi 15 10 18 Profits pi 24 15 25

Tree Vertex Splitting:

• A vertex with in-degree zero is called a source vertex • A vertex with out-degree zero is called a sink vertex • Let T/X be the forest that results when each vertex u is split into two nodes u i and u o such that all the edges ⟨u, j⟩ ∈ E (⟨j, u⟩ ∈ E) are replaced by the edges of the form ⟨u o , j⟩ (⟨j, u i ⟩) • A greedy approach to solve this problem is to compute for each node u ∈ V, the maximum delay d(u) from u to any other node in its subtree. • If u has a parent v such that d(u) + w(v, u) > δ, then the node u gets split and d(u) is set to 0. where C(u) is the set of all children of u. d u = max $v \in C(u)$ { $d\ v + W(u, v)$

Algorithm:

TVS(T, δ)

{ if (T ≠ 0) then

Created with OfficeSuite

{ d[T] = 0; for each child v to T do

{ TVS(v, δ); d[T] = max{d[T], d[v]+w[T,v]};

}

if ((T is not the root) and (d[T] + w(parent(t), T) > δ)) then

{ write(T); d[T] = 0; } } }

<u>Job sequencing with deadlines</u>:

There are n jobs to be processed on a machine. • Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$. • P_i is earned if and only if the job is completed by its deadline. • The job is completed if it is processed on a machine for unit time. • Only one machine is available for processing jobs. • Only one job is processed at a time on the machine. • A feasible solution is a subset of jobs J such that each job is completed by its deadline. • An optimal solution is a feasible solution with maximum profit value

General method of job sequencing algorithm:

Algorithm GreedyJob(d, J, n)

{ J := {1}; for i := 2 to n

do { if (all jobs in J $\cup$ {i} can be completed by their deadlines) then J := J $\cup$ {i};

} }

Created with OfficeSuite

Example 1: Let n = 4, maximum deadline dmax = 2 (p1 , p2 , p3 , p4) = (100,10,15,27) (d1 , d2 , d3 , d4) = (2,1,2,1)

27 + 100 = 127

<u>Minimum Cost Spanning Trees</u>:

Given an undirected and connected graph G = (V, E), a spanning tree of the graph G is a subset of graph G, which has all the vertices connected by minimum number of edges. • The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. • A Minimum Spanning Tree (MST) is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. • There also can be many minimum spanning trees. • There are two famous algorithms for finding the Minimum Spanning Tree:

¬ Prim's Algorithm

¬ Kruskal's Algorithm

<u>MST – Prim's Algorithm:</u>

Prim's Algorithm is used to find the minimum

Created with OfficeSuite

spanning tree from a graph. • Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized. • Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. • The edges with the minimal weights causing no cycles in the graph are selected.

• **Algorithm steps**: Step 1: Select a starting vertex. Step 2: Repeat Steps 3 and 4 until there are vertices not in the tree. Step 3: Select an edge e connecting the tree vertex and the vertex that is not in the tree has minimum weight. Step 4: Add the selected edge and the vertex to the minimum spanning tree T Step 5: Exit

<u>Optimal Storage on tapes</u>:

• n programs are to be stored on a computer tape of length l. • Associated with each program i is a length l_i , $1 \leq i \leq n$. • If the programs are stored in the order $I = i_1 , i_2 , \ldots.. i_n$, the time t_j needed to retrieve the program i_j is • If all the programs are retrieved equally often, then the Mean Retrieval Time (MRT) is • Minimizing the MRT is equivalent to minimizing

Example: n = 3, (l_1 , l_2 , l_3) = (5, 10, 3) n! = 6 possible ordering Ordering I

d(I) 1, 2, 3 5+5+10+5+10+3 = 38

Created with OfficeSuite

1, 3, 2 5+5+3+5+3+5+10 = 31

2, 1, 3 10+10+5+10+5+3 = 43

2, 3, 1 10+10+3+10+3+5 = 41

3, 1, 2 3+3+5+3+5+10 = 29

3, 2, 1 3+3+10+3+10+5 = 34 Optimal ordering is 3, 1, 2 Thus the greedy method implies to store the programs in nondecreasing order of their length.

Optimal Merge patterns:

Merge a set of sorted files of different length into a single sorted file.

- We need to find an optimal solution, where the resultant file will be generated in minimum time.
- If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns.
- As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.
- To merge a m-record file and a n-record file requires possibly m + n record moves

Created with OfficeSuite

• Merge the two smallest files together at each step.

• Two-way merge patterns can be represented by binary merge trees.

• Initially, each element is considered as a single node binary tree.

• The algorithm has as input a list list of n trees.

• Each node in a tree has three fields, lchild, rchild and weight.

• Initially, each tree in list has exactly one node and has lchild and rchild fields zero whereas weight is the length of one of the n files to be merged.

Algorithm

```
Tree(n) { for i = 1 to n-1 do

{ pt = new treenode; pt→lchild = Least(list);

pt→rchild = Least(list);

 pt→weight = pt→lchild→weight +
pt→lchild→weight; insert(list,pt); }

 return Least(list); }

 treenode = record

{ treenode *lchild;

 treenode *rchild;

integer weight; }
```

Created with OfficeSuite

Function Tree uses two functions:

Least(list) and Insert(list, t).

• Least(list) finds a tree in list whose root has least weight and returns a pointer to the tree. This tree is removed from list.

• Insert(list, t) inserts the tree with root t into list

<u>Single-source shortest path :</u>

• Given a edge-weighted graph G = (V, E) and a vertex v ∈ V, find the shortest weighted path from v to every other vertex in V.

• Dijkstra's Algorithm is a greedy algorithm for solving the single-source shortest-path problem on an edge-weighted graph in which all the weights are non-negative.

• It finds the shortest paths from some initial vertex, say v, to all the other vertices one-by-one.

• The paths are discovered in the order of their weighted lengths, starting with the shortest, and proceeding to the longest.

• For each vertex v, Dijkstra's algorithm keeps track of three pieces of information, kv , dv and pv .

• The Boolean valued flag kv indicates that the shortest path to vertex v. Initially, kv = false for all v ∈ V.

Created with OfficeSuite

• The quantity dv is the length of the shortest known path from v0 to v. When the algorithm begins, no shortest paths are known. The distance dv , is a tentative distance

• During the course of the algorithm candidate paths are examined and the tentative distances are modified.

• Initially dv = ∞ for all v ∈ V such that v ≠ v0 , while dv0 = 0. • The predecessor of the vertex v on the shortest path from v0 to v is pv . Initially, pv is unknown for all v ∈ V.

• The following steps are performed in each pass: 1. From the set of vertices for with kv = false, select the vertex v having the smallest tentative distance dv . 2. Set kv ← true. 3. For each vertex w adjacent to v for which kv ≠ true, test whether the tentative distance dv is greater than dv + C(v,w). If it is, set dw ← dv + C(v,w) and set pw ← v.

• In each pass exactly one vertex has its kv set to true. The algorithm terminates after |V| passes are completed at which time all the shortest paths are known.

Initially:

S = {1};

Created with OfficeSuite

D[2] = 10;

D[3] = ∞; D[4] = 30; D[5] = 100

Iteration 1

Select w = 2,

so that S = {1, 2}

D[3] = min(∞, D[2] + C[2, 3]) = 60

D[4] = min(30, D[2] + C[2, 4]) = 30

D[5] = min(100, D[2] + C[2, 5]) = 100

Iteration 2

Select w = 4, so that S = {1, 2, 4}

D[3] = min(60, D[4] + C[4, 3]) = 50

D[5] = min(100, D[4] + C[4, 5]) = 90

Iteration 3

Select w = 3, so that S = {1, 2, 4, 3}

D[5] = min(90, D[3] + C[3, 5]) = 60

Iteration 4

Select w = 5, so that S = {1, 2, 4, 3, 5}

D[2] = 10; D[3] = 50; D[4] = 30; D[5] = 6

Created with OfficeSuite

UNIT - IV

Dynamic Programming: The General Method – Multistage Graphs – All-Pairs Shortest Paths – Single-Source Shortest Paths - OptimalBinary Search Trees - String Editing - 0/1 Knapsack - Reliability Design - The Traveling Salesperson Problem - Flow Shop Scheduling. Basic Traversal and Search Techniques: Techniques for Binary Trees – Techniques for Graphs–Connected Components and Spanning Trees– Bi-connected Components and DFS.

DYNAMIC PROGRAMMING:-

THE GENERAL METHOD:

Dynamic programming is an algorithm design method that can be used when the solution of a problem can be viewed as the result of a sequence of decisions.

An optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision.

Created with OfficeSuite

This is true for all problems solvable by the greedy method.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds.
- Set up the dynamic-programming recurrence equations.
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

MULTI STAGE GRAPHS:

A multistage graph G=(V,E)is a directed graph in which the vertices are partitioned into K>=2 disjoint sets Vi,1<=i<=k.

In addition,if<u,v>is an edge in E,then u E Vi and v E Vi+1 for some i,1<=i<k.

The multistage graph problem can also be solved using the backward approach.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k-2

Created with OfficeSuite

decisions.

ALGORITHM:

Algorithm Fgraph(G,k,n,p)

```
Cost of(i,j).p[1:k]is a minimum cost path
{
Cost[n]:=0.0;
For j:n-1 to step-1 do
{
Let r be a vertex such that(j,r)is an edge of G
c[j,r]+cost[r]is minimum;
cost[j]:=c[j,r]+cost[r];
d[j]:r:
}
p[1]:=1;
p[k]:=n;
for j:=2 to k-1 do p[j]:d[p[j-1]];
}
```

Algorithm Bgraph(G,k,n,p)

Created with OfficeSuite

```
Fgraph
{
Bcost[1]:=0.0;
for j:=2 to n do
{
Bcost [j].
Let r be such that(r,j)is an edge of G
Bcost[r]+c[r,j]is minimum;
Bcost[j]:=Bcost[r]+c[r,j];
D[j]:=r;
}
p[1]:=1;
p[k]:=n;
for j:=k-1 to 2 do p[j]:=d[p[j+1]];
}
```

ALL –PAIRS SHORTEST PATHS:

In the all pairs of shortest path problem,we are to find a shortest path between every pair of vertices in a directed graph G.

Created with OfficeSuite

The two paths are the same when G is undirected.

When no edge as a negative length,the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times,once with each of the n vertices as the source vertex.

AK(i,j)={min{A^{k-1}(i,k)+A^{k-1}(k,j)},c(i,j)}

Algorithm All Paths(Cost,A,n)

cost[i,j]=0.0,for 1<=i<=n

```
{
for i:=1 to n do
for j:=1 to n do
A[i,j]:=cost[i,j];
for k:=1 to n do
for i:=1 to n do
for j:=1 to n do
A[i,j]:=min(A[i,j],A[i,k]+A[k,j]);
}
```

SINGLE-SOURCE SHORTEST PATHS:

Created with OfficeSuite

When there are no cycles of negative length ,there is shortest path between any two vertices of an n-vertex graph that has at most n-1 edges on it.

The maximum number of edges on a cycle-free shortest path algorithm from source vertex to all remaining vertices in the graph.

When some of all of the edges of the directed graph G may have negative length.

When negative edge lengths are permitted,we require that the graph have no cycles of negative length.

$dist^{k}[u]=min\{dist^{k-1}[u],min\{dist^{k-1}[i]+cost[I,u]\}\}$

Algorithm BellmanFord(v,cost,dist,n)

{

for i:=1 to n do

dist[i]:=cost[v,i];

for k:=2 to n-1 do

for each you such that u≠v and u has at last one incoming edge do

for each(i,u)in the graph do

if dist[u]>dist[i]+cost[i,u]then

Created with OfficeSuite

dist[u]:=dist[i]+cost[i,u];

}

OPTIMAL BINARY SEARCH TREES:

Huffman coding tree solved by a greedy algorithm has a limitation of having the data only at the leaves and it must not preserve the property that all nodes to the left of the root have keys,which are less etc.

Construction of an optimal binary search tree is harder,because the data is not constrained to appear only at the leaves,and also because the tree must satisify the binary search tree property and it must preserve the property that all nodes to the left of the root have keys,which are less.

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality.

~nPi(1+di)

i~1

The expected contribution for the internal node for 'ai' is:

p(i)*level(ai).

Created with OfficeSuite

Unsuccessful search terminate with I=0.Hence the cost contribution for this node is:

Q(i)*level(Ei)-1)

The expected cost of binary search tree is:

$^{n}\sim P(i)*level(ai)+^{n}\sim Q(i)*level((Ei)-1)$

The total time to evaluate all the c(i,j)'s and r(i,j)'s is therefore:

$\sim(nm-m^{2})=0(n^{3})1<m<n$

Given a fixed set of identifiers,we wish to create a binary search tree organization.

We may expect different binary search trees for the same identifier set to have different performance characteristics.

$cost(L)=^{k}.P(i)*level(ai)+^{k}.Q(i)*level(Ei)-1,$

i-1 i-0

$cost(ft)=^{n}.P(i)*level(ai)+^{n}.Q(i)*level(Ei)-1,$

i-k i-k

STRING EDITING:

We are given two strings X=x1,x2,…,xn and Y=y1,y2,…,ym,where x;,1<=i<n,and

Created with OfficeSuite

yj,1<=j<=m,are members of a finite set of symbols known as the alphabet.

The cost of a sequence of operation is the sum of the costs of the individual operation in the sequence.

The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X and Y.

A solution to the string editing problem consists of a sequence of decisions,one for each edit operation.

Let D(x1)be the cost of deleting the symbol xi from X,I(y_3)be the cost of inserting the symbol yj into X,and C(xi,yj)be the cost of changing the symbol xi of X into yj.

cost(1,1)=min{cost(0,1)+D(x1),cost(0,0)+C(x1,y2),cost(1,0)+I(y1)}

=min{2,2,2}=2

Next is computed cost(1,2)

Cost(1,2)=min{cost(0,2)+D(x1),cost(0,1)+C(x1,y2),cost(1,1)+I(y_2)}

=min{3,1,3}=1

Created with OfficeSuite

O/I KNAPSACK:

We are given n objects and a knapsack.

Each object i has a positive weight wi and a positive value vi.

The knapsack can carry a weight not exceeding W.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x1,x2,…xn.

A decision on variables xi involves determining which of the values 0 or 1 is to be assigned to it.

$Fn(m)=max\{fn-1(m),fn-1(m-wn)+pn\}$

For arbitrary fi(y),i>0,this equation generalizes to:

$Fi(y)=max\{fi-1(y),fi-1(y-wi)+pi\}$

$Si1=\{(P,W)/(P-pi,W-wi)eS^{i}\}$

The strategy we have come up with can also be derived by attempting to solve the knapsack problem via a systematic examination of the up to 2^{n} possibilities for x1,x2,…xn.

Algorithm DKP(p,w,n,m)

{

Created with OfficeSuite

S^0:={(0,0)};

For i:=1 to n-1 do

{

S^{i-1}:={(P,W)\(P-pi,W-wi)∈S^{i-1} and W<=m};

S^1:=MergePurge(S^{i-1},S1^{i-1});

}

(PX,WX):=last pair in S^{n-1});

(PY,WY):=(p'+p_n,W'+w_n)where W' is the largest W in any pair in S^{n-1} such that W+w_n<=m;

If(PX>PY) then x_n:=0;

Else x_n:=1;

TraceBackFor($x_{n-1,\ldots,}x_1$);

}

RELIABILITY DESIGN:

The problem is to design a system that is compsed of several devices connected in series.

Let ri be the reliability of device Di then the reliability of the entire system is fT ri.

Even if the individual devices are very reliable,the

Created with OfficeSuite

reliability of the system may not be very good.

If stage i contains mi copies of device Di.Then the probability that all mi have a malfunction is(1-ri)mi.Hence the reliability of stage i becomes $1-(1-r)^{mi}._{i}$

Our problem is to use device duplication.This maximization is to be carried out under a cost constraint.

Solve:

Maximize~qi(mi~1-<-i<n

Subject to~Ci mi<C 1-<-i<n

mi>=1 and integer,1<=i<=n

The optimal solution m1,m2……mn is the result of a sequence of decisions,one decision for each mi.

Subject of the constraints:

Cj mj~x and 1<=mj<=uj,1<=j<=i

1<=i<=i

The last decision made requires one to choose mn from{1,2,3,….un}.once a value of mn has been chosen,the remaining decisions must be such as to use the remaining funds C-Cn mn in an optimal way.

Created with OfficeSuite

The principles of optimality holds on

fn~C~~max{0n(mn)fn_1(C-Cn mn)}1<mn<un

The general equation

fn(x)=max{ci(mi)fi-1(x-Ci mi)}1<mi<ui

clearly,f0(x)=1 for all x,0<=x<=C and f(x)=-00 for all x<0.Let S^i consist of tuples of the form(f,x),where f=fi(x).

There is atmost one tuple for each different ‘x’,that result from a sequence of decisions on m1,m2,…..mn.The dominance rule(f1,x1)dominate(f2,x2)if f1>=f2 and x1>=x2.Hence,dominated tuples can be discarded from S^i.

THE TRAVELING SALESPERSON PROBLEM:

Let G=(V,E) be a directed graph with edge costs Cij.The variable cij is defined such that cij>0 for all I and j and cij=a if<i,j>oE.

A tour of G is a directed simple cycle that includes every vertex in V.The cost of tour is the sum of the cost of edges on the tour.

The traveling sales person problem is to find a tour

Created with OfficeSuite

of minimum cost.The tour is to be a simple path that starts and ends at vertex 1.

Hence,this problem can be regarded as a traveling salesperson problem on an n vertex graph with edge cost cij's being the changeover cost from commodity i to commodity j.

Principles of optimality

g(1,V-{1})=2~k~n~c1k~g~k,V~~1,k~~

Generalizing equation1

g(i,s)=min{cij jEs

The equation can be solved for g(1,V-1})if we know g(k,V-{1,k})for all choices of k.

Complexity Analysis:

For each value of [S] there + g,i,s-,j are n-1 choices for i.The number of distinct sets S of size K not including 1 and I is I k~~n-2~

The most serious drawback of this dynamic programming solution is the space needed,which is $O(n\ 2^n)$.This is too large even for modest values of n.

FLOW SHOP SCHEDULING:

Hence,it suffices to generate any schedule for which

Created with OfficeSuite

holds for every pair of adjacent jobs.

If min{a1,a2,…,an,b1,b2,…,bn} is bj,then job j should be the last job in an optimal schedule.This enables us to make a decision as to the positioning of one of the n jobs.

1. Sort all the ai’s and bj’s into nondecreasing order.
2. If the next number in the sequence is aj and job j hasn’t yet been scheduled,schedule job j at the leftmost available spot.
3. If the next number is bj and job j hasn’t yet been scheduled,schedule job j at the rightmost available spot.If j has already been scheduled,go to the next number in the sequence.

BASIC TRAVERSAL AND SEARCH TECHNIQUES:-

TECHNIQUES FOR BINARY TREES:

The solution to many problems involves the manipulation of binary trees,trees,or graphs.Often this manipulation requires us to determine a vertex(node)or a subset of vertices in the given data object that satisfies a given property.

This algorithm is not a traversal algorithm as it does

Created with Office Suite

not examine every vertex in the search tree.There are many operations that we want to perform on binary trees.

A traversal produces a linear order for the information in a tree.This linear order may be familiar and useful.

Six combination of traversal

LDR,LRD,DLR,DRL,RDL, and RLD

Inorder,preorder and postorder.

Algorithm InOrder(t)

```
{
If t ≠ 0 then
{
InOrder(t->l child);
Visit(t);
InOrder(t->r child);
}
}
```

Algorithm preorder

Created with OfficeSuite

```
{
if t≠0 then
{
Visit(t);
PreOrder(t->t child);
PreOrder(t->r child);
}
}
```

Algorithm PostOrder(t)

```
{
If t≠0 then
{
PostOrder(t->t child);
PostOrder(t->r child);
Visit(t);
}
}
```

TECHNIQUES FOR GRAPHS:

Created with OfficeSuite

A fundamental problem concerning graphs is the reachability problem.

In its simplest form it requires us to determine whether there exists a path in the given graph G=(V,E) such that this path starts at vertex v and ends at vertex u.

BREADTH FIRST SEARCH AND TRAVERSAL:

In breadth first search we start at a vertex v and mark it as having been reached.The vertex v is at this time said to be unexplored.

The list of unexplored vertices operates as a queue and can be represented using any of the standard queues representation.

Algorithm BFS(e)

```
{
U:=v;
Visited[v]:=1;
repeat
{
For all vertices w adjacent from u do
{
```

Created with OfficeSuite

```
If(visited[w]=0)then
{
Add w to q;
visited[w]:=1;
}
}
If q is empty then return;
Delete the next element,u,from q;
}
until(false);
}
```

DEPTH FIRST SEARCH AND TRAVERSAL:

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex u begins.

When this new vertex has been explored,the exploration of v continuous.The search terminates when all reached vertices have been fully explored.

Algorithm DFS(e)

Created with OfficeSuite

```
{
Visited[v]:=1;
For each vertex w adjacent from v do
{
If(visited[w]=0)then DFS(w);
}
}
```

CONNECTED AND SPANNING TREES:

If G is a connected undirected graph,then all vertices of G will get visited on the first call to BFS.If G is not connected,then at least two calls to BFS will be needed.

BFS can be used to determine whether G is connected.All newly visited vertices on a call to BFS from BFT represent the vertices in a connected of graph can be obtained using BFT.

Then the subgraph formed by the vertices on this list make up a connected component.Hence,if adjacency lists are used,a breadth first traversal will obtain the connected components in $\theta(n+e)$time.

BFT can also be used to obtain the reflexive

Created with OfficeSuite

transitive closure matrix of an undirected graph G.If A*(i,j)=1 if either i=j or i≠j and i and j are in the same connected components.

As a final application of breadth first search,consider the problem of obtaining a spanning tree for an undirected graph G.The graph G has a spanning tree if G is connected.

BFS easily determines the existence of a spanning tree.

Spanning trees obtained using a breadth first search are called breadth first spanning trees.

IICONNECTED COMPONENTS AND DFS:

The “graph” we always mean an undirected graph.

A vertex v in a connected graph G is an articulation point if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.

A graph G is biconnected if and only if it contains no articulation points.The graph of is not biconnected.

The presence of articulation points in a connected graph can be an undesirable feature in my cases.Once it has been termined that a connected graph G is not

Created with OfficeSuite

biconnected,it may be desired to determine a set of edges the graph is biconnected.

Depth first spanning trees have a property that is very useful in identifying articulation points and biconnected components.

define L[u],

L[u]=min{dfn[u],min{L[w]\w is a child of u},min{dfn[w]!(u,w)is a back edge}}

L[u] can be easily computed if the vertices of the depth first spanning tree are visited in postorder.

Thus,to determine the articulation points,it is necessary to perform a depth first search of the graph G and visit the nodes in the resulting depth first spanning tree in postorder.

Algorithm

for each articulation point a do

{

Let B1,B2,….,Bk be the biconnected components containing vertex a;

Let vi,vi≠a,be a vertex in Bi,1<=i<=k;

Add to G the edges(vi,vi+1),1<=i<k;

Created with OfficeSuite

}

UNIT - V

Backtracking: The General Method – The 8-Queens Problem – Sum of Subsets– Graph Coloring – Hamiltonian Cycles – Knapsack Problem Branch and Bound: Least Cost search - 0/1 Knapsack Problem

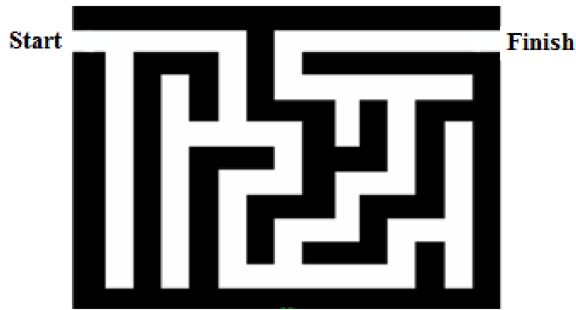
Backtracking (General method)

- ✓ Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.
- ✓ Suppose you have to make a series of decisions among various choices, where
- ✓ You don't have enough information to know what to choose
- ✓ Each decision leads to a new set of choices.
- ✓ Some sequence of choices (more than one choices) may be a solution to your problem.

Backtracking is a methodical (Logical) way of trying out various sequences of decisions, until you find one that "works"

Example : Maze (a tour puzzle)

Created with Office Suite



Given a maze, find a path from start to finish.

- In maze, at each intersection, you have to decide between 3 or fewer choices:
  - ➢ Go straight
  - ➢ Go left
  - ➢ Go right
  - ➢ You don’t have enough information to choose correctly
  - ➢ Each choice leads to another set of choices.
  - ➢ One or more sequences of choices may or may not lead to a solution.
  - ➢ Many types of maze problem can be solved with backtracking.

Example:

- o Sorting the array of integers in a[1:n] is a problem whose solution is expressible by an n-tuple xi is the index in ‘a’ of the it smallest element.
- o The criterion function ‘P’ is the inequality a[xi]≤ a[xi+1] for 1≤ i ≤ n Si is finite and includes the integers 1 through n.

Created with OfficeSuite

- M_i size of set S_i $m=m_1m_2m_3---m_n$ n tuples that possible candidates for satisfying the function P.
- With brute force approach would be to form all these n-tuples, evaluate (judge) each one with P and save those which yield the optimum.
- By using backtrack algorithm; yield the same answer with far fewer than ‘m’ trails.
- Many of the problems we solve using backtracking requires that all the solutions satisfy a complex set of constraints.
- For any problem these constraints can be divided into two categories:

  - Explicit constraints.
  - Implicit constraints.

Explicit constraints: Explicit constraints are rules that restrict each $\mathbf{x_i}$ to take on values only from a given set.
Example:
$\mathbf{x_i \geq 0}$ or s_i={all non negative real numbers}
$X_i=0$ or 1 or $S_i=\{0, 1\}$
$l_i \leq x_i \leq u_i$ or $s_i=\{a: l_i \leq a \leq u_i\}$

- The explicit constraint depends on the particular instance I of the problem being solved.
- All tuples that satisfy the explicit constraints define a possible solution space for I.

Implicit Constraints:

- The implicit constraints are rules that determine

Created with OfficeSuite

which of the tuples in the solution space of I satisfy the criterion function.

- Thus implicit constraints describe the way in which the **Xi** must relate to each other.

Applications of Backtracking:

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

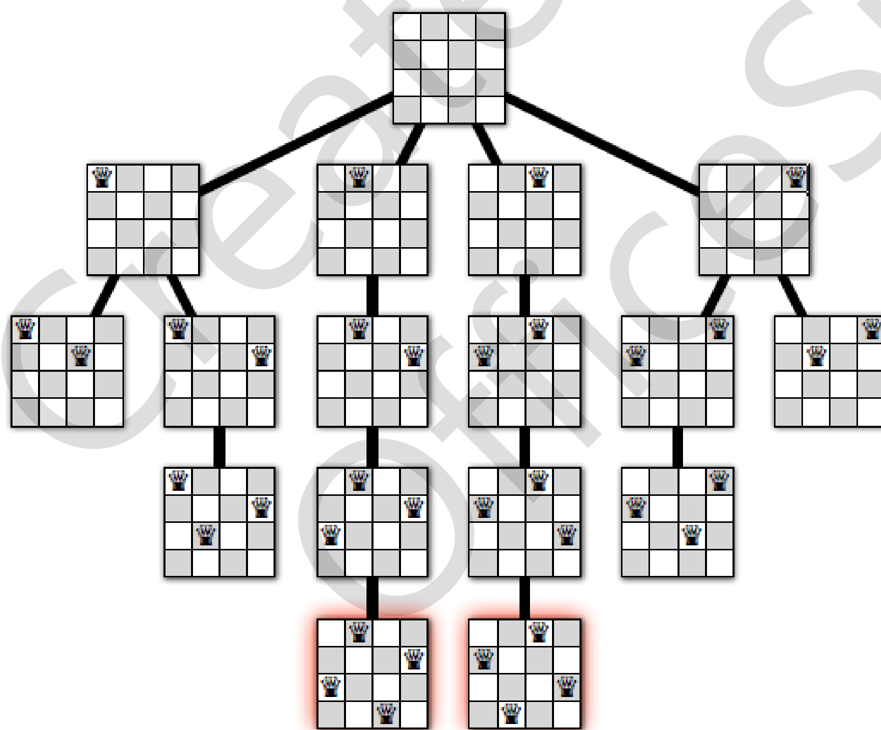
N-Queens Problem:

- It is a classic combinatorial problem.
- The eight queen's puzzle is the problem of placing eight queens puzzle is the problem of placing eight queens on an 8×8 chessboard so that no two queens attack each other.
- That is so that no two of them are on the same row, column, or diagonal.
- The 8-queens puzzle is an example of the more general n-queens problem of placing n queens on an n×n chessboard.

Created with OfficeSuite

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

One solution to the 8-queens problem



The complete recursion tree for our algorithm for the 4 queens problem.

Sum of Subsets Problem:

Created with OfficeSuite

Given positive numbers w_i $1 \le i \le n$, & m, here sum of subsets problem is finding all subsets of w_i whose sums are m.

Definition: Given n distinct +ve numbers (usually called weights), desire (want) to find all combinations of these numbers whose sums are m. this is called sum of subsets problem. To formulate this problem by using either fixed sized tuples or variable sized tuples.

Backtracking solution uses the fixed size tuple strategy.

For example:

If n=4 (w_1, w_2, w_3, w_4)=(11,13,24,7) and m=31.

Then desired subsets are (11, 13, 7) & (24, 7).

The two solutions are described by the vectors (1, 2, 4) and (3, 4).

In general all solution are k-tuples (x_1, x_2, x_3---x_k) $1 \le k \le n$, different solutions may have different sized tuples.

Explicit constraints requires $x_i \in \{j \,/\, j \text{ is an integer } 1 \le j \le n\}$

Implicit constraints requires:

No two be the same & that the sum of the corresponding w_i's be m i.e., (1, 2, 4) & (1, 4, 2) represents the same. Another constraint is $x_i < x_{i+1}$ $1 \le i \le k$

W_i weight of item i

Created with OfficeSuite

Created with OfficeSuite

M Capacity of bag (subset)
Xi the element of the solution vector is either one or zero.
Xi value depending on whether the weight wi is included or not.
If Xi=1 then wi is chosen.
If Xi=0 then wi is not chosen.

$$\underbrace{\sum_{i=1}^{k} W(i)X(i)}_{\text{Total sum till now}} + \underbrace{\sum_{i=k+1}^{n} W(i)}_{\text{Still there}} \geq M$$

The above equation specify that x1, x2, x3, --- xk cannot lead to an answer node if this condition is not satisfied.

$$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$$

The equation cannot lead to solution.

$$B_k(X(1), \ldots, X(k)) = true \text{ iff } \left(\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M \text{ and } \sum_{i=1}^{k} W(i)X(i) + W(k+1) \leq M \right)$$

$$s = \sum_{j=1}^{k-1} W(j)X(j) \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$

Recursive backtracking algorithm for sum of subsets problem

Algorithm SumOfSub(s, k, r)
{

Created with OfficeSuite

// $s = \sum_{j=1}^{k-1} W(j)X(j)$ and $r = \sum_{j=k}^{n} W(j)$

X[k]=1
If(S+w[k]=m) then write(x[1:]); // subset found.
Else if (S+w[k] + w{k+1] ≤ M)
Then SumOfSub(S+w[k], k+1, r-w[k]);
if ((S+r - w{k] ≥ M) and (S+w[k+1] ≤M)) then
{
X[k]=0;
SumOfSub(S, k+1, r-w[]
}

Graph Coloring:

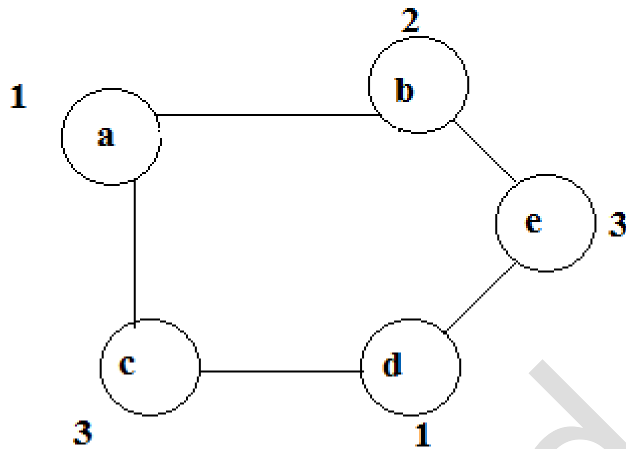
- Let G be a undirected graph and 'm' be a given +ve integer.
- The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only 'm' colors are used.
- The optimization version calls for coloring a graph using the minimum number of coloring.
- The decision version, known as K-coloring asks whether a graph is colourable using at most k-colors.

Note that, if 'd' is the degree of the given graph then it can be colored with 'd+1' colors.
The m- colorability optimization problem asks for

Created with OfficeSuite

the smallest integer ‘m’ for which the graph G can be colored. This integer is referred as “**Chromatic number**” of the graph.

Example



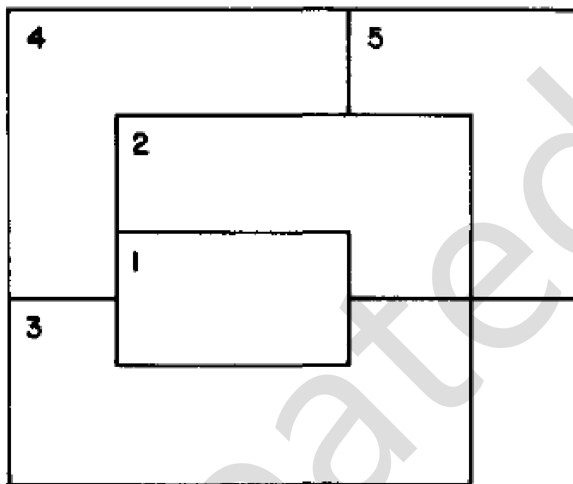
- Above graph can be colored with 3 colors 1, 2, & 3.
- The color of each node is indicated next to it.
- 3-colors are needed to color this graph and hence this graph’ Chromatic Number is 3.
- A graph is said to be planar iff it can be drawn in a plane (flat) in such a way that no two edges cross each other.
- **M-Colorability decision problem** is the 4-color problem for planar graphs.
- Given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only 4-colors are needed?
- To solve this problem, graphs are very useful,

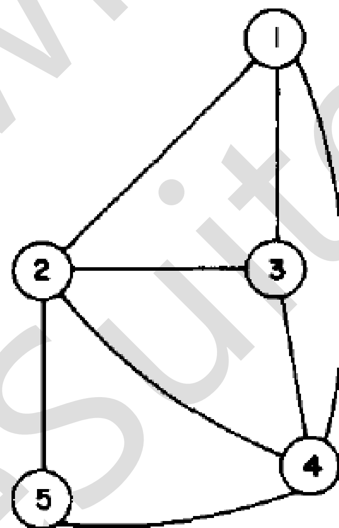
Created with OfficeSuite

because a map can easily be transformed into a graph.

- Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge.

o **Example:**





A map and its planar graph representation

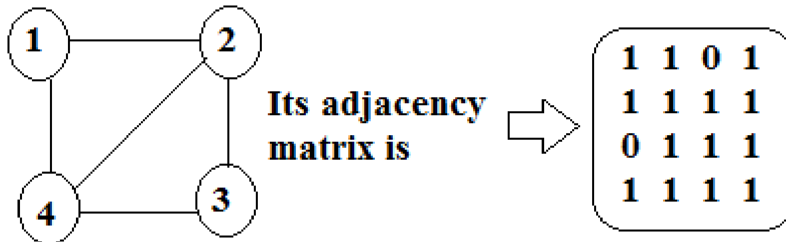
The above map requires 4 colors.

- Many years, it was known that 5-colors were required to color this map.
- After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They show that 4-colors are sufficient.

- Suppose we represent a graph by its adjacency matrix G[1:n, 1:n]

Created with OfficeSuite

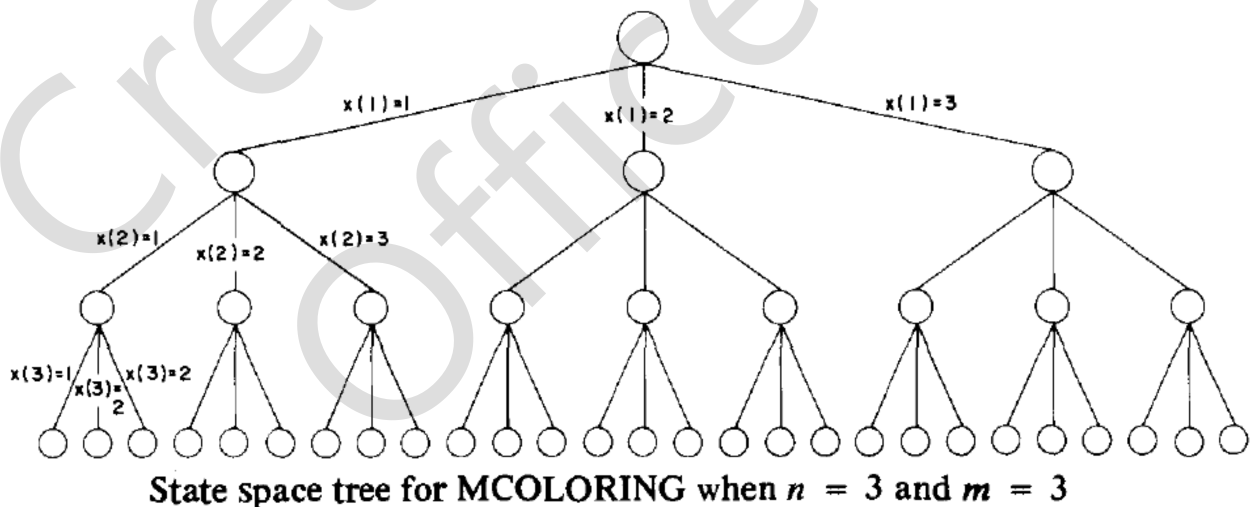
Ex:



- Here G[i, j]=1 if (i, j) is an edge of G, and G[i, j]=0 otherwise.
- Colors are represented by the integers 1, 2,---m and the solutions are given by the n-tuple (x1, x2,---xn)

xiColor of node i.

State Space Tree for
n=3nodes
m=3colors



State space tree for MCOLORING when $n = 3$ and $m = 3$

1st node coloured in 3-ways
2nd node coloured in 3-ways
3rd node coloured in 3-ways

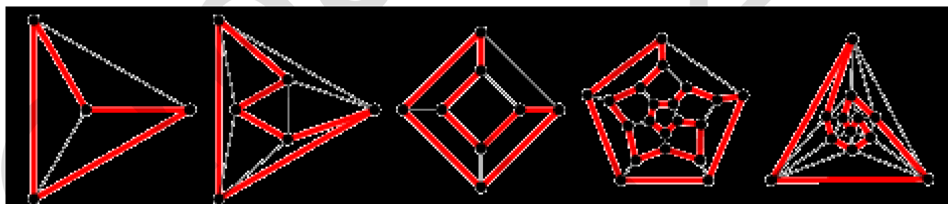
Created with OfficeSuite

So we can colour in the graph in 27 possibilities of colouring.

Hamiltonian Cycles:

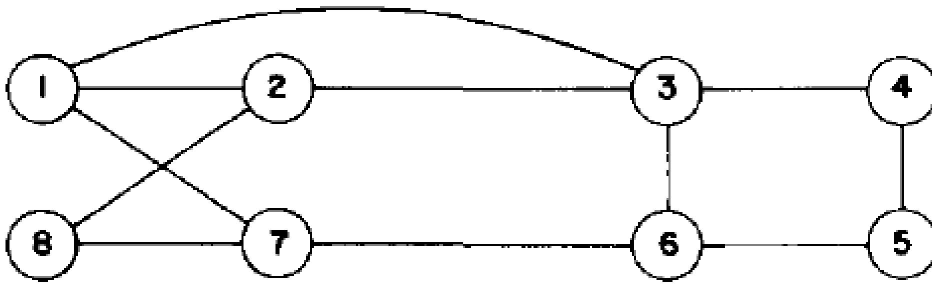
- **Def:** Let G=(V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once & returns to its starting position.
- It is also called the Hamiltonian circuit.
- Hamiltonian circuit is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.
- A graph possessing a Hamiltonian cycle is said to be Hamiltonian graph.

Example:

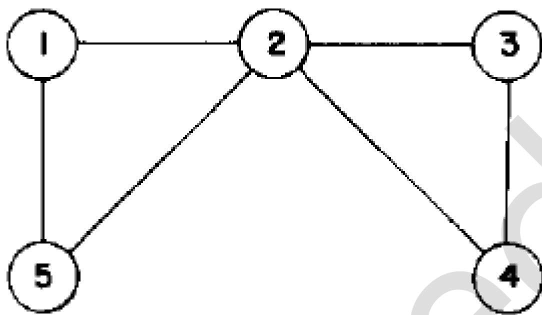


- In graph G, Hamiltonian cycle begins at some vertiex v1 ∈ G and the vertices of G are visited in the order v1,v2,---vn+1, then the edges (vi, vi+1) are in E, $1 \leq i \leq n$.

Created with OfficeSuite



- The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1



The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
- Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
- The graph may be directed or undirected. Only distinct cycles are output.
- From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}
- The backtracking solution vector (x1, x2, --- xn) xi□ ith visited vertex of proposed cycle.

Created with OfficeSuite

Created with OfficeSuite

- By using backtracking we need to determine how to compute the set of possible vertices for xk if x1,x2,x3---xk-1 have already been chosen.
- If k=1 then x1 can be any of the n-vertices.
- By using "NextValue" algorithm the recursive backtracking scheme to find all Hamiltoman cycles.
- This algorithm is started by 1st initializing the adjacency matrix G[1:n, 1:n] then setting x[2:n] to zero & x[1] to 1, and then executing Hamiltonian (2)

Branch & Bound

Branch & Bound (B & B) is general algorithm (or Systematic method) for finding optimal solution of various optimization problems, especially in discrete and combinatorial optimization.

- The B&B strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the B&B method
- Does not limit us to any particular way of traversing the tree.
- It is used only for optimization problem
- It is applicable to a wide variety of discrete combinatorial problem.
- B&B is rather general optimization technique

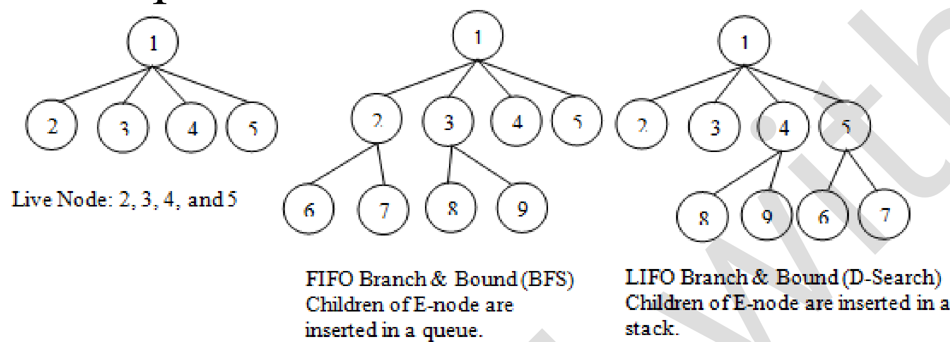
Created with OfficeSuite

that applies where the greedy method & dynamic programming fail.

- It is much slower, indeed (truly), it often (rapidly) leads to exponential time complexities in the worst case.
- The term B&B refers to all state space search methods in which all children of the "E-node" are generated before any other "live node" can become the "E-node"
- **Live node** is a node that has been generated but whose children have not yet been generated.
- **E-node** is a live node whose children are currently being explored.

Created with OfficeSuite

- **Dead node** is a generated node that is not to be expanded or explored any further.
- All children of a dead node have already been expanded.



- Two graph search strategies, BFS & D-search (DFS) in which the exploration of a new node cannot begin until the node currently being explored is fully explored.
- Both BFS & D-search (DFS) generalized to B&B strategies.
- **BFS** like state space search will be called FIFO (First In First Out) search as the list of live nodes is "First-in-first-out" list (or queue).
- **D-search (DFS)** Like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a "last-in-first-out" list (or stack).
- In backtracking, bounding function are used to help avoid the generation of sub-trees that do not contain an answer node.
- We will use 3-types of search strategies in branch and bound

Created with Office Suite

1) FIFO (First In First Out) search

2) LIFO (Last In First Out) search

3) LC (Least Count) search

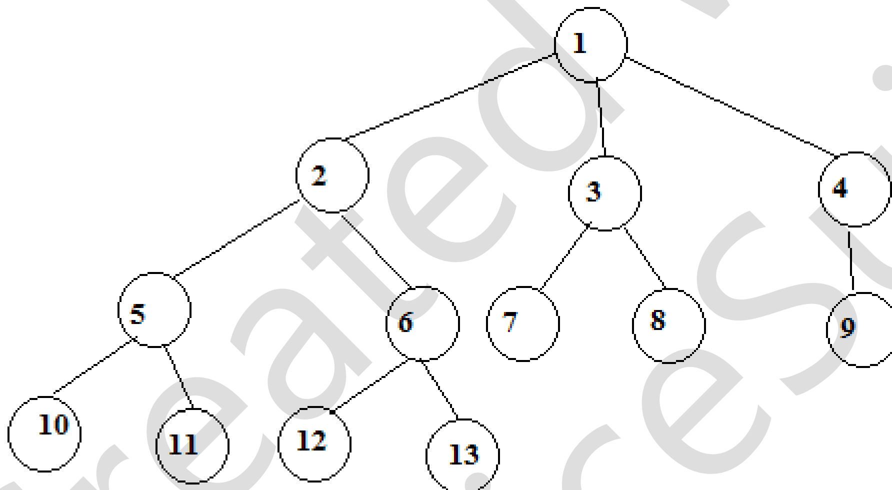
FIFO B&B:

FIFO Branch & Bound is a BFS.

In this, children of E-Node (or Live nodes) are inserted in a queue.

Implementation of list of live nodes as a queue

- Least()□ Removes the head of the Queue
- Add()□ Adds the node to the end of the Queue



Assume that node '12' is an answer node in FIFO search, 1st we take E-node has '1'

Created with OfficeSuite

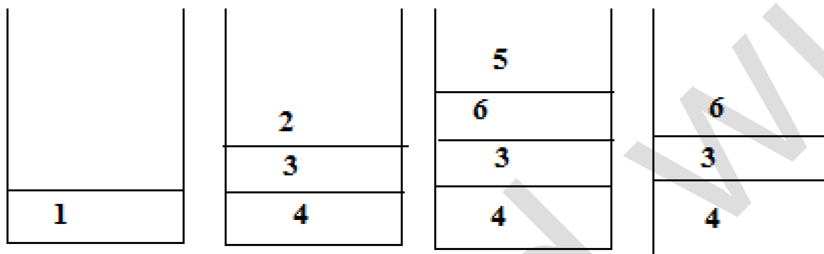
LIFO B&B:

LIFO Brach & Bound is a D-search (or DFS).

In this children of E-node (live nodes) are inserted in a stack

Implementation of List of live nodes as a stack

- Least() Removes the top of the stack
- ADD()Adds the node to the top of the stack.



Least Cost (LC) Search:

- The selection rule for the next E-node in FIFO or LIFO branch and bound is sometimes "blind". i.e., the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.
- The search for an answer node can often be speeded by using an "intelligent" ranking function. It is also called an approximate cost function "$\hat{C}$".
- Expended node (E-node) is the live node with the best $\hat{C}$ value.

Branching:

- A set of solutions, which is represented by a node, can be partitioned into mutually (jointly or commonly) exclusive (special) sets.
- Each subset in the partition is represented by a

Created with OfficeSuite

child of the original node.

Lower bounding:

- An algorithm is available for calculating a lower bound on the cost of any solution in a given subset.
- Each node X in the search tree is associated with a cost: **Ĉ(X) C=**cost of reaching the current node, X(E-node) form the root + The cost of reaching an answer node form X.
- **Ĉ=g(X)+H(X).**

Example:

8-puzzle

Cost function: **Ĉ** = g(x) +h(x)

where h(x) = the number of misplaced tiles

and g(x) = the number of moves so far

Assumption: move one tile in any direction cost 1.

O/1 Knapsack Problem

What is Knapsack Problem:

- Knapsack problem is a problem in combinatorial optimization, Given a set of items, each with a mass & a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit & the total value is as large as possible.

O-1 Knapsack Problem can formulate as.

- Let there be n items, **Z1 to Zn** where **Zi** has value **Pi** & weight **wi**. The maximum weight that can carry in the bag is m.

Created with OfficeSuite

- All values and weights are non negative.
- Maximize the sum of the values of the items in the knapsack, so that sum of the weights must be less than the knapsack's capacity m.
- The formula can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

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

Xi=0 or 1 1 ≤ i ≤ n

To solve o/1 knapsack problem using B&B:

- Knapsack is a maximization problem
- Replace the objective function by the function to make it into a minimization problem
- The modified knapsack problem is stated as

$$\text{Minimize} -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i < m,$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

Fixed tuple size solution space:

o Every leaf node in state space tree represents an answer for which

$$\sum_{1 \le i \le n} w_i x_i \leqslant m$$

is an answer node; other leaf nodes are infeasible

- o For optimal solution, define

$$c(x) = -\sum_{1 \le i \le n} p_i x_i$$

for every answer node x

- o For infeasible leaf nodes,

Created with OfficeSuite

- For non leaf nodes

$c(x) = \min\{c(lchild(x)), c(rchild(x))\}$

  - Define two functions $\hat{c}(x)$ and $u(x)$ such that for every node x,

$\hat{c}(x) \leq c(x) \leq u(x)$

Created with OfficeSuite

➢Computing $\hat{c}(\cdot)$ and $u(\cdot)$

Let x be a node at level j, $1 \leq j \leq n+1$

Cost of assignment: $-\sum_{1 \leq i < j} p_i x_i$

$c(x) \leq -\sum_{1 \leq i < j} p_i x_i$

We can use $u(x) = -\sum_{1 \leq i < j} p_i x_i$

Using $q = -\sum_{1 \leq i < j} p_i x_i$, an improved upper bound function $u(x)$ is

$$u(x) = \texttt{ubound}(q, \sum_{1 \leq i < j} w_i x_i, j-1, m)$$

```
Algorithm ubound ( cp, cw, k, m )
{
// Input: cp: Current profit total
// Input: cw: Current weight total
// Input: k: Index of last removed item
// Input: m: Knapsack capacity
b=cp; c=cw;
for i:=k+1 to n do{
if(c+w[i] ≤ m) then {
c:=c+w[i]; b=b-p[i];
}
}
return b;
}
```

Created with OfficeSuite