**LECTURE NOTES**

**ON**

**DESIGN & ANALYSIS OF ALGORITHEMS**

**II B-Tech II Semester**

****

**INFORMATION TECHNOLOGY**

**CMR TECHNICAL CAMPUS**

**KANDLAKOYA (V), MEDCHAL (M), R.R.DIST.**

****

**CMR TECHNICAL CAMPUS**

**KANDLAKOYA (V), MEDCHAL (M), R.R.DIST.**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**SUB:DESIGN & ANALYSIS OF ALGORITHEMS**

**COURSE MATERIAL**

**Objectives:**

* To analyze performance of algorithms.
* To choose the appropriate data structure and algorithm design method for a specified application.
* To understand how the choice of data structures and algorithm design methods impacts the performance of programs.
* To solve problems using algorithm design methods such as the greedy method, divide and conquer, dynamic programming, backtracking and branch and bound.
* Prerequisites (Subjects) Data structures, Mathematical foundations of computer science.

**UNIT I:**

**Introduction:** Algorithm, Psuedo code for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh notation, Omega notation, Theta notation and Little oh notation, Probabilistic analysis, Amortized analysis.

**Divide and conquer:** General method , applications-Binary search, Quick sort, Merge sort, Strassen’s matrix multiplication.

**UNIT II:**

**Searching and Traversal Techniques:**Efficient non - recursive binary tree traversal algorithm, Disjoint set operations, union and find algorithms, Spanning trees, Graph traversals - Breadth first search and Depth first search, AND / OR graphs, game trees, Connected Components, Bi - connected components.Disjoint Sets- disjoint set operations, union and find algorithms, spanning trees, connected components and biconnected components.

**UNIT III:**

**Greedy method:** General method, applications - Job sequencing with dead lines, 0/1 knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

**Dynamic Programming:** General method, applications-Matrix chain multiplication, Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem,Travelling sales person problem, Reliability design.

**UNIT IV:**

**Backtracking:** General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.

**Branch and Bound:** General method, applications - Travelling sales person problem,0/1 knapsack problem- LC Branch and Bound solution, FIFO Branch and Bound solution.

**UNIT V:**

**NP-Hard and NP-Complete problems:** Basic concepts, non deterministic algorithms, NP - Hard and NPComplete classes, Cook’s theorem.

**TEXT BOOKS :**

1. Fundamentals of Computer Algorithms, Ellis Horowitz,Satraj Sahni and  Rajasekharam,Galgotia publications pvt. Ltd.
2. Foundations of Algorithm, 4th edition, R. Neapolitan and K. Naimipour, Jones and Bartlett Learning.
3. Design and Analysis of Algorithms, P. H. Dave, H. B. Dave, Pearson Education, 2008.

**REFERENCES :**

1. Computer Algorithms, Introduction to Design and Analysis, 3rd Edition, Sara Baase, Allen, Van, Gelder, Pearson Education.
2. Algorithm Design: Foundations, Analysis and Internet examples, M. T. Goodrich and R. Tomassia, John Wiley and sons.
3. Fundamentals of Sequential and Parallel Algorithm, K. A. Berman and J. L. Paul, Cengage Learning.
4. Introducation to the Design and Analysis of Algorithms, A. Levitin, Pearson Education.
5. Introducation to Algorithms, 3rd Edition, T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, PHI Pvt. Ltd.
6. Design and Analysis of algorithm, Aho, Ullman and Hopcroft, Pearson Education, 2004.

**Outcomes:**

* **​**Be able to analyze algorithms and improve the efficiency of algorithms.
* Apply different designing methods for development of algorithms to realistic problems, such as divide and conquer, greedy and etc. Ability to understand and estimate the performance of algorithm.

**Design and Analysis of Algorithm**

**UNIT – I**

**ALGORITHM**

**Informal Definition:**

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the i/p into the o/p.

**Formal Definition:**

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

1. **INPUT** - Zero or more quantities are externally supplied.
2. **OUTPUT** - At least one quantity is produced.
3. **DEFINITENESS** \_Each instruction is clear and unambiguous.
4. **FINITENESS** \_ If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **EFFECTIVENESS** \_ Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

**Issues or study of Algorithm:**

* + How to device or design an algorithm
  + How to express an algorithm- definiteness.
  + How to analysis an algorithm - time and space complexity.
  + How to validate an algorithm - fitness.
  + Testing the algorithm \_checking for error.

**Algorithm Specification:**

Algorithm can be described in three ways.

**1. Natural language like English:**

This method should ensure that each & every statement is definite.

**2. Graphic representation called flowchart**:

This method will work well when the algorithm is small& simple.

**3. Pseudo-code Method:**

In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

**Pseudo-Code Conventions:**

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

Node. Record

{

data type – 1 data-1;

**.**

**.**

**.**

data type – n data – n; node \* link;

}

Here link is a pointer to the record type node. Individual data items of a record can be accessed with  and period.

5. Assignment of values to variables is done using the assignment statement. <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

|  |  |
| --- | --- |
| Logical Operators | AND, OR, NOT |
| Relational Operators | <, <=,>,>=, =, != |

7. The following looping statements are employed.

For, while and repeat-until While Loop:

While < condition > do

{

<statement-1>

**.**

**.**

**.**

<statement-n>

}

**For Loop:**

For variable: = value-1 to value-2 step step do

{

<statement-1>

**.**

**.**

**.**

<statement-n>

}

**repeat-until:**

repeat

<statement-1>

**.**

**.**

**.**

<statement-n> until<condition>

1. A conditional statement has the following forms.
   * If <condition> then <statement>
   * If <condition> then <statement-1> Else <statement-1>

**Case statement:**

Case

{

**:** <condition-1> **:** <statement-1>

**.**

**.**

**.**

* <condition-n> **:** <statement-n>
* else **:** <statement-n+1>

}

1. Input and output are done using the instructions read & write.
2. There is only one type of procedure: Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

As an example, the following algorithm fields & returns the maximum of „n‟ given numbers:

1. algorithm Max(A,n)
2. // A is an array of size n
3. {
4. Result := A[1];
5. for I:= 2 to n do
6. if A[I] > Result then
7. Result :=A[I];
8. return Result;
9. }

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

**Recursive Algorithms:**

* A Recursive function is a function that is defined in terms of itself.
* Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
* An algorithm that calls itself is Direct Recursive.
* Algorithm „A‟ is said to be Indirect Recursive if it calls another algorithm which in turns calls „A‟.
* The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
* The following 2 examples show how to develop a recursive algorithms.
  + - In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.
  1. **Towers of Hanoi:**

**.**

**.**

Tower A Tower B Tower C

* It is Fashioned after the ancient tower of Brahma ritual.
* According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
* The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
* Besides these tower there were two other diamond towers(labeled B & C)
* Since the time of creation, Brehman priests have been attempting to move the disks from tower A to tower B using tower C, for intermediate storage.
* As the disks are very heavy, they can be moved only one at a time.
* In addition, at no time can a disk be on top of a smaller disk.
* According to legend, the world will come to an end when the priest have

completed this task.

* A very elegant solution results from the use of recursion.
* Assume that the number of disks is „n‟.
* To get the largest disk to the bottom of tower B, we move the remaining „n-1‟ disks to tower C and then move the largest to tower B.
* Now we are left with the tasks of moving the disks from tower C to B.
* To do this, we have tower A and B available.
* The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

**Algorithm:**

1. Algorithm TowersofHanoi(n,x,y,z)
2. //Move the top „n‟ disks from tower x to tower y.
3. {

**.**

**.**

**.**

4.if(n>=1) then

* 1. {
  2. TowersofHanoi(n-1,x,z,y);
  3. Write(“move top disk from tower “ X ,”to top of tower “ ,Y);
  4. Towersofhanoi(n-1,z,y,x);
  5. }
  6. }

1. **Permutation Generator:**

* Given a set of n>=1elements, the problem is to print all possible permutations of this set.
* For example, if the set is {a,b,c} ,then the set of permutation is, { (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}
* It is easy to see that given „n‟ elements there are n! different permutations.
* A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)
* The Answer can be constructed by writing

1. a followed by all the permutations of (b,c,d)
2. b followed by all the permutations of(a,c,d)
3. c followed by all the permutations of (a,b,d)
4. d followed by all the permutations of (a,b,c)

**Algorithm:**

Algorithm perm(a,k,n)

{

if(k=n) then write (a[1:n]); // output permutation else //a[k:n] ahs more than one permutation

// Generate this recursively. for I:=k to n do

{

t:=a[k];

a[k]:=a[I];

a[I]:=t;

perm(a,k+1,n);

//all permutation of a[k+1:n] t:=a[k];

a[k]:=a[I];

a[I]:=t;

}

}

**Performance Analysis:**

**1. Space Complexity:**

The space complexity of an algorithm is the amount of money it needs to run to compilation.

**2. Time Complexity:**

The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

**Space Complexity:**

Space Complexity Example: Algorithm abc(a,b,c)

{

return a+b++\*c+(a+b-c)/(a+b) +4.0;

}

 The Space needed by each of these algorithms is seen to be the sum of the following component.

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

1. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by

referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

 The space requirement s(p) of any algorithm p may therefore be written as, S(P) = c+ Sp(Instance characteristics)

Where „c‟ is a constant.

**Example 2:**

Algorithm sum(a,n)

{

s=0.0;

for I=1 to n do s= s+a[I]; return s;

}

* The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by „n‟ is one word, since it is of type integer.
* The space needed by „a‟a is the space needed by variables of tyepe array of floating point numbers.
* This is atleast „n‟ words, since „a‟ must be large enough to hold the „n‟ elements to be summed.
* So,we obtain Ssum(n)>=(n+s)

[ n for a[],one each for n,I a& s]

**Time Complexity:**

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

 The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments  0 steps.

Assignment statements 1 steps.

[Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until Control part of the statement.

|  |  |
| --- | --- |
| 1. | We introduce a variable, count into the program statement to increment count with |

initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

**Algorithm:**

Algorithm sum(a,n)

{

s= 0.0;

count = count+1; for I=1 to n do

{

count =count+1;

s=s+a[I];

count=count+1;

}

count=count+1;

count=count+1; return s;

}

 If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Statement* | | *S/e* | *Frequency* | *Total* |
|  | |  |  |  |
| 1. Algorithm Sum(a,n) | | 0 | - | 0 |
| 2.{ |  | 0 | - | 0 |
| 3. | S=0.0; | 1 | 1 | 1 |
| 4. | for I=1 to n do | 1 | n+1 | n+1 |
| 5. | s=s+a[I]; | 1 | n | n |
| 6. | return s; | 1 | 1 | 1 |
| 7. | } | 0 | - | 0 |
|  |  |  |  |  |
| *Total* | |  |  | 2n+3 |
|  | |  |  |  |

**AVERAGE –CASE ANALYSIS**

* Most of the time, average-case analysis are performed under the more or less realistic assumption that all instances of any given size are equally likely.
* For sorting problems, it is simple to assume also that all the elements to be sorted are distinct.
* Suppose we have „n‟ distinct elements to sort by insertion and all n! permutation of these elements are equally likely.
* To determine the time taken on a average by the algorithm ,we could add the times required to sort each of the possible permutations ,and then divide by n! the answer thus obtained.
* An alternative approach, easier in this case is to analyze directly the time required by the algorithm, reasoning probabilistically as we proceed.

 For any I,2  I n, consider the sub array, T[1….i].

* The partial rank of T[I] is defined as the position it would occupy if the sub array were sorted.
* For Example, the partial rank of T[4] in [3,6,2,5,1,7,4] in 3 because T[1….4] once sorted is [2,3,5,6].
* Clearly the partial rank of T[I] does not depend on the order of the element in
* Sub array T[1…I-1].

**Analysis**

**Best case**:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

**Worst case**:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

**Average case:**

This type of analysis results in average running time over every type of input.

**Complexity:**

Complexity refers to the rate at which the storage time grows as a function of the problem size

**Asymptotic analysis:**

Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

**Asymptotic notation:**

**Big ‘oh’:** the function f(n)=O(g(n)) iff there exist positive constants c and no such thatf(n)≤c\*g(n) for all n, n ≥ no.

**Omega:** the function f(n)=Ω(g(n)) iff there exist positive constants c and no such that f(n)≥c\*g(n) for all n, n ≥ no.

**Theta:** the function f(n)=ө(g(n)) iff there exist positive constants c1,c2 and no such that c1 g(n)≤ f(n) ≤ c2 g(n) for all n, n ≥ no.

**Recursion:**

Recursion may have the following definitions:

-The nested repetition of identical algorithm is recursion. -It is a technique of defining an object/process by itself.

-Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

**When to use recursion:**

Recursion can be used for repetitive computations in which each action is stated in terms of previous result. There are two conditions that must be satisfied by any recursive procedure.

1. Each time a function calls itself it should get nearer to the solution.
2. There must be a decision criterion for stopping the process.

In making the decision about whether to write an algorithm in recursive or non-recursive form, it is always advisable to consider a tree structure for the problem. If the structure is simple then use non-recursive form. If the tree appears quite bushy, with little duplication of tasks, then recursion is suitable.

The recursion algorithm for finding the factorial of a number is given below,

**Algorithm** : factorial-recursion

**Input** : n, the number whose factorial is to be found. **Output :** f, the factorial of n

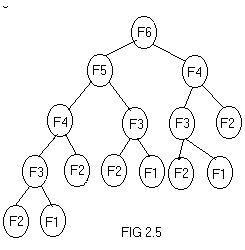
**Method** : if(n=0)f=1

else f=factorial(n-1) \* n if end

algorithm ends.

The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.
2. If the termination criterion is reached perform final computation and goto step 3 otherwise perform final computations and goto step 1



1. Restore the most recently saved parameters, local variable and return address and goto the latest return address. ]

**probabilistic Analysis**

**1. Average cost**

For a given algorithm and an instance size, if instance *i* has cost *ci*, i = 1..m, then

* best-case cost: *Minimum(ci, i = 1..m)*
* worst-case cost: *Maximum(ci, i = 1..m)*
* average-case cost: ∑[i = 1..m] *pici*, where *pi* is the probability for instance *i* to appear to the algorithm.

Average cost matters if the algorithm will be used for many times, the overall efficiency is more important, and exceptions are acceptable. How about real-time systems?

The actual probability distribution is usually domain-specific, and not a property of the algorithm.

When all instances have the same probability (i.e., *1/m*), the average cost is ∑[i = 1..m] *(1/m)ci* = (∑[i = 1..m] *ci) / m*.

For linear search, if the target value has the same probability to appear in any positions in an array, then what is the average number of comparisons? Consider the following three cases:

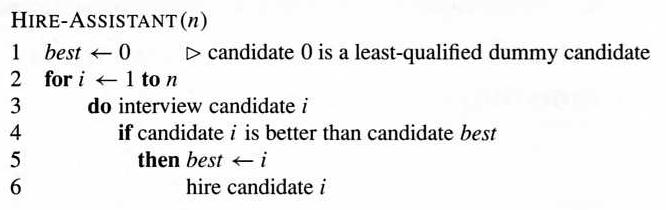
1. The target is in the array.
2. The probability for the target to be not in the array is the same as for it to be in any position of the array.
3. The probability for the target to be not in the array is the same as for it to be in the array.

Relation between average cost and the best/worst cost.

Cost as exact function, approximate function, and order of growth: when to use which?

How about binary search?

**2. Example: the hiring problem**

To hire a new office assistant if the new candidate is the best so far (page [92](http://library.books24x7.com.libproxy.temple.edu/book/id_3444/viewer.asp?bookid=3444&chunkid=280250392)):  


The quantity to be analyzed is the cost of the procedure, reflected by line 3 and 6, not the running time of the algorithm. However, the analysis is similar: we want to know the number of times for line 3 and 6 to be executed, respectively.

[Amortized Analysis](http://en.wikipedia.org/wiki/Amortized_analysis)

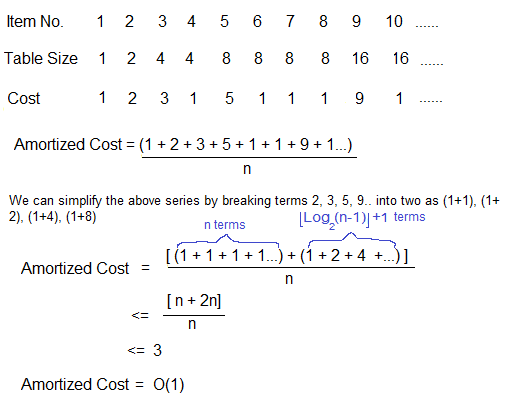
It is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.  
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

The solution to this trade-off problem is to use [Dynamic Table (or Arrays)](http://en.wikipedia.org/wiki/Dynamic_array). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.  
1) Allocate memory for a larger table of size, typically twice the old table.  
2) Copy the contents of old table to new table.  
3) Free the old table.

If the table has space available, we simply insert new item in available space.

**What is the time complexity of n insertions using the above scheme?**  
If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n \* O(n) which is O(n2). This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don’t take Θ(n) time.

[](http://d1gjlxt8vb0knt.cloudfront.net/wp-content/uploads/AmortizedAnalysis.png)

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in [vectors in C++,](http://www.cplusplus.com/reference/vector/vector/) [ArrayList in Java](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html).

Following are few important notes.  
**1)** Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

**2)** The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called [***Accounting Method***](http://en.wikipedia.org/wiki/Accounting_method)and [***Potential Method***](http://en.wikipedia.org/wiki/Potential_method). We will be discussing the other two methods in separate posts.

**3)** The amortized analysis doesn’t involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

**DIVIDE AND CONQUER:**

**General method:**

1. Given a function to suggests splitting the problems.

compute on „n‟ inputs the divide-and-conquer strategy inputs into „k‟ distinct subsets, 1<k<=n, yielding „k‟ sub

* These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
* If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
* Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.
* For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.
* D And C(Algorithm) is initially invoked as D and C(P), where „p‟ is the problem to be solved.
* Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting.
* If this so, the function „S‟ is invoked.
* Otherwise, the problem P is divided into smaller sub problems.
* These sub problems P1, P2 …Pk are solved by recursive application of D And C.
* Combine is a function that determines the solution to p using the solutions to the

„k‟ sub problems.

* If the size of „p‟ is n and the sizes of the „k‟ sub problems are n1, n2 ….nk, respectively, then the computing time of D And C is described by the recurrence relation.

T(n)= { g(n) n small

T(n1)+T(n2)+……………+T(nk)+f(n); otherwise.

Where T(n)  is the time for D And C on any I/p of size „n‟.

g(n)  is the time of compute the answer directly for small I/ps. f(n)  is the time for dividing P & combining the solution to

sub problems.

1. Algorithm D And C(P)
2. {
3. if small(P) then return S(P);
4. else
5. {
6. divide P into smaller instances

P1, P2… Pk, k>=1;

1. Apply D And C to each of these sub problems;
2. return combine (D And C(P1), D And C(P2),…….,D And C(Pk));
3. }
4. }

* The complexity of many divide-and-conquer algorithms is given by recurrences of the form

T(n) = { T(1) n=1

AT(n/b)+f(n) n>1

* Where a & b are known constants.
* We assume that T(1) is known & „n‟ is a power of b(i.e., n=b^k)
* One of the methods for solving any such recurrence relation is called the substitution method.
* This method repeatedly makes substitution for each occurrence of the function. T is the Right-hand side until all such occurrences disappear.

Example:

1. Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n. We have,

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

* + 2[2T(n/2/2)+n/2]+n
  + [4T(n/4)+n]+n
  + 4T(n/4)+2n
  + 4[2T(n/4/2)+n/4]+2n
  + 4[2T(n/8)+n/4]+2n
  + 8T(n/8)+n+2n
  + 8T(n/8)+3n

\*

\*

\*

1. In general, we see that T(n)=2^iT(n/2^i )+in., for any log n >=I>=1.

 T(n) =2^log n T(n/2^log n) + n log n

Corresponding to the choice of i=log n

 Thus, T(n) = 2^log n T(n/2^log n) + n log n

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

= n. T(1) + n log n [since, log 1=0, 2^0=1] = 2n + n log n

**BINARY SEARCH**

* Algorithm Bin search(a,n,x)
* // Given an array a[1:n] of elements in non-decreasing
* //order, n>=0,determine whether „x‟ is present and
* // if so, return „j‟ such that x=a[j]; else return 0.
* {
* low:=1; high:=n;
* while (low<=high) do
* {
* **mid:=[(low+high)/2];**
* if (x<a[mid]) then high;
* else if(x>a[mid]) then

low=mid+1;

* else return mid;
* }
* return 0;
* }
* Algorithm, describes this binary search method, where Binsrch has 4I/ps a[], I , l & x.
* It is initially invoked as Binsrch (a,1,n,x)
* A non-recursive version of Binsrch is given below.
* This Binsearch has 3 i/ps a,n, & x.
* The while loop continues processing as long as there are more elements left to check.
* At the conclusion of the procedure 0 is returned if x is not present, or „j‟ is returned, such that a[j]=x.
* We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one.
* Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if „x‟ is not present.

Example:

1) Let us select the 14 entries. -15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

* Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of „x‟.
* Only the variables, low, high & mid need to be traced as we simulate the algorithm.
* We try the following values for x: 151, -14 and 9.

for 2 successful searches & 1 unsuccessful search.

* Table. Shows the traces of Bin search on these 3 steps.

|  |  |  |  |
| --- | --- | --- | --- |
| X=151 | low | high | mid |
|  | 1 | 14 | 7 |
|  | 8 | 14 | 11 |
|  | 12 | 14 | 13 |
|  | 14 | 14 | 14 |
|  |  |  | Found |
| x=-14 | low | high | mid |
|  | 1 | 14 | 7 |
|  | 1 | 6 | 3 |
|  | 1 | 2 | 1 |
|  | 2 | 2 | 2 |
|  | 2 | 1 | Not found |
| x=9 | low | high | mid |
|  | 1 | 14 | 7 |
|  | 1 | 6 | 3 |
|  | 4 | 6 | 5 |
|  |  |  | Found |

**Theorem:** Algorithm Binsearch(a,n,x) works correctly.

**Proof:**

We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

* Initially low =1, high= n,n>=0, and a[1]<=a[2]<=……..<=a[n].
* If n=0, the while loop is not entered and is returned.
* Otherwise we observe that each time thro‟ the loop the possible elements to be checked of or equality with x and a[low], a[low+1],……..,a[mid],……a[high].
* If x=a[mid], then the algorithm terminates successfully.
* Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
* Clearly, this narrowing of the range does not affect the outcome of the search.
* If low becomes > than high, then „x‟ is not present & hence the loop is exited.

**MERGE SORT**

* As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n)
* This algorithm is called merge sort
* We assume throughout that the elements are to be sorted in non-decreasing order.
* Given a sequence of „n‟ elements a[1],…,a[n] the general idea is to imagine then split into 2 sets a[1],…..,a[n/2] and a[[n/2]+1],….a[n].
* Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of „n‟ elements.
* Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

**Algorithm For Merge Sort:**

1. Algorithm MergeSort(low,high)
2. //a[low:high] is a global array to be sorted
3. //Small(P) is true if there is only one element
4. //to sort. In this case the list is already sorted.
5. {
6. if (low<high) then //if there are more than one element
7. {
8. //Divide P into subproblems
9. //find where to split the set
10. **mid = [(low+high)/2];**
11. //solve the subproblems.
12. mergesort (low,mid);
13. mergesort(mid+1,high);
14. //combine the solutions .
15. merge(low,mid,high);
16. }
17. }

**Algorithm:** Merging 2 sorted subarrays using auxiliary storage.

1. Algorithm merge(low,mid,high)
2. //a[low:high] is a global array containing
3. //two sorted subsets in a[low:mid]
4. //and in a[mid+1:high].The goal is to merge these 2 sets into
5. //a single set residing in a[low:high].b[] is an auxiliary global array.
6. {
7. h=low; I=low; j=mid+1;
8. while ((h<=mid) and (j<=high)) do
9. {
10. if (a[h]<=a[j]) then
11. {
12. b[I]=a[h];
13. h = h+1;
14. }
15. else
16. {
17. b[I]= a[j];
18. j=j+1;
19. }
20. I=I+1;
21. }
22. if (h>mid) then
23. for k=j to high do
24. {
25. b[I]=a[k];
26. I=I+1;
27. }
28. else
29. for k=h to mid do
30. {
31. b[I]=a[k];
32. I=I+1;
33. }
34. for k=low to high do a[k] = b[k];
35. }

* Consider the array of 10 elements a[1:10] =(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)
* Algorithm Mergesort begins by splitting a[] into 2 sub arrays each of size five (a[1:5] and a[6:10]).
* The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3] ) and 2(a[4:5])
* Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])
* The 2 values in a[1:2} are split to find time into one-element sub arrays, and now the merging begins.

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520)

 Where vertical bars indicate the boundaries of sub arrays.

Elements a[I] and a[2] are merged to yield,

(285, 310|179|652, 351| 423, 861, 254, 450, 520)

 Then a[3] is merged with a[1:2] and

(179, 285, 310| 652, 351| 423, 861, 254, 450, 520)

 Next, elements a[4] & a[5] are merged.

(179, 285, 310| 351, 652 | 423, 861, 254, 450, 520)

 And then a[1:3] & a[4:5]

(179, 285, 310, 351, 652| 423, 861, 254, 450, 520)

* Repeated recursive calls are invoked producing the following sub arrays. (179, 285, 310, 351, 652| 423| 861| 254| 450, 520)
* Elements a[6] &a[7] are merged.

Then a[8] is merged with a[6:7]

(179, 285, 310, 351, 652| 254,423, 861| 450, 520)

 Next a[9] &a[10] are merged, and then a[6:8] & a[9:10] (179, 285, 310, 351, 652| 254, 423, 450, 520, 861 )

 At this point there are 2 sorted sub arrays & the final merge produces the fully sorted result.

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

* If the time for the merging operations is proportional to „n‟, then the computing time for merge sort is described by the recurrence relation.

|  |  |
| --- | --- |
| T(n) = { a | n=1,‟a‟ a constant |
| 2T(n/2)+cn | n>1,‟c‟ a constant. |

 When „n‟ is a power of 2, n= 2^k, we can solve this equation by successive substitution.

T(n) =2(2T(n/4) +cn/2) +cn

* + 4T(n/4)+2cn
  + 4(2T(n/8)+cn/4)+2cn

\*

\*

* + 2^k T(1)+kCn.
  + an + cn log n.
* It is easy to see that if s^k<n<=2^k+1, then T(n)<=T(2^k+1). Therefore,

**T(n)=O(n log n)**

**QUICK SORT**

* The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.
* In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.
* In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.
* This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.
* Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.
* No merge is needed. This rearranging is referred to as partitioning.
* Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]
* It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to all elements in a[1:n]
* The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in the set are better in practice.
* The function interchange (a,I,j) exchanges a[I] with a[j].

**Algorithm**: Partition the array a[m:p-1] about a[m]

1. Algorithm Partition(a,m,p)
2. //within a[m],a[m+1],…..,a[p-1] the elements
3. // are rearranged in such a manner that if
4. //initially t=a[m],then after completion
5. //a[q]=t for some q between m and
6. //p-1,a[k]<=t for m<=k<q, and
7. //a[k]>=t for q<k<p. q is returned
8. //Set a[p]=infinite.
9. {
10. v=a[m];I=m;j=p;
11. repeat
12. {
13. repeat
14. I=I+1;
15. until(a[I]>=v);
16. repeat
17. j=j-1;
18. until(a[j]<=v);
19. if (I<j) then interchange(a,i.j);
20. }until(I>=j);
21. a[m]=a[j]; a[j]=v;
22. retun j;
23. }
24. Algorithm Interchange(a,I,j)
25. //Exchange a[I] with a[j]
26. {
27. p=a[I];
28. a[I]=a[j];
29. a[j]=p;
30. }

**Algorithm*:*** Sorting by Partitioning

1. Algorithm Quicksort(p,q)
2. //Sort the elements a[p],….a[q] which resides
3. //is the global array a[1:n] into ascending
4. //order; a[n+1] is considered to be defined
5. // and must be >= all the elements in a[1:n]
6. {
7. if(p<q) then // If there are more than one element
8. {
9. // divide p into 2 subproblems
10. j=partition(a,p,q+1);
11. //‟j‟ is the position of the partitioning element.
12. //solve the subproblems.
13. quicksort(p,j-1);
14. quicksort(j+1,q);
15. //There is no need for combining solution.
16. }
17. }

Record Program: Quick Sort #include <stdio.h>

#include <conio.h> int a[20];

main()

{

int n,I; clrscr();

printf(“QUICK SORT”);

printf(“\n Enter the no. of elements “); scanf(“%d”,&n);

printf(“\nEnter the array elements”); for(I=0;I<n;I++)

scanf(“%d”,&a[I]); quicksort(0,n-1);

printf(“\nThe array elements are”); for(I=0;I<n;I++) printf(“\n%d”,a[I]);

getch();

}

quicksort(int p, int q)

{

int j; if(p,q)

{

j=partition(p,q+1); quicksort(p,j-1); quicksort(j+1,q);

}

}

Partition(int m, int p)

{

int v,I,j; v=a[m]; i=m; j=p;

do

{

do

i=i+1;

while(a[i]<v); if (i<j)

interchange(I,j); } while (I<j);

a[m]=a[j];

a[j]=v; return j;

}

Interchange(int I, int j)

{

int p; p= a[I];

a[I]=a[j];

a[j]=p;

}

Output:

Enter the no. of elements 5 Enter the array elements

3

8

1

5

2

The sorted elements are, 1 2 3 5 8

**STRASSON’S MATRIX MULTIPLICAION**

* Let A and B be the 2 n\*n Matrix. The product matrix C=AB is calculated by using the formula,

C (i ,j )= A(i,k) B(k,j) for all „i‟ and and j between 1 and n.

* The time complexity for the matrix Multiplication is O(n^3).
* Divide and conquer method suggest another way to compute the product of n\*n matrix.
* We assume that N is a power of 2 .In the case N is not a power of 2 ,then enough rows and columns of zero can be added to both A and B .SO that the resulting dimension are the powers of two.
* If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of A & B.
* If n>2,Then the elements are partitioned into sub matrix n/2\*n/2..since „n‟ is a power of 2 these product can be recursively computed using the same formula

.This Algorithm will continue applying itself to smaller sub matrix until „N” become suitable small(n=2) so that the product is computed directly .

* The formula are



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A11 | A12 | B11 | B12 | C11 | C12 |
|  |  | \* |  | = |  |
| A21 | A21 | B21 | B22 | C21 | C22 |

C11 = A11 B11 + A12 B21

C12 = A11 B12 + A12 B22

C21 = A21 B11 + A22 B21

C22 = A21 B12 + A22 B22

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| For EX: |  |  |  |  |  |  |
|  | 2 2 2 2 |  | 1 | 1 | 1 1 | |
| 4 \* 4 = | 2 2 2 2 |  | 1 1 1 1 | | | |
|  | 2 2 2 2 | \* | 1 | 1 1 | | 1 |
|  | 2 2 2 2 |  | 1 | 1 1 | | 1 |



The Divide and conquer method



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 2 |  |  |  | 2 | 2 |  |  |  | 1 | 1 |  |  |  | 1 | 1 |  | 4 4 |  |  |  | 4 | 4 |  |  |
| 2 2 |  |  |  | 2 2 | |  | \* |  | 1 | 1 |  |  |  | 1 | 1 | = | 4 4 |  |  |  | 4 4 | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 2 |  |  |  | 2 | 2 |  |  |  | 1 | 1 |  |  |  | 1 | 1 |  | 4 4 |  |  |  | 4 | 4 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 2 2 |  |  |  | 2 | 2 |  |  |  | 1 | 1 |  |  |  | 1 | 1 |  | 4 4 |  |  |  | 4 | 4 |  |  |

* To compute AB using the equation we need to perform 8 multiplication of n/2\*n/2 matrix and from 4 addition of n/2\*n/2 matrix.
* Ci,j are computed using the formula in equation 4
* As can be sum P, Q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.
* The Cij are required addition 8 addition or subtraction.



T(n)= b n<=2 a &b are

7T(n/2)+an^2 n>2 constant

Finally we get T(n) =O( n ^log27)

Example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 4 |  | 4 | 4 |
|  |  | \* |  |  |
| 4 | 4 |  | 4 | 4 |
|  |  |  |  |  |

P=(4\*4)+(4+4)=64

Q=(4+4)4=32

R=4(4-4)=0

S=4(4-4)=0

T=(4+4)4=32

U=(4-4)(4+4)=0

V=(4-4)(4+4)=0

C11=(64+0-32+0)=32

C12=0+32=32

C21=32+0=32

C22=64+0-32+0=32

So the answer c(i,j) is 32 32

32 32

since n/2n/2 &matrix can be can be added in Cn for some constant C, The overall computing time T(n) of the resulting divide and conquer algorithm is given by the sequence.

|  |  |  |
| --- | --- | --- |
| T(n)= | b | n<=2 a &b are |
|  | 8T(n/2)+cn^2 | n>2 constant |
| That is T(n)=O(n^3) | |  |

\* Matrix multiplication are more expensive then the matrix addition O(n^3).We can attempt to reformulate the equation for Cij so as to have fewer multiplication and possibly more addition .

* Stressen has discovered a way to compute the Cij of equation (2) using only 7 multiplication and 18 addition or subtraction.
* Strassen‟s formula are

P= (A11+A12)(B11+B22)

Q= (A12+A22)B11

R= A11(B12-B22)

S= A22(B21-B11)

T= (A11+A12)B22

U= (A21-A11)(B11+B12)

V= (A12-A22)(B21+B22)

C11=P+S-T+V

C!2=R+t

C21=Q+T

C22=P+R-Q+V

**UNIT-II**

**Searching And Traversal Techniques**

# [Non recursive binary tree traversal - inorder, preorder and postorder](http://crackprogramming.blogspot.com/2012/11/non-recursive-binary-tree-traversal.html)

We learnt the basics about Binary Search tree in the [last](http://crackprogramming.blogspot.com/2012/11/binary-search-tree-insertion-search.html) post. We are now going to build on top of it. In the last post we did recursive traversal of the BST which was fairly simple. Let's now do non recursive traversal of the BST. These algorithm will help us solving a lot of other interesting problem. We will add following four functions to the class BST. In order to compile it you will need to add below lines to the BST class declaration:

void inorder\_non\_recursive(NODE<T>\* nodePtr);

void preorder\_non\_recursive(NODE<T>\* nodePtr);

void postorder\_non\_recursive(NODE<T>\* nodePtr);

void postorder\_non\_recursive\_1(NODE<T>\* nodePtr);

The exact implementation of these functions goes like this:

/\*Inorder tree traversal non recursive.\*/

template<typename T> void BST<T>::inorder\_non\_recursive(NODE<T>\* nodePtr)

{

cout << endl;

stack<NODE<T>\*> s;

while (!s.empty() || nodePtr)

{

/\*If nodePtr is NULL, we need to pop from top of the stack

and print it's value. Then traverse the right sub-tree\*/

if (!nodePtr)

{

nodePtr = s.top();

cout << nodePtr->data << "\t";

nodePtr = nodePtr->right;

s.pop();

}

/\*Keep traversing left unless nodePtr is NULL\*/

if (nodePtr)

{

s.push(nodePtr);

nodePtr = nodePtr->left;

}

}

cout << endl;

return;

}

/\*Non recursive preorder tree traversal\*/

template<typename T> void BST<T>::preorder\_non\_recursive(NODE<T>\* nodePtr)

{

cout << endl;

stack<NODE<T>\*> s;

/\*Root is NULL, nothing to do.\*/

if (!nodePtr)

return;

/\*Push root\*/

s.push(nodePtr);

while (!s.empty())

{

/\*Pop the top element\*/

NODE<T>\* tmpPtr = s.top();

cout << tmpPtr->data << "\t";

s.pop();

/\*Go right before going left as we are using a stack.\*/

if (tmpPtr->right)

s.push(tmpPtr->right);

if (tmpPtr->left)

s.push(tmpPtr->left);

}

cout << endl;

return;

}

/\*Non recursive postorder tree traversal. This is perhaps the most

complicated one too. Let's look at an easier solution using two

stacks first.\*/

template<typename T> void BST<T>::postorder\_non\_recursive(NODE<T>\* nodePtr)

{

cout << endl;

if (!nodePtr)

return;

stack<NODE<T>\*> s1;

stack<NODE<T>\*> s2;

/\*Push root into stack s1\*/

s1.push(nodePtr);

while (!s1.empty())

{

/\*Pop fropm sack s1 and push it to stack s2\*/

NODE<T>\* tmpPtr = s1.top();

s2.push(tmpPtr);

s1.pop();

/\*Now push left child before right. Because when we push it back to

stack s2, the order will be correct.\*/

if (tmpPtr->left)

s1.push(tmpPtr->left);

if (tmpPtr->right)

s1.push(tmpPtr->right);

}

/\*Now s2 has got all the nodes in postorder. We will just pop it one by

one and print.\*/

while (!s2.empty())

{

NODE<T>\* tmpPtr = s2.top();

s2.pop();

cout << tmpPtr->data << "\t";

}

cout << endl;

return;

}

/\*Non recursive postorder tree trraversal using only one stack.\*/

template<typename T> void BST<T>::postorder\_non\_recursive\_1(NODE<T>\* nodePtr)

{

if (!nodePtr)

return;

cout << endl;

stack<NODE<T>\*> s;

/\*We will need current poionter to the node we are currently

traversing and the pointer to the node we traversed previously.\*/

NODE<T>\* current = nodePtr;

NODE<T>\* previous = NULL;

s.push(current);

while (!s.empty())

{

current = s.top();

/\*Traverse the tree down\*/

if (!previous || previous->left == current || previous->right == current)

{

if (current->left)

s.push(current->left);

else if (current->right)

s.push(current->right);

else

{

cout << current->data << "\t";

s.pop();

}

}

/\*Traverse the tree up from the left\*/

else if (current->left == previous)

{

if (current->right)

s.push(current->right);

else

{

cout << current->data << "\t";

s.pop();

}

}

else if (current->right == previous)

{

cout << current->data << "\t";

s.pop();

}

previous = current;

}

cout << endl;

return;

}

**Complexity of the code**  
All the tree traversal logic takes O(N) time because all nodes has to be touched  at most once. Also an additional space of O(h) is needed for stack where 'h' is the maximum height/depth of the tree. The two stack solution for post order tree traversal has additional space complexity of O(N) where N is the number of nodes in the tree.

**Basic Traversal and Search Techniques**

Traversal vs Search

Definition 1 Traversal of a binary tree involves examining every node in the tree.

Definition 2 Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes.

Different nodes of a graph may be visited, possibly more than once, during traversal or search If search results into a visit to all the vertices, it is called traversal

**Techniques for binary trees**

Determine a vertex or a subset of vertices that satisfy a specified property

– Possible problem: Find all nodes in a binary tree with data value less than some specified value

Solved by systematically examining all the vertices

Does searching for a specified item in a binary search tree result into a traversal? Traversal produces a linear order for the information in a tree

– During traversal, treat each node in the binary tree and its subtrees in the same manner Inorder, preorder, and postorder traversals

struct tree\_node\_type

{

item\_type data; // To hold information at each node

tree\_node\_type left\_child; tree\_node\_type right\_child;

};

Algorithm inorder ( t )

* Input: t is a binary tree
* Each node of t is of type tree\_node\_type

{

if ( t != NULL )

{

|  |  |  |  |
| --- | --- | --- | --- |
| inorder ( | | t.left\_child ); | |
| visit ( | t | ); | // Perform a function on data in node being traversed |
| inorder | ( | t.right\_child ); | |

}

}

Algorithm preorder ( t )

* Input: t is a binary tree
* Each node of t is of type tree\_node\_type

{

if ( t != NULL )

{

visit ( t ); // Perform a function on data in node being traversed

|  |  |
| --- | --- |
|  |  |

preorder ( t.left\_child ); preorder ( t.right\_child );

}

}

Algorithm postorder ( t )

* Input: t is a binary tree
* Each node of t is of type tree\_node\_type

{

if ( t != NULL )

{

postorder ( t.left\_child ); postorder ( t.right\_child );

visit ( t ); // Perform a function on data in node being traversed

}

}

Theorem 1 Let Tn and Sn represent the time and space needed by any one of the traversal algorithms when the input tree t has n 0 nodes. If the time and space needed to visit a single node is (1), then Tn = (n) and Sn = O(n).

Proof:

– Each node in the tree is visited three times, requiring constant amount of work

* 1. From parent (or start node, if root)
  2. Return from left subtree
  3. Return from right subtree

– This gives the total time required for traversal to be (n)

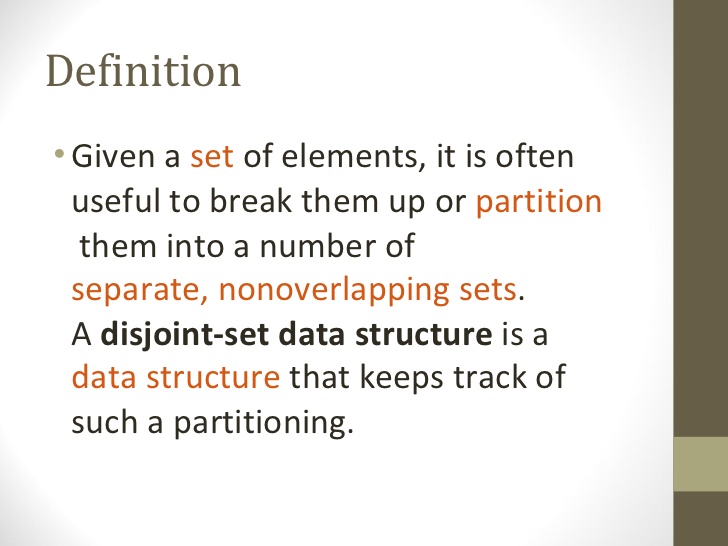
– Additional space is required for recursion stack

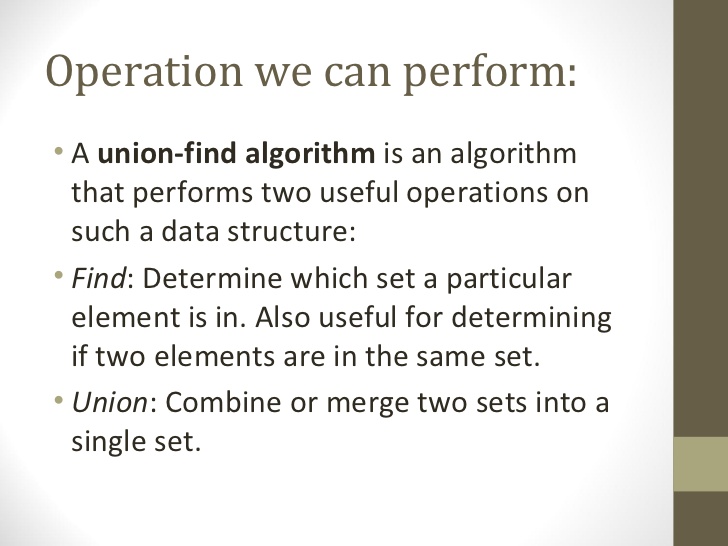
If t has depth d, this space is given by (d) For an n-node binary tree, d n

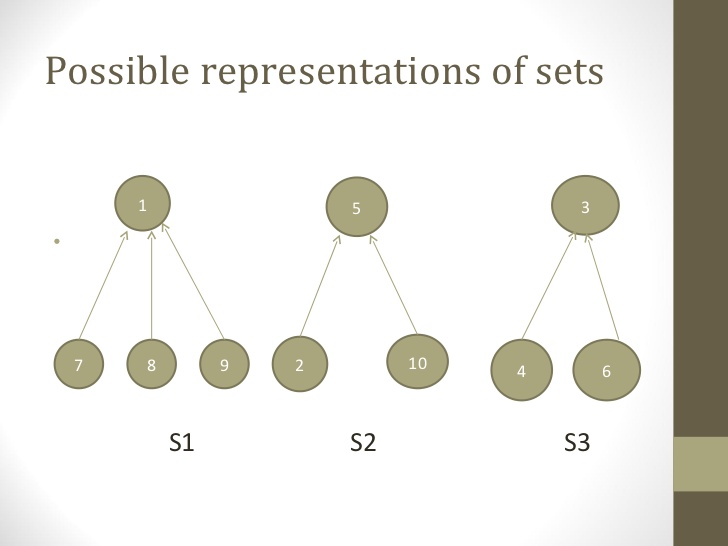
Hence, Sn = O(n)

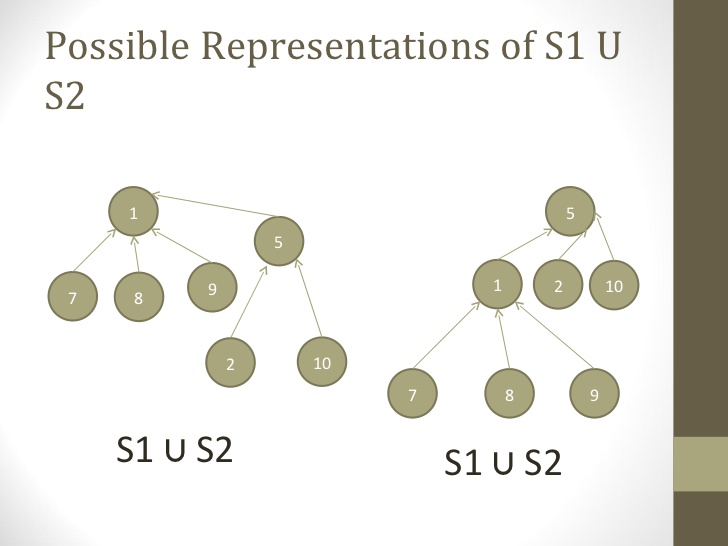
Level-order traversal .

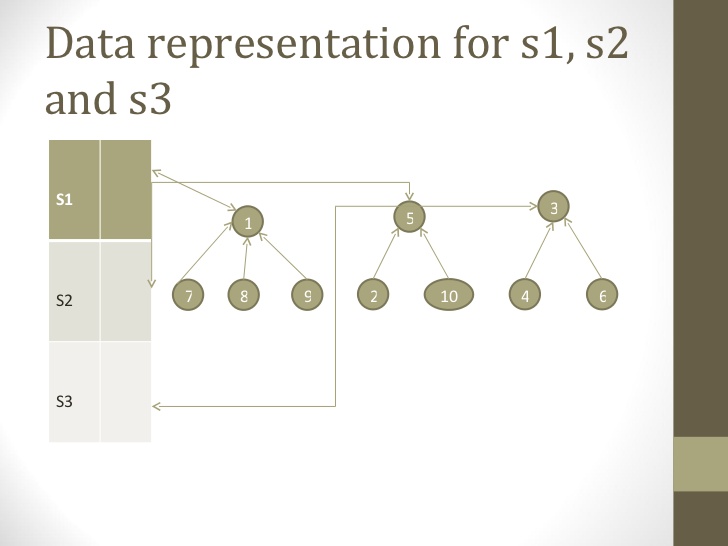
**SETS AND DISJOINT SETS**

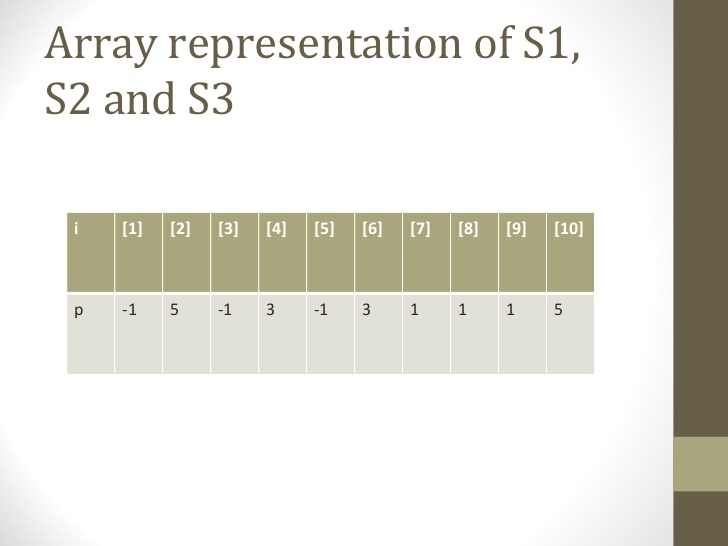








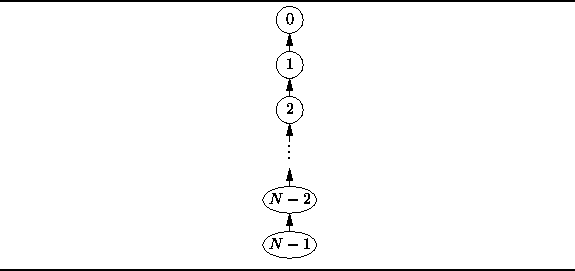




## Simple algorithms for union and find <ul><li>Algorithm SimpleUnion(i,j) </li></ul><ul><li>{ </li></ul><ul><li>P[i]:=j; </l...

## Collapsing Rule For Find

  Unfortunately, using the join algorithm given in Program [gif](http://www.brpreiss.com/books/opus5/html/page407.html#progPartitionAsForestd) can result in particularly bad trees. For example, Figure [gif](http://www.brpreiss.com/books/opus5/html/page408.html#figsets4) shows the worst possible tree that can be obtained. Such a tree is bad because its height is *O*(*N*). In such a tree both the worst case and the average case running time for the find operation is *O*(*N*).



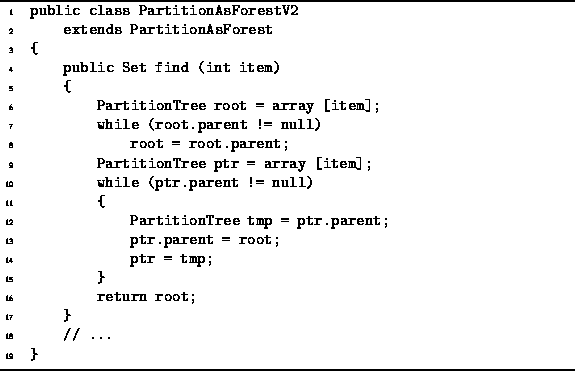
**Figure:** A degenerate tree.

There is an interesting trick we can play that can improve matters significantly. Recall that the find operation starts from a given node and locates the root of the tree containing that node. If, having found the root, we replace the parent of the given node with the root, the next time we do a find it will be more efficient.

In fact, we can go one step further and replace the parent of every node along the search path to the root. This is called a collapsing find   operation. Doing so does not change the asymptotic complexity of the find operation. However, a subsequent find operation which begins at any point along the search path to the root will run in constant time!

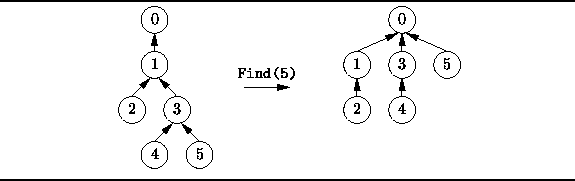
Program [gif](http://www.brpreiss.com/books/opus5/html/page408.html#progPartitionAsForestV2a) gives the code for a collapsing version of the find operation. The find method first determines the root node as before. Then, a second pass is made up the chain from the initial node to the root, during which the parent of each node is assigned the root. Clearly, this version of find is slower than the one given in Program [gif](http://www.brpreiss.com/books/opus5/html/page407.html#progPartitionAsForestc) because it makes two passes up

the chain rather than one. However, the running of this version of find is still *O*(*d*), where *d* is the depth of the node from which the search begins.

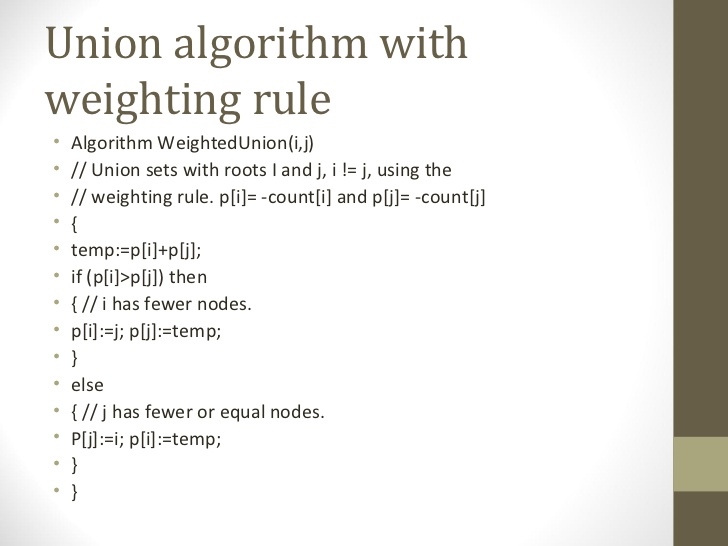
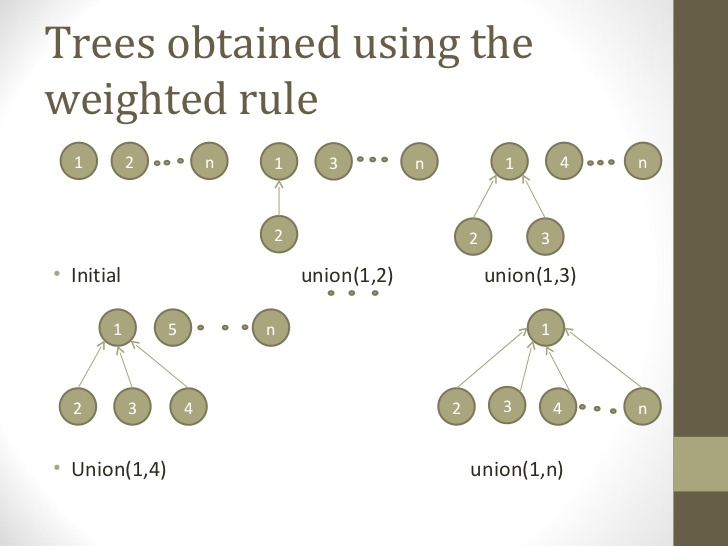


**Program:** PartitionAsForest class collapsing find method.

Figure [gif](http://www.brpreiss.com/books/opus5/html/page408.html#figsets5) illustrates the effect of a collapsing find operation. After the find, all the nodes along the search path are attached directly to the root. That is, they have had their depths decreased to one. As a side-effect, any node which is in the subtree of a node along the search path may have its depth decreased by the collapsing find operation. The depth of a node is never increased by the find operation. Eventually, if we do enough collapsing find operations, it is possible to obtain a tree of height one in which all the non-root nodes point directly at the root.



**Figure:** Example of collapsing find.



**Spanning trees**

A *spanning tree* of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees; for instance the complete graph on four vertices

o---o

|\ /|

| X |

|/ \|

o---o

has sixteen spanning trees:

o---o o---o o o o---o

| | | | | |

| | | | | |

| | | | | |

o o o---o o---o o---o

o---o o o o o o o

\ / |\ / \ / \ /|

X | X X X |

/ \ |/ \ / \ / \|

o o o o o---o o o

o o o---o o o o---o

|\ | / | /| \

| \ | / | / | \

| \| / |/ | \

o o o---o o o o---o

o---o o o o o o---o

|\ | / \ | /|

| \ | / \ | / |

| \ |/ \| / |

o o o---o o---o o o

## [Graph Traversal](http://web.cse.ohio-state.edu/%7Egurari/course/cis680/cis6802.html#QQ2-46-90)

Vertex v is at this time said to be unexplored. A vertex is said to have been explored by an

algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices

adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been

explored. The newly visited vertices haven’t been explored and are put onto the end of a list of

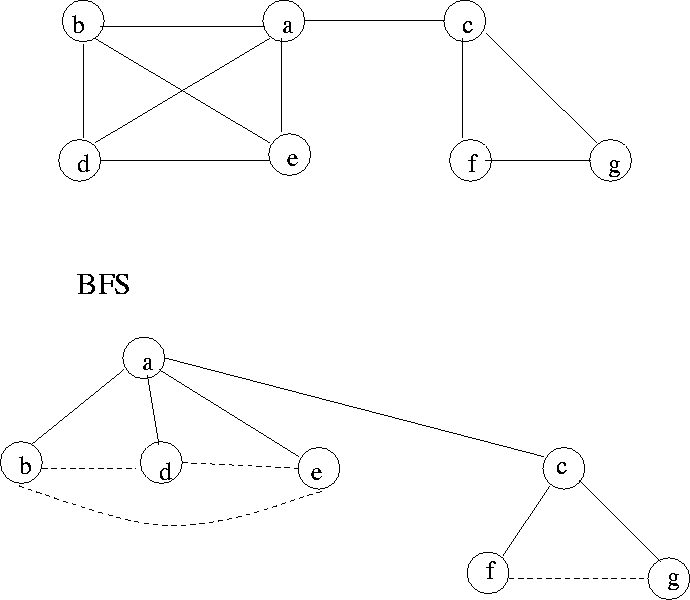
unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues

until no unexplored vertex is left.

**Example:** See Figure

Assume the adjacency lists:

|  |  |
| --- | --- |
| **Vertex** | **Adj. List** |
|  |  |
| a | (b, c, d, e) |
| b | (a, d, e) |
| c | (a, f, g) |
| d | (a, b, e) |
| e | (a, b, d) |
| f | (c, g) |
| g | (c, f) |



**Breadth first search**

We start at a vertex V and mark it as have been reached. The vertex v is at this time said to be unexplorted. All visited vertices adjacent from v are visited next.

If G is represented by its adjacent then the time is O(n2).

Algorithm BFS(v)

{

u := v;

visited[v] := 1;

repeat

{

for all vertices w adjacent from u do

{

if (visited[w] = 0) then

{

add w to q;

visited[w] := 1;

}

}

if q is empty then return;

delet u from q;

} until (false)

}

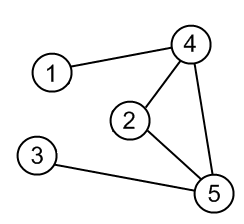
A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached.

At this time the exploration of the new vertex u begins. When the new vertex has been explored, the exploration of v continues.

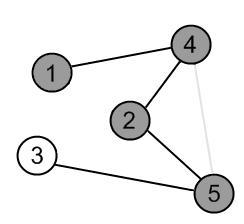
# Depth-first search (DFS) for undirected graphs

**Depth-first search**, or **DFS**, is a way to traverse the graph. Initially it allows visiting vertices of the graph only, but there are hundreds of algorithms for graphs, which are based on DFS. Therefore, understanding the principles of depth-first search is quite important to move ahead into the graph theory. The principle of the algorithm is quite simple: to go forward (in depth) while there is such possibility, otherwise to backtrack.

**Example**



**Source graph**



The exploration of a vertex V is suspended as soon as a new vertex is reached. Algorithm DFS(v)

**visited[v Algoithm DFS(v)**

**{**

**]:=1;**

**for each vertex q adjacent from v do**

**{**

**if (visited[w] =0 ) then DFS(w);**

**}**

**}**

### And-Or Graphs

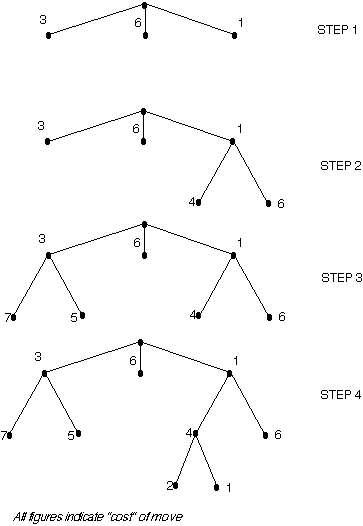
Useful for certain problems where

* The solution involves decomposing the problem into smaller problems.
* We then solve these smaller problems.

Here the alternatives often involve branches where some or all must be satisfied before we can progress.

For example if I want to learn to play a Frank Zappa guitar solo I could (Fig. [2.2.1](http://www.cs.cf.ac.uk/Dave/AI2/node25.html#figandor_tree))

* Transcribe it from the CD. **OR**
* Buy the ``Frank Zappa Guitar Book'' **AND** Read it from there.



**Note** the use of arcs to indicate that one or more nodes must all be satisfied before the parent node is achieved. To find solutions using an And-Or GRAPH the best first algorithm is used as a basis with a modification to handle the set of nodes linked by the AND factor.

***AO*\* Algorithm**

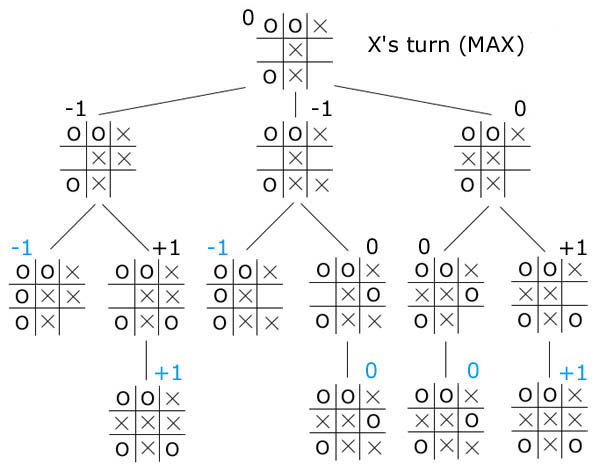
1. Initialise the graph to start node
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
3. Pick any of these nodes and expand it and if it has no successors call this value *FUTILITY* otherwise calculate only *f*' for each of the successors.
4. If *f*' is 0 then mark the node as *SOLVED*
5. Change the value of *f*' for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as *SOLVED* then mark the parent node as *SOLVED*.
7. If starting node is *SOLVED* or value greater than *FUTILITY*, stop, else repeat from 2.

**Game trees**

Consider the problem of implementing a computer program to play a game. To simplify things a bit, we will only consider games with the following two properties:

* Two player - we do not deal with coalitions, etc.
* Zero sum - one player's win is the other's loss; there are no cooperative victories

Examples of these kinds of games include many classic board games, such as tic tac toe, chess, checkers, and go. For these types of games, we can model the game using what is called a *game tree*:



Above is a section of a game tree for tic tac toe. Each node represents a board position, and the children of each node are the legal moves from that position. To score each position, we will give each position which is favorable for player 1 a positive number (the more positive, the more favorable). Similarly, we will give each position which is favorable for player 2 a negative number (the more negative, the more favorable). In our tic tac toe example, player 1 is 'X', player 2 is 'O', and the only three scores we will have are +1 for a win by 'X', -1 for a win by 'O', and 0 for a draw. Note here that the blue scores are the only ones that can be computed by looking at the current position. To calculate the scores for the other positions, we must look ahead a few moves, perhaps by using one of the algorithms below.   
  
 Now that we have a way of representing the game in our program, how do we compute our optimal move? We will assume that the opponent is rational; that is, the opponent can compute moves just as well as we can, and the opponent will always choose the optimal move with the assumption that we, too, will play perfectly. (Contrast this, for example, to the beginning chess player, who will purposely make a move with a trap, in the hopes of catching the opponent into the trap and gaining a quick victory. However, if the opponent does not fall for the trap, our player finds that his position is now critically weakened).   
 One algorithm for computing the best move is the *minimax* algorithm:

minimax(player,board)

if(game over in current board position)

return winner

children = all legal moves for player from this board

if(max's turn)

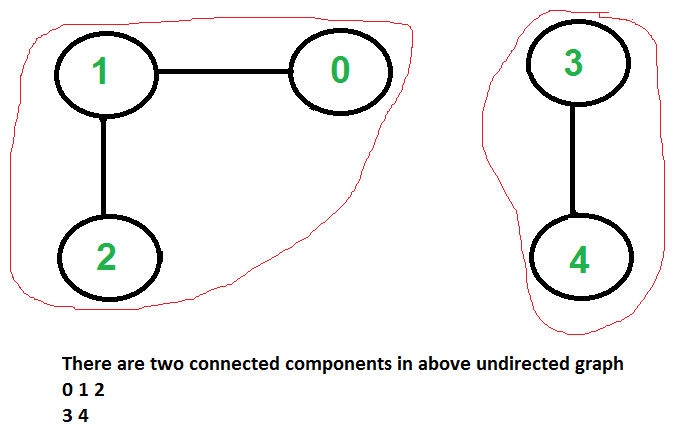
return maximal score of calling minimax on all the children

else (min's turn)

return minimal score of calling minimax on all the children.

# Connected Components in an undirected graph

Given an undirected graph, print all connected components line by line. For example consider the following graph.



**We strongly recommend to minimize your browser and try this yourself first.**

We have discussed algorithms for finding strongly connected components in directed graphs in following posts.  
 Finding connected components for an undirected graph is an easier task. We simple need to do either BFS or DFS starting from every unvisited vertex, and we get all strongly connected components. Below are steps based on DFS.

1) Initialize all vertices as not visited.

2) Do following for every vertex 'v'.

(a) If 'v' is not visited before, call DFSUtil(v)

(b) Print new line character

DFSUtil(v)

1) Mark 'v' as visited.

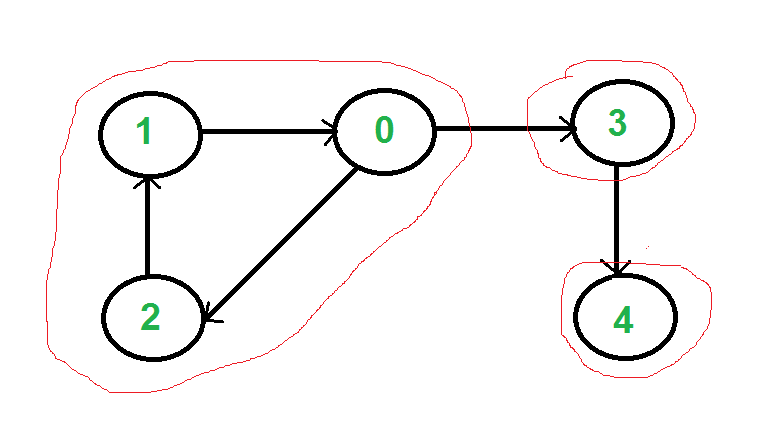
2) Print 'v'

3) Do following for every adjacent 'u' of 'v'.

If 'u' is not visited, then recursively call DFSUtil(u).

# Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



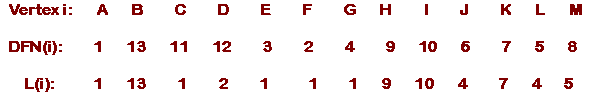
We can find all strongly connected components in O(V+E) time using [Kosaraju’s algorithm](http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm). Following is detailed Kosaraju’s algorithm.  
**1)** Create an empty stack ‘S’ and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.  
**2)** Reverse directions of all arcs to obtain the transpose graph.  
**3)** One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘v’. Take v as source and do DFS (call [DFSUtil(v)](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)). The DFS starting from v prints strongly connected component of v.

Biconnected Graph

|  |  |
| --- | --- |
| **[Basic Concepts]**   * ***Articulation point:* An Articulation point in a connected graph is a vertex that, if delete, would break the graph into two or more pieces (connected component).** * ***Biconnected graph:* A graph with no articulation point called biconnected. In other words, a graph is biconnected if and only if any vertex is deleted, the graph remains connected.** * ***Biconnected component:* A biconnected component of a graph is a maximal biconnected subgraph- a biconnected subgraph that is not properly contained in a larger biconnected subgraph.** * **A graph that is not biconnected can divide into biconnected components, sets of nodes mutually accessible via two distinct paths.**   **[Example] Graph G in Figure 1:**   * **Articulation points: A, H, G, J** * **Biconnected components: {A, C, G, D, E, F}、{G, J, L, B}、B、H、I、K.**   **How to find articulation points?** | *The graphs we discuss below are all about loop-free undirected ones.*  image1.jpg (29545 bytes)  **Figure 1. The graph G that is not biconnected** |

|  |  |  |
| --- | --- | --- |
| Image4.gif (7027 bytes)  **Figure 2. Depth-first panning tree of the graph G** | **[Step 1.]** | **Find the depth-first spanning tree T for G** |
| **[Step 2.]** | **Add back edges in T** |
| **[Step 3.]** | **Determine DNF(i) and L(i)**   **DNF(i): the visiting sequence of vertices i by depth first search**   **L(i): the least DFN reachable frome i through a path consisting of zero or more tree edges followed by zero or one back edge** |
| **[Step 4.]** | **Vertex i is an articulation point of G if and only if eather:**   **i is the root of T and has at least two children**   **i is not the root and has a child j for which L(j)>=DFN(i)** |

**[Example] The DFN(i) and L(i) of Graph G in Figure 1 are:**

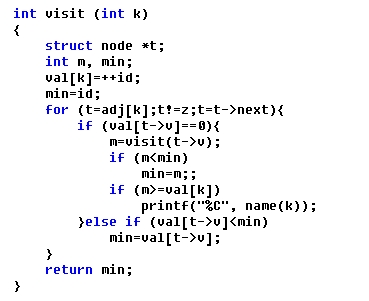


 **Vertex G is an articulation point because G is not the root and in depth-first spanning tree in Figure 2, L(L)>=DFN(G), that L is one of its children**

 **Vertex A is an articulation point because A is the root and in depth-first spanning tree in Figure 2, it has more than one child, B and F**

 **Vertex E is not an articulation point because E is not the root and in depth-first spanning tree in Figure 2, L(G)<DFN(E) and L(D)<DFN(E), that G and D are its children**

**[Program]**



**UNIT-III**

**GREEDY METHOD**

It is used to solve problems that have ‘n’ inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfied the constraints is called as a feasible solution. We need to find the optimum feasible solution i.e. the feasible solution that optimizes the given objective functions.

The greedy method suggests that one can divide the algorithm that works in stages. Considering one input at a time. At each stage a decision is made regarding weather or particular input is in the optimum solution. For this purpose all the inputs must be arranged in a particular order by using a selection process. If the inclusion of next input in to the partially constructed solution will result in an infeasible solution then the input will not be considered and will not be added to partially constructed set otherwise it is added.

**Feasible Solution:-**

Any subset of the solutions that satisfies the constraints of the problem is known as a feasible solution.

**Optimal Solution:-**

The feasible solution that maximizes or minimizes the given objective function is an optimal solution. Every problem will have a unique optimal solution.

**CONTROL ABSTRACTION FOR GREEDY METHOD**: -

Algorithm Greedy (a,n)

//a[1:n] contains the ‘n’ inputs

{ Solution: = 0; //initialize the solution

for i: = 1 to n do

{ x: = select (a);

if feasible 9solution, x) then

solution: = Union (solution, x);

}

return solution;

}

Select is a function which is used to select an input from the set. Feasible is a function which verifies the constraints and determines weather the resultant solution is feasible or not. Union is a function which is used to add elements to the partially constructed set.

**Applications of Greedy methods**

1.

2.

3.

4.

**JOB SEQUENCING WITH DEADLINES**: -

**Problem: -**

We are given a set of ‘n’ jobs. Associated with each job there is a integer dead line di≥0 and a profit pi>0. For any job i the profit pi is earned if and only if the job is completed by its dead line. To complete a job one has to process the job on a machine for one unit of time. Only one machine is available for processing the jobs. A feasible solution for the problem will be a subset ‘j’ of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution ‘j’ is the sine of the profits of the jobs in ‘j’. An optimum solution is a feasible solution with maximum value.

The problem involves identification of a subset of jobs which can be completed by its deadline. Therefore the problem suites the subset methodology and can be solved by the greedy method.

Ex: - Obtain the optimal sequence for the following jobs.

j1 j2 j3 j4

(P1, P2, P3, P4) = (100, 10, 15, 27)

(d1, d2, d3, d4) = (2, 1, 2, 1)

n =4

|  |  |  |
| --- | --- | --- |
| **Feasible soln** | **Processing sequence** | **Value** |
| j1 j2  (1, 2) | (2,1) | 100+10=110 |
| (1,3) | (1,3) or (3,1) | 100+15=115 |
| (1,4) | (4,1) | 100+27=127 |
| (2,3) | (2,3) | 10+15=25 |
| (3,4) | (4,3) | 15+27=42 |
| (1) | (1) | 100 |
| (2) | (2) | 10 |
| (3) | (3) | 15 |
| (4) | (4) | 27 |

In the example solution ‘3’ is the optimal. In this solution only jobs 1&4 are processed and the value is 127. These jobs must be processed in the order j4 followed by j1. the process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time2. Therefore both the jobs are completed within their deadlines. The optimization measure for determining the next job to be selected in to the solution is according to the profit. The next job to include is that which increases ∑pi the most, subject to the constraint that the resulting “j” is the feasible solution. Therefore the greedy strategy is to consider the jobs in decreasing order of profits.

**Algorithm Greedy Job (d,j,n)**

//j is a set of jobs that can be completed by

// their dead lines

{ j: = {1};

for i: = 2 to n do

{ if (all jobs in j u {i} can be completed by their dead lines) then

j: = ju{ i};

}

}

**Problem:-**

Find the optimum job sequence for the following problem.

n = 7 P7  J1 j2  j3  j4 j5 j6 j7

(P1, P2, P3, P4, P5, P6) = (3, 5, 10, 18, 1, 6, 30)

d7

(d1, d2, d3, d4, d5, d6) = (1,3, 4, 3, 2, 1, 2)

**Feasible soln. Processing Sequence Value**

(1, 2, 3, 4) (1,2,4,3) or (1,4,2,3) 3+5+10+18= 36

**KNAPSACK PROBLEM**: -

A knapsack with capacity ‘m’ is given in to which we are required to place certain weights such that the sum of the weights will not exceed the knapsack capacity. Associated with each weight we have associated profit which will be earned by the inclusion of the object in to the knapsack.

If it is not possible to include an object entirely a fraction of the object can be included and accordingly a fraction of the profit is earned.

Given ‘n’ objects and a bag of capacity ‘n’ and each object ‘i’ has a profit pi and weight wi associated with it. If a fraction xj (o≤ x, ≤ 1) of the object i is placed in the bad. A profit pi x xi is made. The objective in the problem is to obtain the maximum profit by filling the bag with given objects.

The knapsack problem can be stated mathematically as follows.

Pi xi Maximize ∑ Pi xi

1 ≤ i ≤ n

S.T.C

∑ Pi xi ≤ m

1 ≤ u ≤ n

o ≤ xi ≤ 1

1 ≤ i ≤ n

Ex: - Consider 3 objects whose profits and weights are defined as

(P1, P2, P3) = ( 25, 24, 15 )

(W1, W2, W3) = ( 18, 15, 10 )

n=3 m=20

Consider a knapsack of capasing 20. Determine the optimum strategy for placing the objects in to the knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

**(1) Greedy about from lorgest profit**: -

(x1, x2, x3) ∑ xiwi ∑ xipi

(1, 2/15, D) 18x1+x15 25x1+x~~24~~ 8

= 20 25+ = 28.2

**(2) Greedy about from lowest weight**: -

5

(0, 2/3, 1) x~~15~~+10x1= 20 15x1+x~~24~~ 8 = 31

**(3) Greedy about maximum ratio of profit / weight:** -

(0, 1, ½ ) 1x15+x10 = 20 1x24+x15 = 31.5

**The (3) method is optimal solution remaining all are feasible solutions.**

**Algorithm Greedy knapsack (m, n)**

//p (1>n) and w(1:n) contain the profits and weights

//resp. of the n objects ordered such that p(i)/W(i)

//≥ p[i+1)/w (i+1]. M is the knapsack size and x (1:n)

// is the solution vector

{

for i: = 1 to n do x(i) = 0.0i// initialize x

u : = m;

for i: = 1 to n do

{ if (w(i) > u) then break;

x(i): = 1.0; u: = u-w(i);

}

if (i≤n) then x(i): = u/w(i);

}

**Analysis**: - If. we do not consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be O(n).

**SPANNING TREE**: - A Sub graph ‘n’ of o graph ‘G’ is called as a spanning tree if

1. It includes all the vertices of ‘G’
2. It is a tree

For a given graph ‘G’ there can be more than one spanning tree. If weights are assigned to the edges of ‘G’ then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.

The greedy method suggests that a minimum cost spanning tree can be obtained by contracting the tree edge by edge. The next edge to be included in the tree is the edge that results in a minimum increase in the some of the costs of the edges included so far.

**PRIM’S ALGORITHM**: -

1. Select an edge with minimum cost and include in to the spanning tree.
2. Among all the edges which are adjacent with the selected edge, select the one with minimum cost.
3. Repeat step 2 until ‘n’ vertices and (n-1) edges are been included. And the sub graph obtained does not contain any cycles.

***Notes:*** - At every state a decision is made about an edge of mi9nimum cost to be included into the spanning tree. From the edges which are adjacent to the last edge included in the spanning tree i.e. at every stage the sub-graph obtained is a tree.

**28**

**10**

**10 11 16**

**11 16**

**18 24**

**25 12 25 12**

**22 22**

**total weight=90**

Algorithm Prim (E, cost, n,t)

// E is the set of edges in G. Cost (1:n, 1:n) is the

// Cost adjacency matrix of an n vertex graph such that

// Cost (i,j) is either a positive real no. or q if no edge

// (i,j) exists. A minimum spanning tree is computed and

// Stored in the array + (1:n-1), 1:2). (t (i, 1), + (i,2)

// is an edge in the minimum cost spanning tree

// The final cost is returned

{

Let (k, l) be an edge with min cost in E

Min cost: = Cost (x,l);

T(1,1):= k; + (1,2):= l;

for i:= 1 to n do//initialize near

if (cost (i,l)<cost (i,k) then n east (i): l;

else near (i): = k;

near (k): = near (l): = 0;

for i: = 2 to n-1 do

{//find n-2 additional edges for t

let j be an index such that near (i) ≠0 & cost (j, near (i)) is minimum;

t (i,1): = j + (i,2): = near (j);

min cost: = Min cost + cost (j, near (j));

near (j): = 0;

for k:=1 to n do // update near ()

if ((near (k) ≠0) and (cost {k, near (k)) > cost (k,j)))

then near Z(k): = ji

}

return mincost;

}

**Analysis**: -

The time required by the prince algorithm is directly proportional to the no/: of vertices. If a graph ‘G’ has ‘n’ vertices then the time required by prim’s algorithm is 0(n2).

**KRUSKAL’S ALGORITHM**: -

In Kruskals algorithm for determining the spanning tree we arrange the edges in the increasing order of cost.

1. All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.
2. At every stage an edge is included; the sub-graph at a stage need not be a tree. Infact it is a forest.
3. At the end if we include ‘n’ vertices and n-1 edges without forming cycles then we get a single connected component without any cycles i.e. a tree with minimum cost.

At every stage, as we include an edge in to the spanning tree, we get disconnected trees represented by various sets. While including an edge in to the spanning tree we need to check it does not form cycle. Inclusion of an edge (i,j) will form a cycle if i,j both are in same set. Otherwise the edge can be included into the spanning tree.

Ex: - Let the partition be {1,2} {3,4,6} {5}

I we want to include the edge (1,4) it can be included as land 4 are in different sets.

So we get (1,2,3,4,6) (5) subsets respectively. If we want to include the edge between (2,6) because 2&6 are in the same subsets and its i9nclusion will lead to a cycle.

**Min spanning Tree**;

6 1

1 11

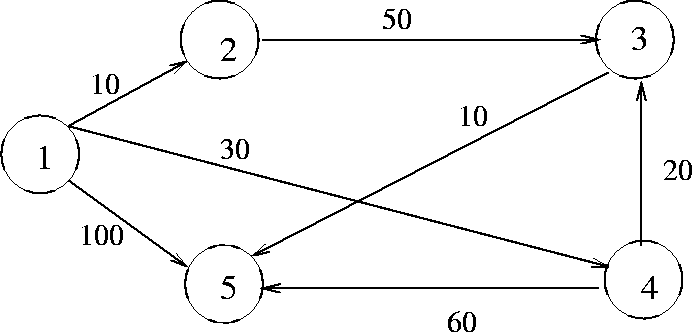
7 3 2

3 8

10 9 4 5

4 5

**SINGLE SOURCE SHORTEST PATH**: -( **Dijkstra's Algorithm**)



**Initially:**

*S* = {1}  *D*[2] = 10  *D*[3] = $\displaystyle \infty$ *D*[4] = 30  *D*[5] = 100

**Iteration 1**

Select w = 2, so that S = {1, 2}

|  |  |  |  |
| --- | --- | --- | --- |
| *D*[3] | = | min($\displaystyle \infty$, *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] | = | 60 |  |

**Complexity of Dijkstra's Algorithm**

With adjacency matrix representation, the running time is O(n2) By using an adjacency list representation and a partially ordered tree data structure for organizing the set V - S, the complexity can be shown to be

O(elog n)

where e is the number of edges and n is the number of vertices in the digraph.

**Algorithm shortest paths (v, cost, dist, n)**

//dist (j), 1≤ j ≤ n, is set to the length of shortest path

//from vertex v to vertex j in a diagraph G with n vertices

//dist(v) is set to zero. G is represented by its cost

//adjacency matrix cost (1:n, 1:n)

{ for i = 1 to n do

{ // initializes

s(i) = false;

dist(i) = cost (v,i);

}

s(v)= tree; dist (v) = 0.0; // put vins

for num: = 2 to n-1 do

{ determine n-1 paths from v choose u from among those vertices not in s.

Such that dist (u) is min.

S(u):= true; //put u ins

For (each u adjancent to u with s(w)= false)do

// update distances

if (dist (w) > dist (u) + cost (u,w)) then

dist (w): = dist (u) + Cost (u,w);

}

**DYNAMIC PROGRAMING**

* The idea of dynamic programming is thus quit simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up a sub instances are solved.
* Divide and conquer is a top-down method.
* When a problem is solved by divide and conquer, we immediately attack the complete instance, which we then divide into smaller and smaller sub-instances as the algorithm progresses.
* Dynamic programming on the other hand is a bottom-up technique.
* We usually start with the smallest and hence the simplest sub- instances.
* By combining their solutions, we obtain the answers to sub-instances of increasing size, until finally we arrive at the solution of the original instances.
* The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated.
* In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences can not be optimal and so will not be generated.

**MULTISTAGE GRAPH**

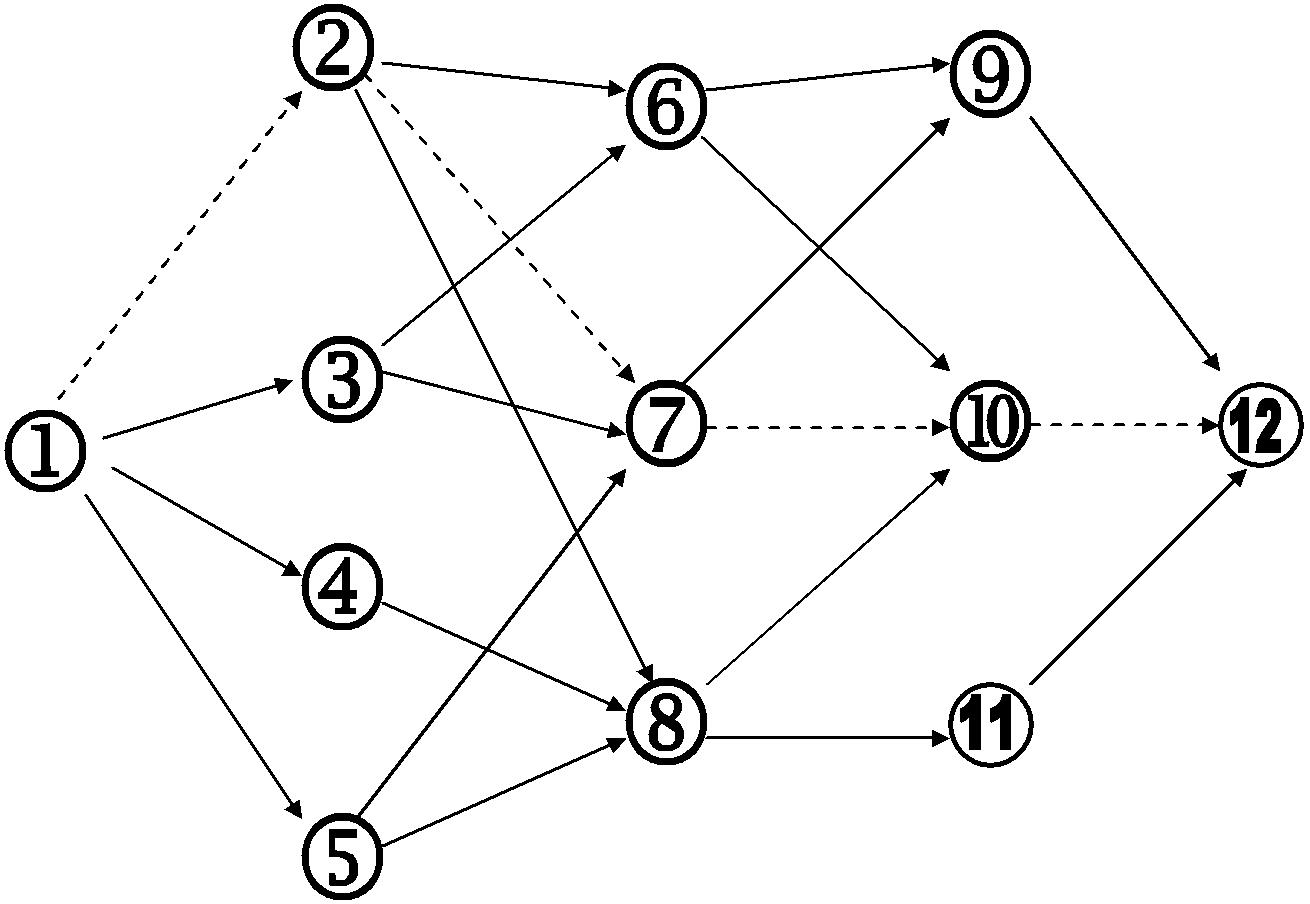
1. A multistage graph G = (V,E) is a directed graph in which the vertices are portioned into K
   * = 2 disjoint sets Vi, 1 <= i<= k.
2. In addition, if < u,v > is an edge in E, then u < = Vi and V  Vi+1 for some i, 1<= i < k.
3. If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.
4. Let ‘s’ and ‘t’ be the source and destination respectively.
5. The cost of a path from source (s) to destination (t) is the sum of the costs of the edger on the path.
6. The *MULTISTAGE GRAPH* problem is to find a minimum cost path from ‘s’ to ‘t’.
7. Each set Vi defines a stage in the graph. Every path from ‘s’ to ‘t’ starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.
8. This *MULISTAGE GRAPH* problem can be solved in 2 ways.
   * 1. Forward Method.
     2. Backward Method.

**FORWARD METHOD**

1. Assume that there are ‘k’ stages in a graph.
2. In this *FORWARD* approach, we will find out the cost of each and every node starling from the ‘k’ th stage to the 1st stage.
3. We will find out the path (i.e.) minimum cost path from source to the destination (ie) [ Stage-1 to Stage-k ].

**PROCEDURE:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| V1 | V2 | V3 | V4 | V5 |
|  | 4 | 6 |  |  |
|  | 2 | 2 |  |  |
|  |  | 5 |  | 4 |
| 9 | 1 |  |  |  |
|  |  | 4 |  |  |
|  | 7 | 3 |  | 2 |
|  | 7 |  |  | t |
| s |  |  |  |  |
|  | 3 |  |  |  |
|  | 11 |  |  | 5 |
|  | 2 |  |  |  |
|  | 11 | 6 |  |  |



8

* Maintain a cost matrix cost (n) which stores the distance from any vertex to the destination.
* If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path, will be stored in the distance array ‘D’.
* In this way we will find out the minimum cost path from each and every vertex.
* Finally cost(1) will give the shortest distance from source to destination.
* For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn give the next nearest vertex and proceed in this way till we reach the Destination.
* For a ‘k’ stage graph, there will be ‘k’ vertex in the path.
* In the above graph V1…V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

|  |  |
| --- | --- |
| **STEPS: -** | **DESTINATION, D** |
| Cost (12)=0 | D (12)=0 |
| Cost (11)=5 | D (11)=12 |
| Cost (10)=2 | D (10)=12 |
| Cost ( 9)=4 | D ( 9)=12 |
|  |  |

 For forward approach,

Cost (i,j) = min {C (j,l) + Cost (i+1,l) } l  Vi + 1

(j,l) E

Cost(8) = min {C (8,10) + Cost (10), C (8,11) + Cost (11) }

* min (5 + 2, 6 + 5)
* min (7,11)
* 7

cost(8) =7 =>D(8)=10

cost(7) = min(c (7,9)+ cost(9),c (7,10)+ cost(10)) (4+4,3+2)

= min(8,5)

= 5

cost(7) = 5 =>D(7) = 10

cost(6) = min (c (6,9) + cost(9),c (6,10) +cost(10))

= min(6+4 , 5 +2)

= min(10,7)

= 7

cost(6) = 7 =>D(6) = 10

cost(5) = min (c (5,7) + cost(7),c (5,8) +cost(8))

= min(11+5 , 8 +7)

= min(16,15)

= 15

cost(5) = 15 =>D(5) = 18

cost(4) = min (c (4,8) + cost(8))

= min(11+7)

= 18

cost(4) = 18 =>D(4) = 8

cost(3) = min (c (3,6) + cost(6),c (3,7) +cost(7))

= min(2+7 , 7 +5)

= min(9,12)

= 9

cost(3) = 9 =>D(3) = 6

cost(2) = min (c (2,6) + cost(6),c (2,7) +cost(7) ,c (2,8) +cost(8))

= min(4+7 , 2+5 , 1+7 )

= min(11,7,8)

= 7

cost(2) = 7 =>D(2) = 7

cost(1) = min (c (1,2)+cost(2) ,c (1,3)+cost(3) ,c (1,4)+cost(4) ,c(1,5)+cost(5))

= min(9+7 , 7 +9 , 3+18 , 2+15)

= min(16,16,21,17)

= 16

cost(1) = 16 =>D(1) = 2

The path through which you have to find the shortest distance.



(i.e.)



Start from vertex - 2

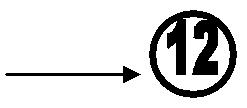
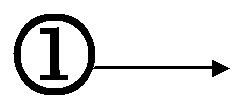
D ( 1) = 2

D ( 2) = 7

D ( 7) = 10

D (10) = 12

So, the minimum –cost path is,



9  2  3  2



 The cost is 9+2+3+2+=16

**ALGORITHM: FORWARD METHOD**

**Algorithm FGraph (G,k,n,p)**

* The I/p is a k-stage graph G=(V,E) with ‘n’ vertex.
* Indexed in order of stages E is a set of edges.
* and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.

{

cost[n]=0.0;

for j=n-1 to 1 step-1 do

{

//compute cost[j],

* + - let ‘r’ be the 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;

}

// find a minimum cost path.

P[1]=1;

P[k]=n;

For j=2 to k-1 do P[j]=d[p[j-1]];

}

**ANALYSIS:**

The time complexity of this forward method is O( V + E )

**BACKWARD METHOD**

**1.**if there one ‘K’ stages in a graph using back ward approach. we will find out the cost of each & every vertex starting from 1st

stage to the kth stage.

**2.**We will find out the minimum cost path from destination to source (ie)[from stage k to stage 1]

**PROCEDURE:**

* It is similar to forward approach, but differs only in two or three ways.
* Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
* Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
* To find out the path star from vertex ‘k’, then the distance array D (k) will give the minimum cost neighbor vertex which in turn gives the next nearest neighbor vertex and proceed till we reach the destination.

**STEP:**

Cost(1) = 0 => D(1)=0

Cost(2) = 9 => D(2)=1

Cost(3) = 7 => D(3)=1

Cost(4) = 3 => D(4)=1

Cost(5) = 2 => D(5)=1

Cost(6) =min(c (2,6) + cost(2),c (3,6) + cost(3)) =min(13,9)

cost(6) = 9 =>D(6)=3

Cost(7) =min(c (3,7) + cost(3),c (5,7) + cost(5) ,c (2,7) + cost(2))

=min(14,13,11)

cost(7) = 11 =>D(7)=2

Cost(8) =min(c (2,8) + cost(2),c (4,8) + cost(4) ,c (5,8) +cost(5)) =min(10,14,10)

cost(8) = 10 =>D(8)=2

Cost(9) =min(c (6,9) + cost(6),c (7,9) + cost(7)) =min(15,15)

cost(9) = 15 =>D(9)=6

Cost(10)=min(c(6,10)+cost(6),c(7,10)+cost(7)),c (8,10)+cost(8)) =min(14,14,15) cost(10)= 14 =>D(10)=6

Cost(11) =min(c (8,11) + cost(8))

cost(11) = 16 =>D(11)=8

cost(12)=min(c(9,12)+cost(9),c(10,12)+cost(10),c(11,12)+cost(11))

=min(19,16,21)

cost(12) = 16 =>D(12)=10

**PATH:**

Start from vertex-12

D(12) = 10

D(10) = 6

D(6) = 3

D(3) = 1

So the minimum cost path is,

1 7 3 2 6 5 10 2 12



The cost is 16.

**ALGORITHM : BACKWARD METHOD**

**Algorithm BGraph (G,k,n,p)**

//The I/p is a k-stage graph G=(V,E) with ‘n’ vertex.

//Indexed in order of stages E is a set of edges.

// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.

{

bcost[1]=0.0; for j=2 to n do

{

//compute bcost[j],

* + let ‘r’ be the vertex 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;

}

// find a minimum cost path.

P[1]=1;

P[k]=n;

For j= k-1 to 2 do P[j]=d[p[j+1]];

}

**0-1 Knapsack Problem**

The difference between this problem and the fractional one is that you can't take a fraction of an item. You either take the whole thing or none of it. So here, is the problem formally described:

Your goal is to maximize the value of a knapsack that can hold at most W units worth of goods from a list of items I0, I1, ... In-1. Each item has two attributes:

1) Value - let this be vi for item Ii.

2) Weight - let this be wi for item Ii.

Now, instead of being able to take a certain weight of an item, you can only either take the item or not take the item.

The naive way to solve this problem is to cycle through all 2n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack. But, we can find a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Our first attempt might be to characterize a sub-problem as follows:

Let Sk be the optimal subset of elements from {I0, I1,... Ik}. But what we find is that the optimal subset from the elements {I0, I1,... Ik+1} may not correspond to the optimal subset of elements from {I0, I1,... Ik} in any regular pattern. Basically, the solution to the optimization problem for Sk+1 might NOT contain the optimal solution from problem Sk.

To illustrate this, consider the following example:

Item Weight Value

I0 3 10

I1 8 4

I2 9 9

I3 8 11

The maximum weight the knapsack can hold is 20.

The best set of items from {I0, I1, I2} is {I0, I1, I2} but the best set of items from {I0, I1, I2, I3} is {I0, I2, I3}. In this example, note that this optimal solution, {I0, I2, I3}, does NOT build upon the previous optimal solution, {I0, I1, I2}. (Instead it build's upon the solution, {I0, I2}, which is really the optimal subset of {I0, I1, I2} with weight 12 or less.)

So, now, we must rework our example. In particular, after trial and error we may come up with the following idea:

Let B[k, w] represent the maximum total value of a subset Sk with weight w. Our goal is to find B[n, W], where n is the total number of items and W is the maximal weight the knapsack can carry.

Using this definition, we have B[0, w] = w0, if w >= w0.

= 0, otherwise

Now, we can derive the following relationship that B[k, w] obeys:

B[k, w] = B[k - 1,w], if wk > w

= max { B[k - 1,w], B[k - 1,w - wk] + vk}

In English, here is what this is saying:

1) The maximum value of a knapsack with a subset of items from {I0, I1, ... Ik} with weight w is the same as the maximum value of a knapsack with a subset of items from {I0, I1, ... Ik-1} with weight w, if item k weighs greater than w.

Basically, you can NOT increase the value of your knapsack with weight w if the new item you are considering weighs more than w – because it WON'T fit!!!

2) The maximum value of a knapsack with a subset of items from {I0, I1, ... Ik} with weight w could be the same as the maximum value of a knapsack with a subset of items from {I1, I2, ... Ik-1} with weight w, if item k should not be added into the knapsack.

OR

3) The maximum value of a knapsack with a subset of items from {I0, I1, ... Ik} with weight w could be the same as the maximum value of a knapsack with a subset of items from {I0, I1, ... Ik-1} with weight w-wk, plus item k.

You need to compare the values of knapsacks in both case 2 and 3 and take the maximal one.

Recursively, we will STILL have an O(2n) algorithm. But, using dynamic programming, we simply have to do a double loop - one loop running n times and the other loop running W times.

Here is a dynamic programming algorithm to solve the 0-1 Knapsack problem:

Input: S, a set of n items as described earlier, W the total weight of the knapsack. (Assume that the weights and values are stored in separate arrays named w and v, respectively.)

Output: The maximal value of items in a valid knapsack.

int w, k;

for (w=0; w <= W; w++)

B[w] = 0

for (k=0; k<n; k++) {

for (w = W; w>= w[k]; w--) {

if (B[w – w[k]] + v[k]> B[w])

B[w] = B[w – w[k]] + v[k]

}

}

Note on run time: Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops. When comparing this with the previous O(2n), we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient. (For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable.)

**Let's run through an example:**

|  |  |  |  |
| --- | --- | --- | --- |
| i | Item | wi | vi |
| 0 | I0 | 4 | 6 |
| 1 | I1 | 2 | 4 |
| 2 | I2 | 3 | 5 |
| 3 | I3 | 1 | 3 |
| 4 | I4 | 6 | 9 |
| 5 | I5 | 4 | 7 |

W = 10

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Item | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | 0 | 0 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 1 | 0 | 0 | 4 | 4 | 6 | 6 | 10 | 10 | 10 | 10 | 10 |
| 2 | 0 | 0 | 4 | 5 | 6 | 9 | 10 | 11 | 11 | 15 | 15 |
| 3 | 0 | 3 | 4 | 7 | 8 | 9 | 12 | 13 | 14 | 15 | 18 |
| 4 | 0 | 3 | 4 | 7 | 8 | 9 | 12 | 13 | 14 | 16 | 18 |
| 5 | 0 | 3 | 4 | 7 | 8 | 10 | 12 | 14 | 15 | 16 | 19 |

All Pairs Shortest Paths

If we want to find the shortest distance between any two nodes in a graph, we can:

1. run Bellman-Ford algorithm (or any other single-source shortest paths algorithm) using each node as the source

2. invent some other algorithm.

Simple ideas:

What is in the weight matrix? The shortest paths consisting of one edge.

How do we get the shortest paths consisting of, say, two edges?

For example, we have the following graph.

b

d

a

c

-4

5

3

2

1

10

a b c d

O 5 10 3

W= X O 1 -4

X X O X

X X 2 O

We have to think about what we want to put

in places of O and X. X means “no connection”

and O means “no self loop.”

There is a path from a to c of weight 10. But, if we allow paths of 2 edges, then there

is a shorter path adc, and also abc. How do we get to find them using the W matrix?

So, the best path so far is the smallest path between ac, ab+bc, and ad+dc.

Idea: “multiply” the W matrix by itself. Instead of multiplication, use addition, and instead of summing, use minimum between several values.

If we put 0 for O and infinity for X, then we will find shortest paths of at most two edges. If we put infinity for both O and X, then we will find shortest paths of exactly two edges, and then at the end we will still have to find the shortest path out of all possible shortest paths of 1, 2, 3, …., n-1 edges. So, it is easier to do the work as we are going along and find all shortest paths of at most m edges. So, we will use 0 for O and infinity for X.

Then, how do we get the shortest path of at most three edges? For ac, now we have to pick the minimum between ac, abc, adc, and abdc.

Let’s formalize:

“Regular” all-pairs shortest paths

Main idea: keep on “growing” the adjacency matrix so that we find the shortest paths of at most one edge, two edges, three edges, etc.

lij(m) represents the weight of the shortest path from node i to node j such that the entire path contains at most m edges.

Solve by using modified matrix multiplication.

“Regular” matrix multiplication: C = AB cij = Σk=1, n (aik \* bkj)

“Shortest-path” matrix multiplication: lij(m) = min (lij(m-1) + wkj)

1≤ k ≤ n

Exercise1: program the above formula as a dynamic programming solution.

Exercise2: also include pseudocode to reconstruct the shortest path.

Slow-n-sure way:

L(1) = W

L(2) = L(1) ° W

L(3) = L(2) ° W

…

L(n-1) = L(n-2) ° W

Faster way: Since L(s) = L(n-1) for all s ≥ n-1, we can skip all intermediate steps and work by doubling the order of L.

L(1) = W

L(2) = W ° W

L(4) = L(2) ° L(2)

L(8) = L(4) ° L(4)

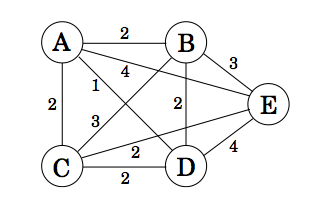
…

L(s) = L(s/2) ° L(s/2) for the first s such that s ≥ n-1 and lg s =k, where k Є I.

**Some All Pairs Shortest Paths Algorithms (Dynamic programming)**

|  |  |  |
| --- | --- | --- |
| **Algorithm name:** | “Regular” | Floyd-Warshall |
| **Negative weight cycles OK?** | no | no |
| **Uses matrix:** | L  L(1) , L(2) , …, L(n-1) | D  D(0) , D(1) , …, D(n) |
| which contains elements | lij  lij(1) , lij(2) , …, lij(n-1) | dij  dij(0) , dij(1) , …, dij(n) |
| which represent | lij (k) = d(i, j) such that  lij (k) is the min weight of all paths from vertex i to vertex j that contain at most k edges. | dij (k) = d(i, j) such that  dij (k) is the min weight of all paths from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, 3, 4, …, k}. |
| **Final solution is:** | L(n-1) | D(n) |
| Start from: | L(1) = W  lij (1) = wij | D(0) = W  dij (0) = wij |
| Build as: | lij(m)= min(lij(m-1), min ((lik(m-1) + wkj))  1≤ k ≤ n  lij(m)= min ((lik(m-1) + wkj))  1≤ k ≤ n  Process the entire row/column.    L(2) = W \*W  L(3) = L(2) \* W  …  Or keep on doubling up | dij(k) = min (dij(k-1),dik(k-1) + dkj(k-1))  Process only two elements. |
| **The shortest paths are in:**  Predecestor matrices | π(1) , π(2) , …, π(n-1) matrices | π(0) , π(1) , …, π(n) matrices  use 25.2-7, or  πij (0) = NIL, if i=j or wij = ∞  = i, if i≠j or wij < ∞  πij (k) = πij (k-1),  if dij(k-1) ≤dik(k-1) + dkj(k-1)  = πkj (k-1),  if dij(k-1) >dik(k-1) + dkj(k-1) |

**The traveling salesman problem**  
  
 A traveling salesman is getting ready for a big sales tour. Starting at his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returns home. Given the pair wise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?  
  
Denote the cities by 1,...,n, the salesman’s hometown being 1, and let D = (*dij*) be the matrix of intercity distances. The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length. Figure shows an example involving five cities.



Let's dive right into the DP. So what is the appropriate sub-problem for the TSP? In this case the most obvious partial solution is the initial portion of a tour. Suppose we have started at city 1 as required, have visited a few cities, and are now in city j. What information do we need in order to extend this partial tour? We certainly need to know j, since this will determine which cities are most convenient to visit next. And we also need to know all the cities visited so far, so that we don’t repeat any of them. Here, then, is an appropriate sub-problem.  
  
**For a subset of cities S ⊆ {1,2,...,n} that includes 1, and j ∈ S, let C(S,j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.**  
  
  
When |S| > 1, we define C(S, 1) = ∞ since the path cannot both start and end at 1.  
  
Now, let’s express C(S,j) in terms of smaller sub-problems. We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some i ∈ S, so the overall path length is the distance from 1 to i, namely, C(S − {j}, i), plus the length of the final edge,*dij*. We must pick the best such i:  
  
***C*(*S*,*j*)=min*i*∈*S*:*i*≠*jC*(*S*−{*j*},*i*)+*dij***  
  
The sub-problems are ordered by |S|. Here’s the code.  
  
  
C({1},1) = 0  
for s = 2 to n:  
    for all subsets S ⊆ {1,2,...,n} of size s and containing 1:   
        C(S,1) = ∞  
        for all *j*∈*S*,*j*≠1:   
            C(S, j) = *min*{*C*(*S*−{*j*},*i*)+*dij*:*i*∈*S*,*i*≠*j*}  
return *minjC*({1,...,*n*},*j*)+*dj*1  
  
There are at most 2*n*.*n* sub-problems, and each one takes linear time to solve. The total running time is therefore *O*(*n*2.2*n*).

**Optimal Binary Search Tree Problem**

Let *C*[*i,j*] be minimum average number of comparisons made in T[*i,j*], optimal BST for keys *ai* < …< *aj* , where 1 ≤ *i* ≤ *j* ≤ *n.* Consider optimal BST among all BSTs with some *ak* (*i* ≤ *k* ≤ *j* ) as their root; T[*i,j*] is the best among them.

Fig 8

***C*[*i,j*] =**

**min {*pk ·* 1 +**

**∑ *ps* (level *as* in T[*i,k-*1] +1)** **+**

**∑ *ps* (level *as* in T[*k+*1*,j*] +1)}**

**After simplifications, we obtain the recurrence for *C*[*i,j*]:**

***C*[*i,j*] =**  **min {*C*[*i*,*k*-1] + *C*[*k*+1,*j*]} + ∑ *ps*** **for** **1 ≤ *i* ≤ *j* ≤ *n***

***C*[*i,i*] = *pi* for 1 ≤ *i* ≤ *j* ≤ *n***

**The tables below are filled diagonal by diagonal: the left one is filled using the recurrence**

***C*[*i,j*] =**  **min {*C*[*i*,*k*-1] + *C*[*k*+1,*j*]} + ∑ *ps , C*[*i,i*] = *pi* ;**

**the right one, for trees’ roots, records *k*’s values giving the minima**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** |
| **1** | **0** | **.1** | **.4** | **1.1** | **1.7** |
| **2** |  | **0** | **.2** | **.8** | **1.4** |
| **3** |  |  | **0** | **.4** | **1.0** |
| **4** |  |  |  | **0** | **.3** |
| **5** |  |  |  |  | **0** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** |
| **1** |  | **1** | **2** | **3** | **3** |
| **2** |  |  | **2** | **3** | **3** |
| **3** |  |  |  | **3** | **3** |
| **4** |  |  |  |  | **4** |
| **5** |  |  |  |  |  |

