**VIVEKANANDA COLLEGE**

**THAKURPUKUR**

**KOLKATA-700063**

**NAACACCREDITED ‘A’ GRADE**

| | |
|---|---|
| **Topic** | **: Introduction to Algorithms & its Applications** |
| **Course Title** | **:Basics of Algorithm Design Technique** |
| **Paper** | **:CMSA-CC9** |
| **Semester** | **:4th semester** |
| **Name of the Teacher** | **:Mahuya Paul** |
| **Name of the Department** | **:Computer Science** |

## Basic of Algorithm

**Definition:** An algorithm is a finite set of instructions which accomplish a particular task. Every computer program that ends with a result is basically based on an Algorithm. Algorithms, however, are not just confined for use in computer programs; these are at the core of most computing tasks. Algorithm must satisfy some characteristics:

i) Input: There are some (possibly empty) input data which are externally supplied to the algorithm.
ii) Output: There will be at least one output.
iii) Definiteness: Each instructions/steps of the algorithm must be clear and unambiguous.
iv) Finiteness: The algorithm will terminate after a finite number of steps.
v) Effectiveness: The steps of an algorithm must be sufficiently basic that it can be carried out by a person mechanically using pen and paper.

Depending on the strategy for solving a particular task algorithms can be divided into multiple types. In an algorithm design, different problems require the use of different kinds of techniques. A good programmer uses all these techniques based on the type of problem. Some commonly-used techniques are:

Let’s take a look at some of the important ones.

- Recursive Algorithm
- Greedy Algorithm
- Divide and Conquer Algorithm
- Dynamic Programming Algorithm.

## RECURSION (Recursive algorithm)

One of the important aspects of **recursion** is the resulting conceptual simplification of algorithms. Recursion by itself does not necessarily lead to more efficient algorithms.

However, it yields both efficient and elegant algorithms when it is combined with other techniques such as balancing, divide-and-conquer, and algebraic simplification.

A procedure that calls itself directly or indirectly is said to be ***recursive.*** At the heart of recursive procedure implementation is ‘a stack’ in which the data are stored used by each call of a procedure which has not yet terminated. The stack is divided into *stack frames,* which are blocks of consecutive locations (registers). Each call of a procedure uses a stack frame whose length depends on the particular procedure called.

The time required for a procedure call is proportional to the time required to evaluate the actual parameters and store pointers to their values on the stack. The time for a return is certainly no greater than this.

In accounting for the time spent by a collection of recursive procedures, it is usually easiest to charge the cost of a call to the procedure during the calling. The time spent to call a procedure depends on the function of input size of the called procedure.

The **running time of a recursive call** can be described by a **recurrence equation or recurrence relation**. Then mathematical tools are used to solve the recurrence and provide bounds on the performance of the algorithm.

**[** A ***recurrence relation*** is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s). The simplest form of a recurrence relation is the case where the next term depends only on the immediately previous term. If we denote the nth term in the sequence by $x_n$, such a recurrence relation is of the form

$$x_{n+1}=f(x_n) \text{ for some function f.}$$

A recurrence relation can also be higher order, where the term $x_{n+1}$ could depend not only on the previous term $x_n$ but also on earlier terms such as $x_{n-1}$, $x_{n-2}$, etc. A second order recurrence relation depends just on $x_n$ and $x_{n-1}$and is of the form

$x_{n+1}=f(x_n, x_{n-1})$ for some function f with two inputs.

For example, the recurrence relation xn+1=xn+xn−1 can generate the Fibonacci numbers

To generate sequence based on a recurrence relation, one must start with some initial values. For a first order recursion $x_{n+1}=f(x_n)$, one just needs to start with an initial value $x_0$ and can generate all remaining terms using the recurrence relation. For a second order recursion $x_{n+1}= f(x_n, x_{n-1})$, one needs to begin with two values $x_0$ and $x_1$. Higher order recurrence relations require correspondingly more initial values.**]**

**T**here are **3 methods for solving recurrences**. They are as follows:

1. **Substitution method**
2. **Recursion-Tree method**
3. **Master method(Master theorem)**

1**. Substitution method:**

In this method, after guessing a bound the mathematical induction is used to prove the guess correct; I,e this method follows the two steps:

a) Guess the form of the solution
b) Use mathematical induction to found the constants and show that the solution works.

This method can be used to establish either upper or lower bounds for any recurrence relation where it is easy to guess the form of the answer. But the method is slow comparing to others method.

Example:
```
void recur (int n)  //main function named recur() requires time T(n)
{
        if (n>1)
                {
                  for(i=0;i<n;i++)
                        {
                            printf("%d",n);   // this statement requires time 'n ' for n number of inputs.
                        }
                 recur (n/2);    // recursive call of function recur() requires time' n/2' for n/2 number of inputs
                 recur(n/2);    // recursive call of function recur() requires time' n/2' for n/2 number of inputs
                }
}
```
Now total time required for the above function is $T(n)= n +T(n/2)+ T(n/2)$
$=2T(n/2)+n$

The recurrence relation can be written as

$T(n)= 1$   n=1   [as this is an dividing function recur (n/2),the smallest input value is 1]
And $T(n)= 2T(n/2)+n$   n>1

Now $T(n)= 2T(n/2)+n$…………..equ(1)
$T(n)=2[\boldsymbol{2T(n/2^2)+n/2}]+n$ [substituting n=n/2 in the equation (1) $T(n)= 2T(n/2)+n$ as ***T(n/2)=2T((n/2)/2)+n/2***]
$T(n)=2^2T(n/2^2)+n+n$
$T(n)=2^2T(n/2^2)+2n$…………equ(2)
$T(n)= 2^2 [2T(n/2^3)+n/2^2]+2n$ [substituting $n=n/2^2$ in the equation (1) $T(n)= 2T(n/2)+n$ as $\boldsymbol{T(n/2^2)=2T((n/2^2)/2)+n/2^2}$]
$T(n)= 2^3 T(n/2^3)+n+2n$
$T(n)=2^3T(n/2^3)+3n$…………equ(3)

………………………………

………………………………..

Now substituting k times we get

$T(n)=2^kT(n/2^k)+kn$

Assuming $T(n/2^k)=T(1)$ [for smallest input]

$n/2^k=1$ ; $n=2^k$; **$k=\log n$**

Now $T(n)=2^kT(n/2^k)+kn$

$=2^k\ T(1)+kn$ [since $T(n/2^k)=T(1)$ ]

$=n\times1+\log n\times n$ [since $n=2^k$ and $T(1)=1$ and $k=\log n$]

$=n + n \log n$

Taking higher part we may write **$T(n)= O(n \log n)$.**

2. **Recursion-tree method:**

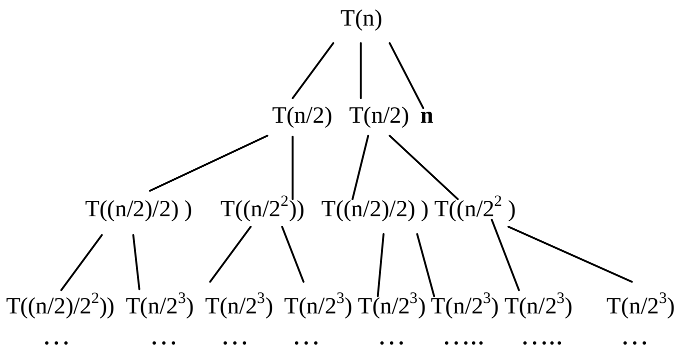
This method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion; then techniques are used for bounding summations to solve the recurrence. Recursion trees are particularly useful when the recurrence describes the running time of a divide and conquer algorithm. Basically, a recursion tree is best used to generate a good guess, which is then verified by the substitution method.

Example: For the above mentioned function

The recurrence relation can be written as

$T(n)= 1$ $n=1$ [as this is an dividing function recur (n/2),the smallest input value is 1]

And $T(n)= 2T(n/2)+n$ $n>1$



T(n)
T(n/2) T(n/2) n
T((n/2)/2) ) T((n/2²)) T((n/2)/2) ) T((n/2² )
T((n/2)/2²)) T(n/2³) T(n/2³) T(n/2³) T(n/2³) T(n/2³) T(n/2³) T(n/2³)
… … … … … ….. ….. …

[**n** is also there for each level ,due to reduce complexity of the figure ignore the lower order n.]

Now the levels are repeat for $n/2^k$ times,so k steps are there, so the total required time is nk. Because the time taken for each level is n [as at level 2, $n/2+n/2=n$; at level 3, $n/2^2+n/2^2+n/2^2+n/2^2=n$…and so on]

Now Assuming $T(n/2^k)=T(1)$ [for smallest input]

$n/2^k=1$ ; $n=2^k$; **$k=\log n$**

**so total time will be nk=n log n**

we may write **$T(n)= O(n \log n)$.**

3. **Master method:**

This method provides bounds for recurrences of the form $T(n)=aT(n/b)+f(n)$,where $a\geq 1,b>1$ and $f(n)$ is a given function.

and $f(n)=O(n^k \log^p n)$

Now **Case 1**: if $\log_b^a>k$ then $T(n)=O(n\wedge \log_b^a)$

**Case 2**: if $\log_b^a=k$ then

i) If $p>-1$ $T(n)= O(n^k \log^{p+1} n)$

ii) If $p=-1$ $T(n)= O(n^k \log\log n)$

iii) If $p<-1$ $T(n)= O(n^k)$

**Case 3**: if $\log_b^a<k$ then

i) If $p\geq 0$ then $T(n)=O(n^k \log^p n)$

ii) If $p<0$ then $T(n)=O(n^k)$

** So to solve a recurrence using master method we should always find the value of 1) $\log_b^a$ and 2)k

Example 1:$T(n)=2T(n/2)+1$

Here a=2,b=2 [since $T(n)=aT(n/b)+f(n)$]

And $f(n)=O(1)$

$=O(n^0\log^0 n)$ [as we have to write f(n) in the form of $O(n^k \log^p n)$

And here $n^0=1$and there is no value for log so $\log^0 n$]

Now here k=0 and $\log_b^a= \log_2^2=1$

i.e. here case 1 $\log_b^a>k$ is proved.

So $T(n)=O(n^1)=O(n)$.

Example 2:$T(n)=4T(n/2)+n$

Here $\log_b^a= \log_2^4=2>k=1$[k is the always power of n here it is 1 and p(power of log)=0]

So according to case 1, $T(n)= O(n\wedge \log_b^a)=O(n^2)$

**We may concluded as when the value of $\log_b^a>$power of n(value of k), $T(n)= O(n\wedge \log_b^a)$

Example 3:$T(n)=2T(n/2)+n$

Here $\log_b^a= \log_2^2=1$ and k=1 and p=0 [as there f(n)=n, no log value as f(n)]

So according to case 2, If $p>-1$ $T(n)= O(n^k \log^{p+1} n)$

$=O(n^1\log^{0+1} n)$

$=O(n \log n)$

Example 4:$T(n)=4T(n/2)+n^2\log^2 n$

So according to case 2, If $p>-1$ $T(n)= O(n^k \log^{p+1} n)$

$=O(n^2\log^{2+1} n)$[as $\log_b^a= \log_2^4=2$ and k=2 and p=2]

$=O(n^2 \log^3 n)$

**We may concluded as when the value of $\log_b^a$=power of n(value of k), $T(n)= O(n^{\text{power of } n} \log(\text{rest part}))$

# GREEDY ALGORITHM APPROACH

A greedy algorithm **always makes the choice that seems to be the best at that moment**. It **makes a locally-optimal choice** in the hope that this choice will **lead to a globally-optimal** solution. This approach is used to find the optimization result in a given problem.

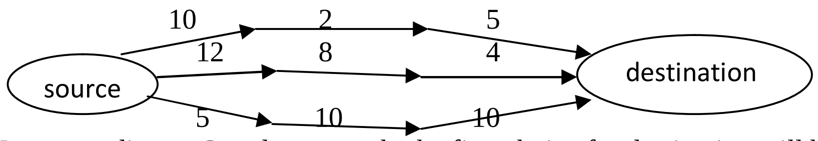
[A solution that satisfies the constraints (condition) in a given problem is **feasible solution**. There are more than one feasible solution of a given problem. But the feasible solution that satisfies the optimal objective (minimization/maximization result) of a given problem is referred to as **optimal solution**.]

More generally, the greedy algorithm follows the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one sub problem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice , so that the greedy choice is always safe.
3. Demonstrate that if we combine an optimal solution to the sub problems with the greedy choice we have made, we arrive at an optimal solution to the original problem.

The disadvantage of Greedy approach is that this approach focuses only on the local optimal solution of each stage that may or may not be the final global optimal solution; i.e. the **Greedy choice property** is to consider making choice that looks best in the current problem (locally optimal) without considering results from sub problems.

Example: One Source to Destination having many paths via another paths along with their cost is as follows:



Now according to Greedy approach, the first choice for destination will be the route of currents minimum cost '5' at first stage, whereas the total cost towards this route will be 25(5+10+10), maximum cost compare to the other two sub problems [two routes (12+8+4=24 and 10+2+5=17)], that will not be the optimal solution. However a greedy choice at each step yields a globally optimal solution requires cleverness. Matroids involving combinatorial structure is a theory useful for finding the optimal solution using greedy algorithm.

**Matroid:** A matroid consists of a base set U and a collection I of independent subsets. Independence will be related to different objects depending on the problem **-** for the minimum spanning tree, an independent subset could be a tree. In linear algebra, an independent subset could be linearly independent vectors.

**[[**A matroid's notion of independence is as follows. A, B $\subseteq$ U satisfy:

1. If A is a subset of B and B is independent subset, then A is independent subset, too. In other words: $A \subseteq B, B \in I \Rightarrow A \in I$

2. The empty set is independent.

3. Suppose A and B are indepedent sets and A is smaller than B. Then there is some element e in B that is not in A such that A plus e is also an independent set. In other words: $A, B \in I, |A| < |B| \Rightarrow \exists e \in B \backslash A$ such that $A \cup \{e\} \in I$

Now suppose that we are given any nonnegative weighting of elements of U. Another way of writing this is $w : U \to R+$. Finding a max weight independent set can be done using the greedy algorithm**]]**

Some problems those use Greedy approach to solve it are-

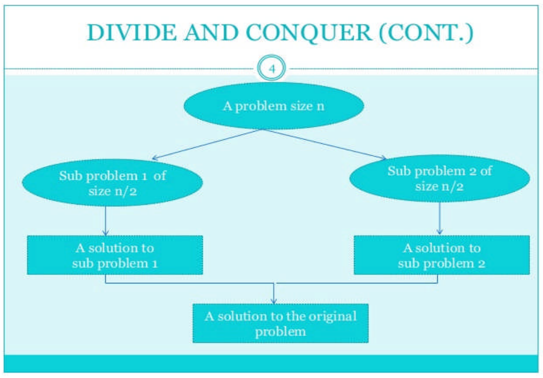
1. Knapsack problem
2. Job sequencing problem
3. Huffman coding problem
4. Minimum spanning tree
5. Optimal merge pattern
6. Dijkstra's algorithm

## DIVIDE AND CONQUER METHOD

Divide and conquer approach partitions a problem into smaller parts (several sub problems) that are similar to the original problem until it become enough to solve directly, find solutions for the sub parts recursively, and then combines these solutions to create a solution for the original problem. It is a top down approach.

A typical Divide and Conquer algorithm solves a problem using following three steps.

1. **Divide**: Divide the problem into a number of sub problems that are smaller instances of the same problem.
2. **Conquer**: Conquer the sub problems by solving them recursively.
3. **Combine**: Combine the solutions of the sub problems into the solution for the original problem.


DIVIDE AND CONQUER (CONT.)
4
A problem size n
Sub problem 1 of size n/2
Sub problem 2 of size n/2
A solution to sub problem 1
A solution to sub problem 2
A solution to the original problem

When an algorithm contains a **recursive call** to itself, the overall running time on a problem of smaller size inputs n can be described by **a recurrence equation or recurrence.** Typically a Divide and conquer algorithm has the time complexity as recurrence relation of the following form:

$T(n)= \Theta(1)$ if $n \leq c$ and

$T(n)= aT(n/b)+D(n)+C(n)$ otherwise.

Where T(n) be the running time of problem size is small enough,

$n \leq c$ for some constant c [straight forward solution takes constant time writing as $\Theta(1)$] ,

there are 'a' no of sub problems each of which is 1/b the size of the original problem,

D(n) is the time taken to divide the problem into sub problems,

C(n) is the time taken to combine the solutions to the subproblems into the original solution.

**Advantages:**

1. In this method input elements of a problem are subdivided into two halves each time, so it requires **less number of comparisons** and so it is **faster** than sequential solution.
2. **Time** spent on executing the problem is smaller than others
3. Suited for **parallel computations** in which each sub problem can be solved simultaneously by its own processor.
4. Divide-and-conquer algorithms naturally tend to make **efficient use of memory caches**. The reason is that once a sub-problem is small enough, its and all its sub- problems can be solved within the cache, without accessing the slower main memory.

**Disadvantages:**

1. This strategy uses recursion that makes it a little slower
2. Usage of explicit stacks may make use of extra space.
3. If performing a recursion for no. times greater than the stack in the CPU then the system may crash.
4. Choosing base cases as small one is also a limitation
5. Sometimes a case where the problem when broken down results in same sub problems which may needlessly increase the solving time and extra space may be consumed.

**Applications of Divide and Conquer Strategy**

1. Binary Search
2. Merge Sort
3. Quick Sort
4. Closest Pair of Points (a set of n ***points*** are given on the 2D plane, we have to find the ***pair of points***, whose distance is minimum.)
5. Strassen's Multiplication ( helps us to **multiply** two **matrices**(of size n X n).)
6. Karatsuba Algorithm (The **Karatsuba an algorithm is** a fast multiplication algorithm that uses a divide and conquers approach to multiply two numbers.)
7. Cooley-Tukey Algorithm (The **Cooley–Tukey algorithm**, named after J. W. Cooley and John Tukey, is the most common fast Fourier transform (FFT) algorithm.)

## Dynamic programming

**Dynamic programming** approach divides a problem into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, **dynamic** algorithm will try to examine the results of the previously solved sub-problems.

**Dynamic Programming** refers to a very large class of algorithms. The idea is to break a large problem down (if possible) into incremental steps so that, at any given stage, optimal solutions are known to sub-problems.

The development of a dynamic –programming algorithm can be broken into a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information.

In Dynamic Programming, we need to break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems

Dynamic programming basically implements recursion using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells that the function is implemented in a way that the recursive calls are done in advance and stored for easy access, it will make the program faster. This is what we call **Memorization -** it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

**The intuition behind dynamic programming is that we trade space for time**, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

Fibonacci (n) = 1; if n = 0
Fibonacci (n) = 1; if n = 1
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: **1, 1, 2, 3, 5, 8, 13, 21...** and so on!

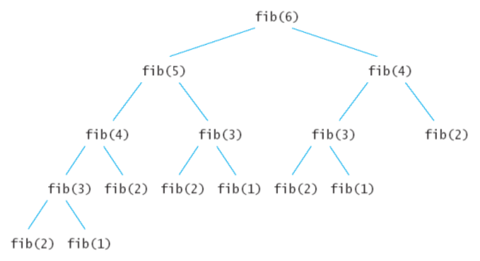
A code for it using pure recursion:

```
 int fib (int n) {
      if (n < 2)
          return 1;
      return fib(n-1) + fib(n-2);
   }
```

Using Dynamic Programming approach with memorization:

```
 void fib () {
      fibresult[0] = 1;
      fibresult[1] = 1;
      for (int i = 2; i<n; i++)
         fibresult[i] = fibresult[i-1] + fibresult[i-2];
   }
```

In the recursive code, a lot of values are being recalculated multiple times. We could do well with calculating each unique quantity only once. Take a look at the image to understand that how certain values were being recalculated in the recursive way:



- Here value of fib(1),fib(2),fib(3)……… are used for several times as a result of sub problems depend on the input range ‘n’. So the value of this similar sub problems (fib(1),fib(2),fib(3)…..) are solved just once and saves the answer to avoid the work of re-computing.

**Comparison among Greedy approach, Dynamic Programming and Divide And Conquer algorithm:**

In **Greedy approach**,

- A globally optimal solution can be arrived by making a **locally optimal**(greedy) choice; i.e, the choice is considered that looks best in the **current problem** without considering the results from sub problems, which may or may not lead a global optimal solution always. Ex. Prim's algorithm
- The choice of solution at any stage depends on choices so far, but it cannot depend on any future choices of other sub problems.
- This strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.
- It may or may not give the optimal solution always.
- It solves the sub problem in a top down manner.

**[N.B. Follow the doc file of Algorithm design_ Greedy approach]**

In **Dynamic Programming**

- Combines the features of Greedy approach and divide and conquer method in some extent.
- It divides a problem into smaller one, makes the choice to find the solution usually depends on the solutions to other sub problems (optimal value)
- Combine the solutions by **overlapping the similar sub problem**.
- In this method, sub problems are generally dependent.
- It solves every similar sub problems just once and saves the answer to avoid the work of re-computing.
- It always leads a global **optimal solution**. Ex. Fibonacci series with recursion.
- It solves the sub problem in a bottom up manner.

In **divide and conquer method**,

- A Problem is divided into subparts, solve each of them recursively and then combines all the solutions to get the final solution of the original problem
- There is no overlapping of sub problems; i.e, usually results of similar sub problems can't be stored for the solution of other sub problem.
- In this method, **sub problems are independent**, so that it requires more work than necessary, repeatedly solving the common sub problems. Ex.Merge sort

**[N.B. Follow the doc file of Algorithm design_ Divide and Conquer approach]**