and Output: ABCD ACBD ACDB CABD CADB CDAB. An interleaved string of given two strings preserves the order of characters in individual strings. For example, in all the interleavings of above first example, 'A' comes before 'B' and 'C' comes before 'D'.

Solution: Let the length of $str1$ be m and the length of $str2$ be n. Let us assume that all characters in $str1$ and $str2$ are different. Let Count(m, n) be the count of all interleaved strings in such strings. The value of Count(m, n) can be written as following.

Count(m, n) = Count(m-1, n) + Count(m, n-1)
Count(1, 0) = 1 and Count(1, 0) = 1

To print all interleavings, we can first fix the first character of str1[0..m-1] in output string, and recursively call for str1[1..m-1] and str2[0..n-1]. And then we can fix the first character of str2[0..n-1] and recursively call for str1[0..m-1] and str2[1..n-1].

```
void PrintInterleavings(char *str1, char *str2, char *iStr, int m, int n, int i){
    // Base case: If all characters of str1 & str2 have been included in output string, then print the output string
    if ( m==0 && n ==0 )
      printf("%s\n", iStr) ;

    // If some characters of str1 are left to be included, then include the
    // first character from the remaining characters and recur for rest
    if ( m != 0 )  {
      iStr[i] = str1[0];
      PrintInterleavings(str1 + 1, str2, iStr, m-1, n, i+1);
    }

    // If some characters of str2 are left to be included, then include the
    // first character from the remaining characters and recur for rest
    if ( n != 0 ) {
      iStr[i] = str2[0];
      PrintInterleavings(str1, str2+1, iStr, m, n-1, i+1);
    }
}

// Allocates memory for output string and uses printIlsRecur() for printing all interleavings
void Print(char *str1, char *str2, int m, int n){
    // allocate memory for the output string
    char *iStr= (char*)malloc((m+n+1)*sizeof(char));

    // Set the terminator for the output string
    iStr[m+n] = '\0';

    // print all interleavings using printIlsRecur()
    PrintInterleavings(str1, str2, iStr, m, n, 0);

    free(iStr);
}
```


ALGORITHMS DESIGN TECHNIQUES

Chapter-16

16.1 Introduction

In all previous chapters, we have seen many algorithms for solving different kinds of problems. If we get a new problem, before solving it, the general tendency is to identify the similarity of current problem with other problems for which we have solutions. This helps us in getting the solution easily.

In this chapter, we will see different ways of classifying the algorithms and in subsequent chapters focus will be given to few of them in detail (say, Greedy, Divide and Conquer and Dynamic Programming).

16.2 Classification

There are many ways of classifying the algorithms and few of them are shown below:

- By Implementation Method
- By Design Method
- Other Classifications

16.3 Classification by Implementation Method

Recursion or Iteration

A *recursive* algorithm is one that calls itself repeatedly until a base condition is satisfied. It is a common method used in functional programming languages like $C, C++$, etc..

Iterative algorithms use constructs like loops and sometimes other data structures like stacks, queues to solve the problems.

Some problems are suited for recursive and other suits for iterative. For example, *Towers of Hanoi* problem can be easily understood in recursive implementation. Every recursive version has an iterative version, and vice versa.

Procedural or Declarative (Non-Procedural)

In *Declarative* programming languages, we say what we want without having to say how to do it. With *procedural* programming, we have to specify exact steps to get the result. For example, SQL is more declarative than procedural, because the queries don't specify steps to produce the result. Examples for procedural languages includes: C, PHP, PERL, etc..

Serial or Parallel or Distributed

In general, while discussing the algorithms we assume that computers execute one instruction at a time. These are called *serial* algorithms.

Parallel algorithms, takes advantage of computer architectures to process several instructions at a time. They divide the problem into subproblems and serve them to several processors or threads. Iterative algorithms are generally parallelizable.

Distributing the parallel algorithms on to different machines then we call such algorithms as *distributed* algorithms.

Deterministic or Non-Deterministic

Deterministic algorithms solve the problem with a predefined process whereas $non-deterministic$ algorithms guesses the best solution at each step through the use of heuristics.

Exact or Approximate

As we have seen, for many problems we are not able to find the optimal solutions. That means, the algorithms for which we are able to find the optimal solutions are called *exact* algorithms. In computer science, if we do not have optimal solution, then we try to give approximation algorithms. Approximation algorithms are generally associated with NP-hard problems (refer *Complexity Classes* chapter for more details).

16.4 Classification by Design Method

Another way of classifying algorithms is by their design method.

Greedy Method

Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future consequences. Generally, this means that some *local best* is chosen. It assumes that local good selection makes the *global* optimal solution.

Divide and Conquer

The D & C strategy solves a problem by:

1) Divide: Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
2) Recursion: Recursively solving these sub problems.
3) Conquer: Appropriately combining their answers.

Examples: merge sort and binary search algorithms.

Dynamic Programming

Dynamic programming (DP) and memoization work together. The difference between DP and divide and conquer is that in-case of divide and conquer there is no dependency among the subproblems, where as in DP there will be overlap of subproblems. By using memoization [maintaining a table for already solved subproblems], DP reduces the exponential complexity to polynomial complexity ($O(n^2)$, $O(n^3)$, etc..) for many problems.
The difference between dynamic programming and recursion is in memoization of recursive calls. When subproblems are independent and if there is no repetition, memoization does not help, hence dynamic programming is not a solution for all problems. By using memoization [maintaining a table of subproblems already solved], dynamic programming reduces the complexity from exponential to polynomial.

Linear Programming

In linear programming, we will be having inequalities in terms of inputs and maximize (or minimize) some linear function of the inputs. Many problems (example: maximum flow for directed graphs) can be discussed using linear programming.

Reduction [Transform and Conquer]

In this method we solve the difficult problem by transforming it into a known problem for which we have asymptotically optimal algorithms. In this method, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms. For example, selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called as *transform and conquer*.

16.5 Other Classifications

Classification by Research Area

In computer science each field has its own problems and needs efficient algorithms. Examples: search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, geometric algorithms, combinatorial algorithms, machine learning, cryptography, parallel algorithms, data compression algorithms and parsing techniques and more.

Classification by Complexity

In this classification, algorithms are classified by the time they take to find a solution based on their input size. Some algorithms take linear time complexity ($O(n)$) and others may take exponential time, and some never halt. Note that, some problems may have multiple algorithms with different complexities.

Randomized Algorithms

Few algorithms make choices randomly. For some problems the fastest solutions must involve randomness. Example: Quick sort.

Branch and Bound Enumeration and Backtracking

These were used in Artificial Intelligence and we do not need to explore these fully. For backtracking method refer *Recusion and Backtracking* chapter.

Note: In the next few chapters we discuss these [greedy, divide and conquer and dynamic programming] design techniques. The importance was given to these techniques as the numbers of problems solved with these techniques are more compared to other.

Chapter-17

GREEDY ALGORITHMS

17.1 Introduction

Let us start our discussion with simple theory which will give us an idea about the greedy technique. In *Chess* game, every time while making a decision we need to think about the future consequences as well. Where as, in *Tennis* (or *Volley Ball*) game, we just need to act based on current situation which looks good at that moment, without bothering about the future consequences. This means that in some cases making a decision which looks good at that situation gives the best solution (*Greedy*) and for others it's not. Greedy technique best suits for the second class of problems.

17.2 Greedy strategy

Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future. This means that some *local best* is chosen. It assumes local good selection makes global optimal solution.

17.3 Elements of Greedy Algorithms

The two basic properties of optimal greedy algorithms are:

1) Greedy choice property
2) Optimal substructure

Greedy choice property

This property says that globally optimal solution can be obtained by making a locally optimal solution (greedy). The choice made by a greedy algorithm may depend on earlier choices but not on future. It iteratively makes one greedy choice after another and reduces the given problem into a smaller one.

Optimal substructure

A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems. That means we can solve subproblems and build up the solutions to solve larger problems.

17.4 Does Greedy Works Always?

Making locally optimal choices does not work always. Hence, greedy algorithms will not give best solutions always. We will see such examples in *Problems* section and in *Dynamic Programming* chapter.

17.5 Advantages and Disadvantages of Greedy Method

The main advantage of greedy method is that they are straightforward, easy to understand and easy to code. In greedy algorithms, once we make a decision, we do not have to spend time in re-examining already computed values.
Its main disadvantage is that for many problems there is no greedy algorithm. That means, in many cases there is no guarantee that making locally optimal improvements in a locally optimal solution gives the optimal global solution.

17.6 Greedy Applications

- Sorting: Selection sort, Topological sort
- Priority Queues: Heap sort

- Huffman coding compression algorithm
- Prim's and Kruskal's algorithms
- Shortest path in Weighted Graph [Dijkstra's]
- Coin change problem
- Fractional Knapsack problem
- Disjoint sets-UNION by size and UNION by height (or rank)
- Job scheduling algorithm
- Greedy techniques can be used as approximation algorithm for complex problems

17.7 Understanding Greedy Technique

For better understanding let us go through an example. For more details, refer the topics of *Greedy* applications.

Huffman coding algorithm

Definition: Given a set of n characters from the alphabet A [each character c $\in$ A] and their associated frequency freq(c), find a binary code for each character c $\in$ A, such that $\sum_{c \in A}$ freq(c)|binarycode(c)| is minimum, where |binarycode(c)| represents the length of binary code of character c. That means sum of lengths of all character codes should be minimum [sum of each characters frequency multiplied by number of bits in the representation].

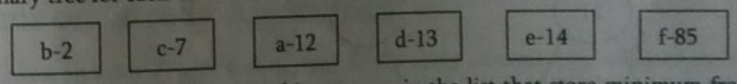
The basic idea behind Huffman coding algorithm is to use fewer bits for more frequently occurring characters. Huffman coding algorithm compresses the storage of data using variable length codes. We know that each character takes 8 bits for representation. But in general, we do not use all of them. Also, we use some characters more frequently than others. When reading a file, generally system reads 8 bits at a time to read a single character. But this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character $'e'$ is used 10 times more frequently than the character $'q'$. It would then be advantageous for us to use a 7 bit code for e and a 9 bit code for q instead because that could reduce our overall message length.

On average, using Huffman coding on standard files can reduce them anywhere from 10% to 30% depending to the character frequencies. The idea behind the character coding is to give longer binary codes for less frequent characters and groups of characters. Also, the character coding is constructed in such a way that no two character codes are prefixes of each other.

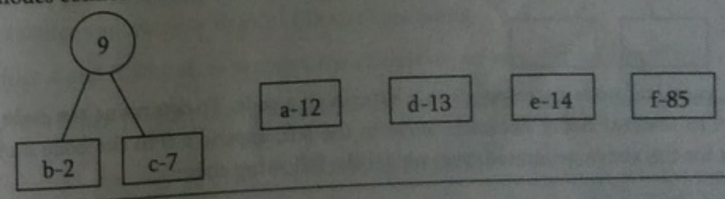
Example: Let's assume that after scanning a file we found the following character frequencies:

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

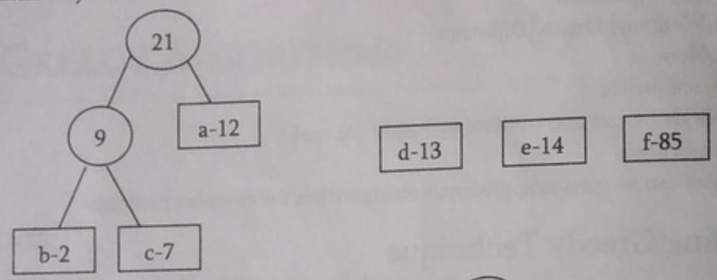
In this, create a binary tree for each character that also stores the frequency with which it occurs (as shown below).

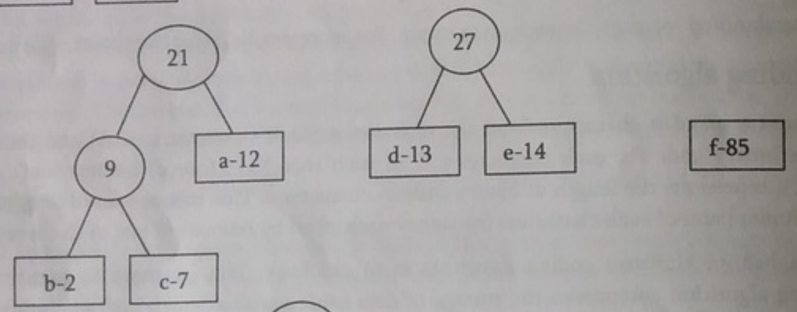


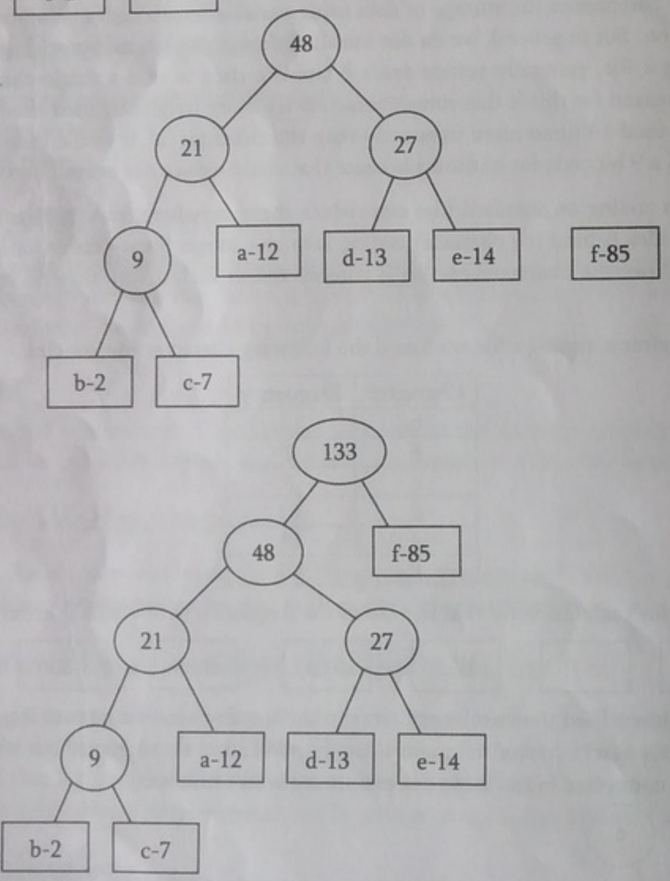
The algorithm works as follows: Find the two binary trees in the list that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store no character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like follows:



Repeat this process until only one tree is left:







Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, traverse from the root to the leaf node. For each move to the left, append a 0 to the code and for each move right append a 1. As a result for the above generated tree, we get the following codes:

Letter	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

Calculating Bits Saved: Now, let us see how many bits that Huffman coding algorithm is saving. All we need to do for this calculation is see how many bits are originally used to store the data and subtract from that how many bits are used to store the data using the Huffman code.

In the above example, since we have six characters, let's assume each character is stored with a three bit code. Since there are 133 such characters (multiply total frequencies with 3), the total number of bits used is $3 * 133 = 399$. Using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

Thus, we saved $399 - 238 = 161$ bits, or nearly 40% storage space.

```
HuffmanCodingAlgorithm(int A[], int n) {
        Initialize a priority queue, PQ, to contain the n elements in A;
        struct BinaryTreeNode *temp;
        for (i = 1; i<n; i++) {
                temp  = (struct *)malloc(sizeof(BinaryTreeNode));
                temp→left = Delete-Min(PQ);
                temp→right = Delete-Min(PQ);
                temp→data = temp→left→data + temp→right→data;
                Insert temp to PQ;
        }
        return PQ;
}
```

Time Complexity: $O(nlogn)$, since there will be *one* build_heap, $2n - 2$ delete_mins, and $n - 2$ inserts, on a priority queue that never has more than n elements. Refer *Priority Queues* chapter for details.

17.8 Problems on Greedy Algorithms

Problem-1 Given an array F with size n. Assume the array content $F[i]$ indicates the length of the i^{th} file and we want to merge all these files into one single file. Check whether the following algorithm gives the best solution for this problem or not?

Algorithm: Merge the files contiguously. That means select the first two files and merge them. Then select the output of previous merge and merge with third file and keep going.

Note: Given two files A and B with sizes m and n, the complexity of merging is $O(m + n)$.

Solution: This algorithm will not produce the optimal solution. For counter example, let us consider the following file sizes array.

$F = \{10,5,100,50,20,15\}$

As per the above algorithm, we need to merge the first two files (10 and 5 size files) and as a result we get the following list of files. In the below list, 15 indicates the cost of merging two files with sizes 10 and 5.

{15,100,50,20,15}

Similarly, merging 15 with next file 100 produces: {115,50,20,15}. For the subsequent steps the list becomes,

{165,20,15}, {185,15}

Finally, {200}

The total cost of merging = Cost of all merging operations = $15 + 115 + 165 + 185 + 200 = 680$.

To see whether the above result is optimal or not, consider the order: $\{5, 10, 15, 20, 50, 100\}$. For this example, following the same approach, the total cost of merging = $15 + 30 + 50 + 100 + 200 = 395$. So, the given algorithm is not giving the best (optimal) solution.

Problem-2 Similar to Problem-1, does the following algorithm gives optimal solution?

Algorithm: Merge the files in pairs. That means after the first step, the algorithm produces the $n/2$ intermediate files. For the next step, we need to consider these intermediate files and merge them in pairs and keep going.

Note: Sometimes this algorithm is called 2-way merging. Instead of two files at a time, if we merge K files at a time then we call it as K-way merging.

Solution: This algorithm will not produce the optimal solution and consider the same previous example for counter example. As per the above algorithm, we need to merge the first pair of files (10 and 5 size files), second pair of files (100 and 50) and third pair of files (20 and 15). As a result we get the following list of files.

$\{15, 150, 35\}$

Similarly, merge the output in pairs and this step produces: In the below, the third element does not have pair element, so keep it same.

{165,35}

Finally, {185}

The total cost of merging = Cost of all merging operations = $15 + 150 + 35 + 165 + 185 = 550$. This is much more than 395 (of the previous problem). So, the given algorithm is not giving the best (optimal) solution.

Problem-3 For the Problem-1, what is the best way to merge them into a single file?

Solution: Using greedy algorithm we can reduce the total time for merging the given files. Let us consider the following algorithm.

Algorithm

1. Store file sizes in a priority queue. The key of elements are file lengths.
2. Repeat the following until there is only one file:
   a. Extract two smallest elements X and Y.
   b. Merge X and Y and insert this new file in the priority queue.

Variant of same algorithm:

1. Sort the file sizes in ascending order.
2. Repeat the following until there is only one file:
   a. Take first two elements (smallest) X and Y.
   b. Merge X and Y and insert this new file in the sorted list.

To check the above algorithm, let us trace it with previous example. The given array is:

$F = \{10,5,100,50,20,15\}$

As per the above algorithm, sorting the list it becomes: $\{5, 10, 15, 20, 50, 100\}$. We need to merge the two smallest files (5 and 10 size files) and as a result we get the following list of files. In the below list, 15 indicates the cost of merging two files with sizes 10 and 5.

{15,15,20,50,100}

Similarly, merging two smallest elements (15 and 15) produces: {20,30,50,100}. For the subsequent steps the list becomes,

{50,50,100} //merging 20 and 30

{100,100} //merging 20 and 30

Finally, {200}

The total cost of merging = Cost of all merging operations = $15 + 30 + 50 + 100 + 200 = 395$. So, this algorithm is producing the optimal solution for this merging problem.

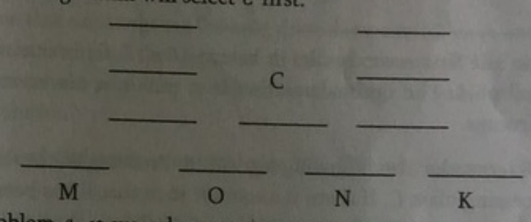
Time Complexity: $O(nlogn)$ time using heaps to find best merging pattern plus the optimal cost of merging the files.

Problem-4 **Interval Scheduling Algorithm:** Given a set of n intervals $S = \{(start_i, end_i) | 1 \le i \le n\}$. Let us assume that we want to find a maximum subset S' of S such that no pair of intervals in S' overlaps. Check whether the following algorithm works or not.

Algorithm:

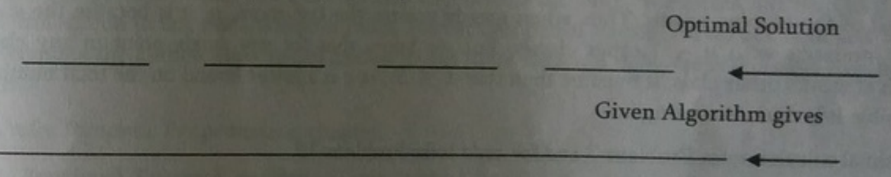
```
while (S is not empty) {
        Select the interval I that overlaps the least number of other intervals.
        Add I to final solution set S'.
        Remove all intervals from S that overlap with I.
}
```

Solution: This algorithm does not solve the problem of finding a maximum subset of non-overlapping intervals. Consider the following intervals. The optimal solution is $\{M, O, N, K\}$. However, the interval that overlaps with the fewest others is C, and the given algorithm will select C first.



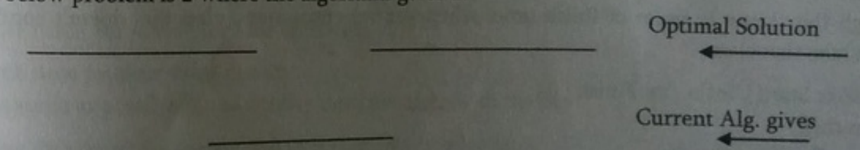
Problem-5 For the Problem-4, if we select the interval that starts earliest (also not overlapping with already chosen intervals), does it gives optimal solution?

Solution: No. It will not give optimal solution. Let us consider the below example. It can be seen that optimal solution to the below problem is 4 where as the given algorithm gives 1.



Problem-6 For the Problem-4, if we select the shortest interval (but is not overlapping the already chosen intervals), does it gives optimal solution?

Solution: This also will not give optimal solution. Let us consider the below example. It can be seen that optimal solution to the below problem is 2 where the algorithm gives 1.



Problem-7 For the Problem-4, what is the optimal solution?

Solution: Now, let us concentrate on the optimal greedy solution.

Algorithm:

Sort intervals according to the right-most ends [end times];
for every consecutive interval [

- If the left-most end is after the right-most end of the last selected interval then we select this interval
- Otherwise we skip it and go to the next interval

}

Time complexity = Time for sorting + Time for scanning = $O(nlogn + n) = O(nlogn)$.

Problem-8 Consider the following problem.

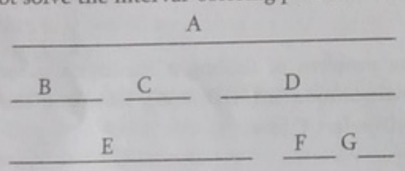
Input: $S = \{(start_i, end_i) | 1 \leq i \leq n\}$ of intervals. The interval $(start_i, end_i)$, we can treat as a request for a room for a class with time $start_i$ to time end_i.

Output: Find an assignment of classes to rooms that uses the fewest number of rooms.

Consider the following iterative algorithm. Assign as many classes as possible to the first room, then assign as many classes as possible to the second room, then assign as many classes as possible to the third room, etc. Does this algorithm give the best solution?

Note: In fact, this problem is similar to interval scheduling algorithm. The only difference is the application.

Solution: This algorithm does not solve the interval-coloring problem. Consider the following intervals:



Maximizing the number of classes in the first room results in having $\{B, C, F, G\}$ in one room, and classes A, D, and E each in their own rooms, for a total of 4. The optimal solution is to put A in one room, $\{B, C, D\}$ in another, and $\{E, F, G\}$ in another, for a total of 3 rooms.

Problem-9 For the Problem-8, consider the following algorithm. Process the classes in increasing order of start times. Assume that we are processing class C. If there is a room R such that R has been assigned to an earlier class, and C can be assigned to R without overlapping previously assigned classes, then assign C to R. Otherwise, put C in a new room. Does this algorithm solve the problem?

Solution: This algorithm solves the interval-coloring problem. Note that if the greedy algorithm creates a new room for the current class c_i, then because it examines classes in order of start times, c_i start point must intersect with the last class in all of the current rooms. Thus when greedy creates the last room, n, it is because the start time of the current class intersects with $n - 1$ other classes. But we know that for any single point in any class it can only intersect with at most s other class, it must be then that $n \leq S$. As s is a lower bound on the total number needed and greedy is feasible it is thus also optimal.

Note: For optimal solution refer Problem-7 and for code refer Problem-10.

Problem-10 Suppose we are given two arrays $Start[1..n]$ and $Finish[1..n]$ listing the start and finish times of each class. Our task is to choose the largest possible subset $X \in \{1, 2, \ldots, n\}$ so that for any pair $i, j \in X$, either $Start\ [i] > Finish[j]$ or $Start\ [j] > Finish\ [i]$

Solution: Our aim is to finish the first class as early as possible, because that leaves us with the most remaining classes. We scan through the classes in order of finish time, whenever we encounter a class that doesn't conflict with latest class so far then take that class.

```
int LargestTasks(int Start[], int n,  int Finish []) {
        sort Finish[];
        rearrange Start[] to match;
        count = 1;
        X[count] = 1;
        for (i = 2; i<n; i++) {
                if(Start[i] > Finish[X[count]])      {
                        count = count + 1;
```



```
                X[count] = I;
        }
    }
    return X[1 .. count];
}
```

This algorithm clearly runs in $O(nlogn)$ time due to sorting.

Problem-11 Consider the making change problem in country India. The input to this problem is an integer M. The output should be the minimum number of coins to make M rupees of change. In India, assume the available coins are $1, 5, 10, 20, 25, 50$ rupees. Assume that we have an unlimited number of coins of each type.

For this problem, does the following algorithm produce optimal solution or not? Take as many coins as possible from the highest denominations. So for example, to make change for 234 rupees the greedy algorithms would take four 50 rupee coins, one 25 rupee coin, one 5 rupee coin, and four 1 rupee coins.

Solution: The greedy algorithm is not optimal for the problem of making change with the minimum number of coins when the denominations are $1, 5, 10, 20, 25$, and 50. In order to make 40 rupees, the greedy algorithm would use three coins of 25, 10, and 5 rupees. The optimal solution is to use two 20-shilling coins.

Note: For optimal solution, refer *Dynamic Programming* chapter.

Problem-12 Let us assume that we are going for long drive between cities A and B. In preparation for our trip, we have downloaded a map that contains the distances in miles between all the petrol stations in our route. Assume that our cars petrol tank can hold petrol for n miles. Assume that the value n is given. Suppose if we stop at every point, does it gives best solution?

Solution: Here the algorithm does not produce optimal solution. Obvious Reason: filling at each petrol station does not produce optimal solution.

Problem-13 For the problem Problem-12, stop if and only if you don't have enough petrol to make it to the next gas station, and if you stop, fill the tank up all the way. Prove or disprove that this algorithm correctly solves the problem.

Solution: The greedy approach works: We start our trip in A with a full tank. We check our map to determine the farthest away petrol station in our route within n miles. Stop at that petrol station, fill up our tank and we check our map again to determine the farthest away petrol station in our route within n miles from this stop. Repeat the process until we get to B.

Note: For code, refer *Dynamic Prograaming* chapter.

Problem-14 **Fractional Knapsack problem:** Given items $t_1, t_2, \ldots, t_n$ (items we might want to carry in our backpack) with associated weights $s_1, s_2, \ldots, s_n$ and benefit values $v_1, v_2, \ldots, v_n$, how can we maximize the total benefit considering that we are subject to an absolute weight limit C?

Solution:

Algorithm:

1) Compute value per size density for each item $d_i = \frac{v_i}{s_i}$.
2) Sort each item by their value density.
3) Take as much as possible of the density item not already in the bag

Time Complexity: $O(nlogn)$ for sorting and $O(n)$ for greedy selections.

Note: The items can be entered into a priority queue and retrieved one by one until either the bag is full or all items have been selected. This actually has a better runtime of $O(n + clogn)$ where c is the number of items that actually get selected in the solution. There is a savings in runtime if $c = O(n)$, but otherwise there is no change in the complexity.

Problem-15 **Number of railway-platforms:** At a rail-station, we have time-table of trains arrival and departure. We need to find the minimum number of platforms so that all the trains can be accommodated as per their schedule.

Example: Time table is like below, the answer is 3. Otherwise, the rail-station will not be able to accommodate all the trains.

Rail	Arrival	Departure
Rail A	0900 hrs	0930 hrs
Rail B	0915 hrs	1300 hrs
Rail C	1030 hrs	1100 hrs
Rail D	1045 hrs	1145 hrs

Solution: Let's take the same example as described above. Calculating the number of platforms is nothing but the maximum number of trains at the rail-station at any time.

First, sort all the arrival(A) and departure(D) time in an array. Then, save the corresponding arrival or departure in the array also. After sorting our array will look like this:

0900	0915	0930	1030	1045	1100	1145	1300
A	A	D	A	A	D	D	D

Now modify the array by placing 1 for A and -1 for D. The new array will look like:

1	1	-1	1	1	-1	-1	-1

Finally make a cumulative array out of this:

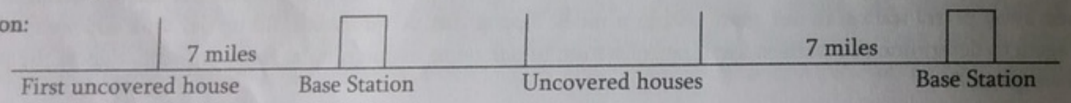
1	2	1	2	3	2	1	0

Our solution will be the maximum value in this array. Here it is 3.

Note: If we have a train arriving and another departing at same time then put departure time first in sorted array.

Problem-16 Consider a country with very long roads and houses along the road. Assume that the residents of all houses use cell phone. We want to place cell phone towers along the road. Each cell phone towers covers 7 kilo meters range. Give an efficient algorithm that gives few cell phone towers.

Solution:



The algorithm to locate the least number of cell phone towers:

1) Start from the beginning of the road
2) Find the first uncovered house on the road
3) If there is no such house, terminate this algorithm, otherwise, go to next step
4) Locate a cell phone tower at 7 miles away after we find this house along the road
5) Go to step 2

Problem-17 **Preparing Songs Cassette** Suppose we have a set of n songs and want to store on a tape. In the future, users will want to read those songs from the tape. Reading a song from tape is not like reading from disk, first we have to fast-forward past all the other songs, and that takes a significant amount of time. Let $A[1..n]$ be an array listing the lengths of each song, specifically, song i has length $A[i]$. If the songs are stored in order from 1 to n, then the cost of accessing the k^{th} song is:

$$C(k) = \sum_{i=1}^{k} A[i]$$

The cost reflects the fact that before we read song k we must first scan past all the earlier songs on the tape. If we change the order of the songs on the tape, we change the cost of accessing the songs, some songs become more expensive to read, but others become cheaper. Different song orders are likely to result in different expected costs.

If we assume that each song is equally likely to be accessed, which order should we use if we want the expected cost to be as small as possible?

Solution: The answer is simple and trivial. We should store the songs in the order from shortest to longest. Storing the small songs at the beginning reduces the forwarding times for remaining jobs.

Problem-18 Let us consider a set of events at *HITEX* (*Hyderabad Convention Center*). Assume that there are n events where each takes one unit of time. Event i will provide a profit of P[i] rupees (P[i] > 0) if started at or before time $T[i]$, where $T[i]$ is an arbitrary number. If an event is not started by $T[i]$ then there is no benefit in scheduling it at all. All events can start as early as time 0. Give the efficient algorithm to find a schedule that maximizes the profit.

Solution:

Algorithm:

- Sort the jobs according to floor($T[i]$) (sorted from largest to smallest).
- Let time t is the current time being considered (where initially t = floor($T[i]$)).
- All jobs i where floor($T[i]$) $= t$ are inserted into a priority queue with the profit g_i used as the key.
- An *DeleteMax* is performed to select the job to run at time t.
- Then t is decremented and the process is continued.

Clearly the time complexity is O($nlogn$). (The sort takes O($nlogn$) and there are at most n insert and *DeleteMax* operations performed on the priority queue, each which takes O($logn$) time.

Problem-19 Let us consider a customer-care server (say, mobile customer-care) with n customers to be severed in the queue. For simplicity assume that the service time required by each customer is known in advance and it is w_i minutes for customer i. So if, for example, the customers are served in order of increasing i, then the i^{th} customer has to wait: $\sum_{j=1}^{n-1} w_j$ *minutes*. The total waiting time of all customers can be given as $= \sum_{i=1}^{n} \sum_{j=1}^{i-1} w_j$. What is the best way to serve the customers so that the total waiting time can be reduced?

Solution: This problem can be easily solved using greedy technique. Since our objective is to reduce the total waiting time, what we can do is, select the customer whose service time is less. That means, if we process the customers in the increasing order of service times then we can reduce the total waiting time.

Time Complexity: O($nlogn$).

Chapter-18

DIVIDE AND CONQUER ALGORITHMS

18.1 Introduction

In *Greedy* chapter, we have seen that for many problems *Greedy* strategy failed to provide optimal solutions. Among those problems, there are some problems which can be easily solved by using Divide & Conquer (D & C) technique. Divide & Conquer is an important algorithm design technique based on recursion. D & C algorithms works by recursively breaking down a problem into two or more subproblems of the same type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

18.2 What is Divide and Conquer Strategy?

The D & C strategy solves a problem by:

1) *Divide*: Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
2) *Recursion*: Recursively solving these sub problems.
3) *Conquer*: Appropriately combining their answers.

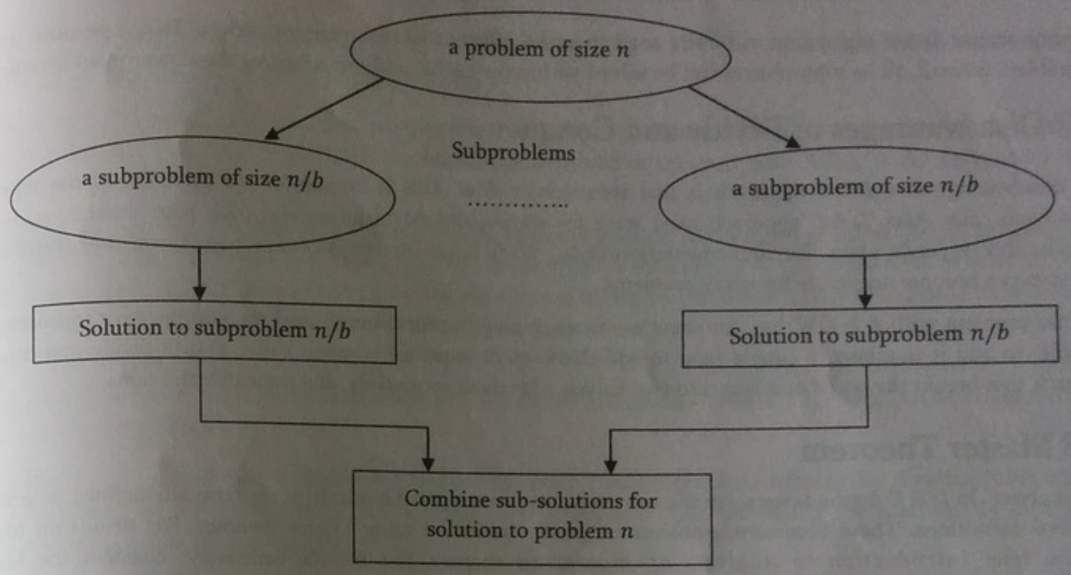
18.3 Does Divide and Conquer Work Always?

It's not possible to solve all the problems with Divide & Conquer technique. As per the definition of D & C the recursion solves the subproblems which are of same type. For all problems it is not possible to find the subproblems which are same size and D & C is not a choice for all problems.

18.4 Divide and Conquer Visualization

For better understanding, consider the following visualization. Assume that n is the size of original problem. As described above, we can see that the problem is divided into subproblems with each of size n/b (for some constant b). We solve the subproblems recursively and combine their solutions to get the original problems solution.

```
DivideAndConquer ( P ) {
        if( small ( P ) )
        // P is very small so that a solution is obvious
                return solution ( n );
        divide the problem P into k sub problems P1, P2, ..., Pk;
        return (
                Combine (
                        DivideAndConquer ( P1 ),
                        DivideAndConquer ( P2 ),
                        ...
                        DivideAndConquer ( Pk )
                        )
                );
}
```

18.5 Understanding Divide and Conquer

For clear understanding of D & C, let us consider our childhood story. There is a old man who is a rich farmer and had seven sons. He was afraid that when he died, his land and his possessions would be divided among his seven sons, and that they would quarrel with one another.

So he gathered them together and showed them seven sticks that he had tied together and told them that anyone who could break the bundle would inherit everything. They all tried, but no one could break the bundle. Then the old man untied the bundle and broke the sticks one by one. The brothers decided that they should stay together and work together and succeed together. The moral for problem solvers is different. If we can't solve the problem, divide it into parts, and solve one part at a time.

In earlier chapters we have already solved many problems based on D & C strategy: like Binary Search, Merge Sort, Quick Sort, etc.... Refer those topics to get an idea of how D & C works. Below are few other real-time problems which can easily be solved with D & C strategy. For all these problems we can find the subproblems which are similar to original problem.

1. Looking a name in a phone book: We have a telephone phone book with names in alphabetical order. Given a name, how do we find whether that name is there in the phone or not?
2. Breaking a stone into dust: We want to convert a stone into dust (very small stones).
3. Finding the exit from within a hotel: We are at the end of a very long hotel lobby with a long series of doors, with one door next to you. We are looking for the door that leads to the exit.
4. Finding our car in a parking lot.

18.6 Advantages of Divide and Conquer

Solving difficult problems: D & C is a powerful method for solving difficult problems. As an example, consider Tower of Hanoi problem. This requires breaking the problem into subproblems, solving the trivial cases and combining subproblems to solve the original problem. Dividing the problem into subproblems so that the subproblems can be combined again is the major difficulty in designing a new algorithm. For many such problems D & C provides simple solution.

Parallelism: Since D & C allows us to solve the subproblems independently, they allow execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because different subproblems can be executed on different processors.

Memory access: D & C algorithms naturally tend to make efficient use of memory caches. This is because once a subproblem is small, all its subproblems can be solved within the cache, without accessing the slower main memory.

18.7 Disadvantages of Divide and Conquer

One disadvantage of a D & C approach is that recursion is slow. This is because of the overhead of the repeated subproblem calls. Also, D & C approach need stack for storing the calls (the state at each point in the recursion). Actually this depends upon the implementation style. With large enough recursive base cases, the overhead of recursion can become negligible for many problems.

Another problem with D & C is that, for some problems, it may be more complicated than an iterative approach. For example, to add n numbers, a simple loop to add them up in sequence is much easier than a divide-and-conquer approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums.

18.8 Master Theorem

As said above, in D & C method, we solve the sub problems recursively. All problems are generally defined in terms of recursive definitions. These recursive problems can easily be solved using Master theorem. For details on Master theorem refer *Introduction to Analysis of Algorithms* chapter. Just for the continuity, consider the Master theorem.

If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, where $a \geq 1$, $b > 1, k \geq 0$ and p is a real number, then the complexity can be directly given as:

1) If $a > b^k$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

18.9 Divide and Conquer Applications

- Binary Search
- Merge Sort
- Quick Sort
- Median Finding
- Min and Max Finding
- Matrix Multiplication
- Closest Pair problem

18.10 Problems on Divide and Conquer

Problem-1 Let us consider an algorithm A which solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time. What is the complexity of this algorithm?

Solution: Let us assume that the input size is n and $T(n)$ defines the solution to the given problem. As per the description, algorithm is dividing the problem in to 5 sub problems with each of size $\frac{n}{2}$. So we need to solve $5T(\frac{n}{2})$ subproblems. After solving these subproblems it's scanning the given array (linear time) to combine these solutions.

The total recurrence algorithm for this problem can be given as: $T(n) = 5T\left(\frac{n}{2}\right) + O(n)$. Using Master theorem (of D & C), we get the complexity as $O(n^{log_2^5}) \approx O(n^{2+}) \approx O(n^3)$.

Problem-2 Similar to Problem-1, an algorithm B solves problems of size n by recursively solving two subproblems of size $n - 1$ and then combining the solutions in constant time. What is the complexity of this algorithm?

Solution: Let us assume that input size is n and $T(n)$ defines the solution to the given problem. As per the description of algorithm we are dividing the problem in to 2 sub problems with each of size n − 1. So we need to solve $2T(n-1)$ subproblems. After solving these subproblems, the algorithm is taking only a constant time to combine these solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T(n-1) + O(1)$$

Using Master theorem (of *Subtract and Conquer*), we get the complexity as $O\left(n^0 2^{\frac{n}{1}}\right) = O(2^n)$. (Refer *Introduction* chapter for more details).

Problem-3 Again similar to Problem-1, another algorithm C solves problems of size n by dividing them into nine subproblems of size $\frac{n}{3}$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time. What is the complexity of this algorithm?

Solution: Let us assume that input size is n and $T(n)$ defines the solution to the given problem. As per the description of algorithm we are dividing the problem in to 9 sub problems with each of size $\frac{n}{3}$. So we need to solve $9T(\frac{n}{3})$ subproblems. After solving the subproblems, the algorithm is taking quadratic time to combine these solutions. The total recurrence algorithm for this problem can be given as: $T(n) = 9T\left(\frac{n}{3}\right) + O(n^2)$. Using D & C Master theorem, we get the complexity as $O(n^2 logn)$.

Problem-4 Write a recurrence and solve it.

```
void function(n) {
        if(n > 1) {
                printf("*");
                function(n/2);
                function(n/2);
        }
}
```

Solution: Let us assume that input size is n and $T(n)$ defines the solution to the given problem. As per the given code after printing the character, and dividing the problem in to 2 subproblems with each of size $\frac{n}{2}$ and solving them. So we need to solve $2T(\frac{n}{2})$ subproblems. After solving these subproblems, the algorithm is not doing anything for combining the solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

Using Master theorem (of D & C), we get the complexity as $O\left(n^{log_2^2}\right) \approx O(n^1) = O(n)$.

Problem-5 Given an array, give an algorithm for finding the maximum and minimum.

Solution: Refer *Selection Algorithms* chapter.

Problem-6 Discuss Binary Search and its complexity.

Solution: Refer *Searching* chapter for discussion on Binary Search.

Analysis: Let us assume that input size is n and $T(n)$ defines the solution to the given problem. Since the elements are in sorted order. In binary search we take the middle element and check whether the element to be search is equal to that element or not. If it is equal then we return that element.

If the element to be searched is greater than the middle element then we consider the left sub-array for finding the element and discard the right sub-array. Similarly, if the element to be searched is less than the middle element then we consider the right sub-array for finding the element and discard the left sub-array.
What this means is, in both the cases we are discarding half of the sub-array and considering the remaining half only. Also, at every iteration we are dividing the elements into two equal halfs.

As per the above discussion every time we are dividing the problem in to 2 sub problems with each of size $\frac{n}{2}$ and need to solve one $T(\frac{n}{2})$ subproblem. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

Using Master theorem (of D & C), we get the complexity as $O(logn)$.

Problem-7 Consider the modified version of binary search. Let us assume that the array is divided into 3 equal parts (ternary search) instead of two equal parts. Write the recurrence for this ternary search and find its complexity.

Solution: From Problem-5 discussion, binary search has the recurrence relation: $T(n) = T\left(\frac{n}{2}\right) + O(1)$. Similar to Problem-5 discussion, instead of "2" in the recurrence relation we need use "3". That indicates that we are dividing the array into 3 sub-arrays with equal and considering only one of them. So, the recurrence for the ternary search can be given as:

$$T(n) = T\left(\frac{n}{3}\right) + O(1)$$

Using Master theorem (of D & C), we get the complexity as $O(log_3^n) \approx O(logn)$ (we don't have to worry about the base of log as they are constants).

Problem-8 For Problem-5, what if we divide the array into two sets of sizes approximately one-third and two-thirds.

Solution: We now consider a slightly modified version of ternary search in which only one comparison is made which creates two partitions, one of roughly $\frac{n}{3}$ elements and the other of $\frac{2n}{3}$. Here the worst case comes when the recursive call is on the larger $\frac{2n}{3}$ element part. So the recurrence corresponding to this worst case is:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Using Master theorem (of D & C), we get the complexity as $O(logn)$. It is interesting to note that we will get the same results for general k-ary search (as long as k is a fixed constant which does not depend on n) as n approaches infinity.

Problem-9 Discuss Merge Sort and its complexity.

Solution: Refer *Sorting* chapter for discussion on Merge Sort. In Merge Sort, if the number of elements are greater than 1 then divide them into two equal subsets, the algorithm is recursively invoked on the subsets, and the returned sorted subsets are merged to provide a sorted list of the original set. The recurrence equation of the Merge Sort algorithm is:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n), if\ n > 1 \\ 0 \qquad\qquad\quad , if n = 1 \end{cases}$$

If we solve this recurrence using D & C Master theorem gives O(nlogn) complexity.

Problem-10 Discuss Quick Sort and its complexity.

Solution: Refer *Sorting* chapter for discussion on Quick Sort. For Quick Sort we have different complexities for best case and worst case.

Best Case: In Quick Sort, if the numbers of elements are greater than 1 then dividing them into two equal subsets, the algorithm is recursively invoked on the subsets. After solving the sub problems we don't need to combine them. This is because in Quick Sort they were already in sorted order. But, we need scan the complete elements to partition the elements. The recurrence equation of the Quick Sort best case is

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n), if\ n > 1 \\ 0 \qquad\qquad\quad , if n = 1 \end{cases}$$

If we solve this recurrence using Master theorem of D & C gives $O(nlogn)$ complexity.

Worst Case: In the worst case, Quick Sort divides the input elements into two sets and one of them contains only one element. That means other set has $n-1$ elements to be sorted. Let us assume that the input size is n and $T(n)$ defines the solution to the given problem. So we need to solve $T(n-1)$, $T(1)$ subproblems. But to divide the input into two sets Quick Sort needs one scan of the input elements (this takes $O(n)$).

After solving these subproblems the algorithm is taking only a constant time to combine these solutions. The total recurrence algorithm for this problem can be given as: $T(n) = T(n-1) + O(1) + O(n)$.

This is clearly a summation recurrence equation. So, $T(n) = \frac{n(n+1)}{2} = O(n^2)$.

Note: For the average case analysis, refer *Sorting* chapter.

Problem-11 Given an infinite array in which the first n cells contain integers in sorted order and the rest of the cells are filled with some special symbol (say, \$). Assume we do not know the n value. Give an algorithm that takes an integer K as input and finds a position in the array containing K, if such a position exists, in $O(logn)$ time.

Solution: Since we need an $O(logn)$ algorithm, we should not search for all the elements of the given list (which gives $O(n)$ complexity). To get $O(logn)$ complexity one possibility is to use binary search. But in the given scenario we cannot use binary search as we do not know the end of list. Our first problem is to find the end of the list. To do that, we can start at first element and keep searching with doubled index. That means we first search at index 1 then, 2, 4, 8 ...

```
int FindInInfiniteSeries(int A[]) {
        int mid, l = r = 1;
        while( A[r] != '$') {
                l = r;
                r = r × 2;
        }
        while( (r − l > 1 ) {
                mid = (r − l)/2 + l;
                if( A[mid] == '$')
                        r = mid;
                else    l = mid;
        }
}
```

It is clear that, once we identified a possible interval $A[i,\ldots,2i]$ in which K might be, its length is at most n (since we have only n numbers in the array A), so searching for K using binary search takes $O(logn)$ time.

Problem-12 Given a sorted array of non-repeated integers $A[1..n]$, check whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(logn)$.

Solution: We can't use binary search on the array as it is. If we want to keep the $O(logn)$ property of the solution we have to impelemnt our own binary search. If we modify the array (in place or in a copy) and subtract i from A[i], we can then use binary search. The complexity for doing so is $O(n)$.

Problem-13 We are given two sorted lists of size n. Give an algorithm for finding the median element in the union of the two lists.

Solution: Using Merge Sort process. Use *merge* procedure of merge sort (refer *Sorting* chapter). Keep track of count while comparing elements of two arrays. If count becomes n (since there are $2n$ elements), we have reached the median. Take the average of the elements at indexes $n-1$ and n in the merged array.

Time Complexity: $O(n)$.

Problem-14 Can we give algorithm if the sizes of two lists are not same?

Solution: The solution is same as previous problem. Let us assume that the lengths of two lists are m and n. In this case we need to stop when counter reaches $(m+n)/2$.

Time Complexity: $O((m+n)/2)$.

Problem-15 Can we improve the time complexity of the Problem-13 to $O(logn)$?

Solution: Yes, using D & C approach. Let us assume that the given two lists are $L1$ and $L2$.

Algorithm:

1. Find the medians of the given sorted input arrays $L1[]$ and $L2[]$. Assume that those medians are $m1$ and $m2$.
2. If $m1$ and $m2$ both are equal then return $m1$ (or $m2$).
3. If $m1$ is greater than $m2$, then the final median will be in below two sub arrays.
4. From first element of $L1$ to $m1$.
5. From $m2$ to last element of $L2$.
6. If $m2$ is greater than $m1$, then median is present in one of the below two sub arrays.
7. From $m1$ to last element of $L1$.
8. From first element of $L2$ to $m2$.
9. Repeat the above process until size of both the sub arrays becomes 2.
10. If size of the two arrays are 2 then use below formula to get the median.
11. Median = $(max(L1[0], L2[0]) + min(L1[1], L2[1])/2$

Time Complexity: $O(logn)$. Since we are considering only half of the input and throwing the remaining half.

Problem-16 Given an input array A. Let us assume that there can be duplicates in the list. Now search for an element in the list in such a way that we get the highest index if there are duplicates.

Solution: Refer *Searching* chapter.

Problem-17 Discuss Strassen's Matrix Multiplication Algorithm using Divide and Conquer. That means, given two $n \times n$ matrices, A and B, compute the $n \times n$ matrix $C = A \times B$, where the elements of C are given by

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} B_{k,j}$$

Solution: Before Strassen's algorithm, first let us see the basic divide and conquer algorithm. The general approach we follow for solving this problem is given below. To determine, $C[i,j]$ we need to multiply the i^{th} row of A with j^{th} column of B.

```
// Initialize C.
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C[i, j] += A[i, k] * B[k, j];
```

The matrix multiplication problem can be solved with D & C technique. To implement a D & C algorithm we need to break the given problem into several subproblems that are similar to the original one. In this instance we view each of the $n \times n$ matrices as a 2×2 matrix, the elements of which are $\frac{n}{2} \times \frac{n}{2}$ submatrices. So, the original matrix multiplication, $C = A \times B$ can be written as:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

where each $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$ is a $\frac{n}{2} \times \frac{n}{2}$ matrix.

From the given definition o f $C_{i,j}$, we get that the result sub matrices can be computed as follows:

$$C_{1,1} = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}$$
$$C_{1,2} = A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2}$$
$$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1}$$
$$C_{2,2} = A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}$$

Here the symbols + and × are taken to mean addition and multiplication (respectively) of $\frac{n}{2} \times \frac{n}{2}$ matrices.

In order to compute the original $n \times n$ matrix multiplication we must compute eight $\frac{n}{2} \times \frac{n}{2}$ matrix products (*divide*) followed by four $\frac{n}{2} \times \frac{n}{2}$ matrix sums (*conquer*). Since matrix addition is an $O(n^2)$ operation, the total running time for the multiplication operation is given by the recurrence:

$$T(n) = \begin{cases} O(1) & , for\ n = 1 \\ 8T\left(\frac{n}{2}\right) + O(n^2) & , for\ n > 1 \end{cases}$$

Using master theorem, we get, $T(n) = O(n^3)$.

Fortunately, it turns out that one of the eight matrix multiplications is redundant (found by Strassen). Consider the following series of seven $\frac{n}{2} \times \frac{n}{2}$ matrices:

$$M_0 = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$$
$$M_1 = (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})$$
$$M_2 = (A_{1,1} - A_{2,1}) \times (B_{1,1} + B_{1,2})$$
$$M_3 = (A_{1,1} + A_{1,2}) \times B_{2,2}$$
$$M_4 = A_{1,1} \times (B_{1,2} - B_{2,2})$$
$$M_5 = A_{2,2} \times (B_{2,1} - B_{1,1})$$
$$M_6 = (A_{21} + A_{2,2}) \times B_{1,1}$$

Each equation above has only one multiplication. Ten additions and seven multiplications are required to compute M_0 through M_6. Given M_0 through M_6, we can compute the elements of the product matrix C as follows:

$$C_{1,1} = M_0 + M_1 - M_3 + M_5$$
$$C_{1,2} = M_3 + M_4$$
$$C_{2,1} = M_5 + M_6$$
$$C_{2,2} = M_0 - M_2 + M_4 - M_6$$

This approach requires seven $\frac{n}{2} \times \frac{n}{2}$ matrix multiplications and 18 $\frac{n}{2} \times \frac{n}{2}$ additions. Therefore, the worst-case running time is given by the following recurrence:

$$T(n) = \begin{cases} O(1) & , for\ n = 1 \\ 7T\left(\frac{n}{2}\right) + O(n^2) & , for\ n = 1 \end{cases}$$

Using master theorem, we get, $T(n) = O\left(n^{log_2^7}\right) = O(n^{2.81})$.

Problem-18 **Stock Pricing Problem:** Consider the stock price of *CareerMonk.com* in n consecutive days. That means the input consists of an array with stock prices of the company. We know that stock price will not be same on all the days. In the input stock prices there may the dates where the stock is high where we can sell the current holdings and there may the days where we can buy the stock. Now our problem is to find the day on which we can buy the stock and the day on which we can sell the stock so that we can make maximum profit out of trading.

Solution: As given in problem let us assume that the input is an array with stock prices [integers]. Let us say the given array is $A[1], \ldots, A[n]$. From this array we have to find two days [one for buy and one for sell] in such a way that we can make maximum profit. Also, another point to make is that buy date should be before sell date. One simple approach is to look at all possible buy and sell dates.

```
void StockStrategy(int A[], int n, int *buyDateIndex, int *sellDateIndex) {
    int j, profit=0;
    *buyDateIndex =0; *sellDateIndex =0;
```

```
    //indicates buy date
    for (int i  = 1; i < n; i++)
        //indicates sell date
        for( j  = i; j < n;  j++)
            if(A[j] - A[i] > profit)      {
                profit = A[j] - A[i];
                *buyDateIndex = i;
                *sellDateIndex = j;
            }
}
```

The two nested loops takes $n(n + 1)/2$ computations, so this takes time $\Theta(n^2)$.

Problem-19 For Problem-18, can we improve the time complexity?

Solution: Yes. Divide-and-Conquer $\Theta(nlogn)$ solution. Divide the input list into two parts and recursively find the solution in both the parts. In this case we get three cases:

- *buyDateIndex* and *sellDateIndex* both are in the earlier time period.
- *buyDateIndex* and *sellDateIndex* both are in the later time period.
- *buyDateIndex* is in the earlier part and *sellDateIndex* is in the later part of the time period.

The first two cases can be solved with recursion. The third case needs care. This is because *buyDateIndex* is one side and *sellDateIndex* is on other side. In this case we need to find the minimum and maximum prices in the two sub-parts and this we can solve in linear-time.

```
void StockStrategy(int A[], int left, int right) {
    //Declare the necessary variables;
    if(left + 1 = right)
        return (0, left, left);
    mid  = left + (right - left) / 2;
    (profitLeft, buyDateIndexLeft, sellDateIndexLeft) =  StockStrategy(A, left, mid);
    (profitRight, buyDateIndexRight, sellDateIndexRight)  = StockStrategy(A, mid, right);

    minLeft  = Min(A, left, mid);
    maxRight  = Max(A, mid, right);
    profit  = A[maxRight] - A[minLeft];
    if(profitLeft > max{profitRight, profit})
        return (profitLeft, buyDateIndexLeft, sellDateIndexLeft);
    else if(profitRight > max{profitLeft, profit})
        return (profitRight, buyDateIndexRight, sellDateIndexRight);
    else    return (profit, minLeft, maxRight);
}
```

Algorithm *StockStrategy* calls itself recursively on two problems of half the size of the input, and in addition spends $\Theta(n)$ time searching for the maximum and minimum prices. So the time complexity is characterized by the recurrence $T(n) = 2T(n/2) + \Theta(n)$ and by the Master theorem we get O(nlogn).

Problem-20 We are testing "unbreakable" laptops and our goal is to find out how unbreakable they really are. In particular, we work in an n-story building and want to find out the lowest floor from which we can drop the laptop without it breaking (call this "the ceiling"). Suppose we are given two laptops and want to find the highest ceiling possible. Give an algorithm that minimizes the number of tries we need to make $f(n)$ (hopefully, $f(n)$ is sub-linear, as a linear $f(n)$ yields a trivial solution).

Solution: For the given problem, we cannot use binary search as we cannot divide the problem and solve them recursively. Let us take some example for understanding the scenario. Let us say 14 is the answer. That means we need

14 drops for finding the answer. First we drop from height 14, if it breaks we try all floors from 1 to 13. If it doesn't break then we have left 13 drops, so we will drop it from $14 + 13 + 1 = 28^{th}$ floor. The reason being if it breaks at 28^{th} floor we can try all the floors from 15 to 27 in 12 drops (total of 14 drops). Now if it did not break then we have left 11 drops and we can figure out whether we can find out whether we can figure out the floor in 14 drops.

From the above example, it can be seen that we first tried with a gap of 14 floors, and then followed by 13 floors, then 12 and so on. So if the answer is k then we are trying the intervals at $k, k-1, k-2 \ldots .1$. Given number of floors is n, we have to relate these two. Since the maximum floor from which we can try is n, the total skips should be less than n. This gives:

$$\begin{aligned} k + (k-1) + (k-2) + \cdots + 1 &\leq n \\ \frac{k(k+1)}{2} &\leq n \\ k &\leq \sqrt{n} \end{aligned}$$

Complexity of this process is $O(\sqrt{n})$.

Problem-21 Given n numbers, check if any two are equal?

Solution: Refer *Searching* chapter.

Problem-22 Give an algorithm to find out if an integer is a square? E.g. 16 is, 15 isn't.

Solution: Initially let us say $i = 2$. Compute the value $i \times i$ and see if it is equal to the given number. If it is equal then we are done otherwise increment the i vlaue. Continue this process until we reach $i \times i$ greater than or equal to the given number.

Time Complexity: $O(\sqrt{n})$. Space Complexity: O(1).

Problem-23 Given an array of $2n$ integers in the following format $a1\ a2\ a3 \ldots an\ b1\ b2\ b3 \ldots bn$. Shuffle the array to $a1\ b1\ a2\ b2\ a3\ b3 \ldots an\ bn$ without any extra memory [MA].

Solution: Let us take an example (for brute force solution refer *Searching* chapter)

1. Start with the array: $a1\ a2\ a3\ a4\ b1\ b2\ b3\ b4$
2. Split the array into two halves: $a1\ a2\ a3\ a4 : b1\ b2\ b3\ b4$
3. Exchange elements around the center: exchange $a3\ a4$ with $b1\ b2$ you get: $a1\ a2\ b1\ b2\ a3\ a4\ b3\ b4$
4. Split $a1\ a2\ b1\ b2$ into $a1\ a2 : b1\ b2$ then split $a3\ a4\ b3\ b4$ into $a3\ a4 : b3\ b4$
5. Exchange elements around the center for each subarray you get: $a1\ b1\ a2\ b2$ and $a3\ b3\ a4\ b4$

Please note that this solution only handles the case when $n = 2^i$ where $i = 0, 1, 2, 3$ etc. In our example $n = 2^2 = 4$ which makes it easy to recursively split the array into two halves. The basic idea behind swapping elements around the center before calling the recursive function is to produce smaller size problems. A solution with linear time complexity may be achieved if the elements are of specific nature for example if you can calculate the new position of the element using the value of the element itself. This is nothing but a hashing technique.

```
void ShuffleArray(int A[], int l, int r) {
    //Array center
    int c = l + (r-l)/2, q = 1 + l + (c-l)/2;
    if(l == r) //Base case when the array has only one element
        return;
    for (int k = 1, i = q; i <= c; i++, k++) {
        //Swap elements around the center
        int tmp = A[i];    A[i] = A[c + k];    A[c + k] = tmp;
    }
    ShuffleArray(A, l, c);              //Recursively call the function on the left and right
    ShuffleArray(A, c + 1, r); );       //Recursively call the function on the right
}
```

Time Complexity: O($nlogn$).

Problem-24 **Nuts and Bolts Problem:** Given a set of n nuts of different sizes and n bolts such that there is a one-to-one correspondence between the nuts and the bolts, find for each nut its corresponding bolt. Assume that we can only compare nuts to bolts (cannot compare nuts to nuts and bolts to bolts).

Solution: Refer *Sorting* chapter.

Problem-25 **Maximum Value Contiguous Subsequence:** Given a sequence of n numbers $A(1) \ldots A(n)$, give an algorithm for finding a contiguous subsequence $A(i) \ldots A(j)$ for which the sum of elements in the subsequence is maximum. **Example:** {-2, 11, -4, 13, -5, 2} → 20 and {1, -3, 4, -2, -1, 6} → 7.

Solution: Divide this input into two halves. The maximum contiguous subsequence sum can occur in one of 3 ways:

- Case 1: It can be completely in the first half
- Case 2: It can be completely in the second half
- Case 3: It begins in the first half and ends in the second half

We begin by looking at case 3. To avoid the nested loop that results from considering all $n/2$ starting points and $n/2$ ending points independently. Replace two nested loops by two consecutive loops. The consecutive loops, each of size $n/2$ combine to require only linear work. Any contiguous subsequence that begins in the first half and ends in the second half must include both the last element of the first half and first element of the second half. What we can do in cases 1 and 2 is apply the same strategy of dividing into more halves. In summary, we do the following:

1. Recursively compute the maximum contiguous subsequence that resides entirely in the first half.
2. Recursively compute the maximum contiguous subsequence that resides entirely in the second half.
3. Compute, via two consecutive loops, the maximum contiguous subsequence sum that begins in the first half but ends in the second half.
4. Choose the largest of the three sums.

```
int MaxSumRec(int A[], int left, int right) {
        int MaxLeftBorderSum = 0, MaxRightBorderSum = 0, LeftBorderSum = 0, RightBorderSum = 0;
        int mid = left + (right - left) / 2;
        if(left == right) // Base Case
                return A[left] > 0 ? A[left] : 0;
        int MaxLeftSum = MaxSumRec(A, left, mid);
        int MaxRightSum = MaxSumRec(A, mid + 1, right);
        for (int i = mid; i >= left; i--) {
                LeftBorderSum += A[i];
                if(LeftBorderSum > MaxLeftBorderSum)
                        MaxLeftBorderSum = LeftBorderSum;
        }
        for (int j = mid + 1; j <= right; j++) {
                RightBorderSum += A[j];
                if(RightBorderSum > MaxRightBorderSum)
                        MaxRightBorderSum = RightBorderSum;
        }
        return Max(MaxLeftSum, MaxRightSum,MaxLeftBorderSum + MaxRightBorderSum);
}
int MaxSubsequenceSum(int A, int n) {
        return n > 0 ? MaxSumRec(A, 0, n – 1) : 0;
}
```

The base case cost is 1. The program performs two recursive calls plus the linear work involved in computing the maximum sum for case 3. The recurrence relation is:

$$T(1) = 1$$

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

Using D & C Master theorem, we get the time complexity as $T(n) = O(nlogn)$.

Note: For efficient solution refer *Dynamic Programming* chapter.

Problem-26 **Closest-Pair of Points:** Given a set of n points $S = \{p_1, p_2, p_3, \ldots, p_n\}$, where $p_i = (x_i, y_i)$. Find the pa of points having smallest distance among all pairs (assume that all points are in one dimension).

Solution: Let us assume that we have sorted the points. Since the points are in one dimension, all the points are in line after we sort them (either on X-axis or Y-axis). The complexity of sorting is O($nlogn$). After sorting we can g through them to find the consecutive points with least difference. So the problem in one dimension is solved i O($nlogn$) time which is mainly dominated by sorting time.

Time Complexity: O($nlogn$).

Problem-27 For the Problem-26, how do we solve if the points are in two dimensional space?

Solution: Before going to algorithm, let us consider the following mathematical equation:

$$distance(p_1, p_2) = \sqrt{(x_1 - x_2)^2 - (y_1 - y_2)^2}$$

The above equation calculates the distance between two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$.

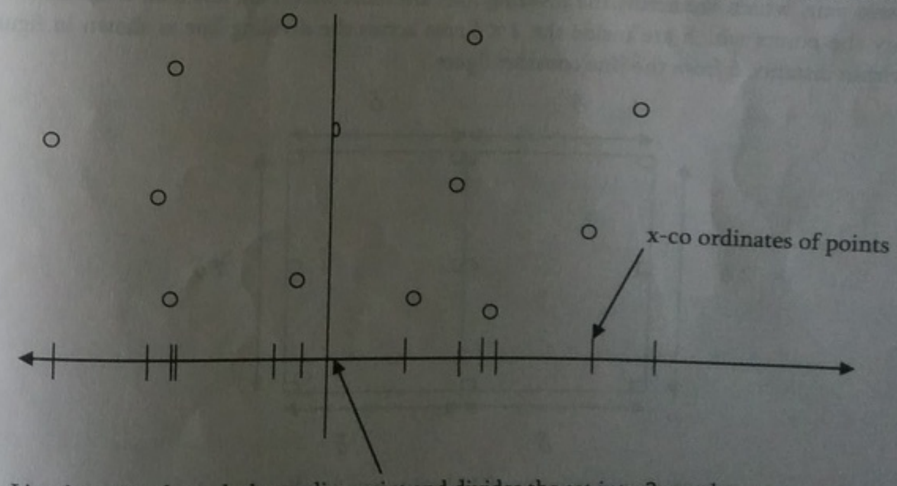
Brute Force Solution:

- Calculate the distances between all the pairs of points. From n points there are n_{c_2} ways of selecting 2 points. ($n_{c_2} = O(n^2)$).
- After finding distances for all n^2 possibilities, we select the one which is giving the minimum distance and this takes O(n^2).

The overall time complexity is O(n^2).

Problem-28 Give O($nlogn$) solution for *closest pair* problem (Problem-27)?

Solution: To find O($nlogn$) solution, we should think about using D & C technique. Before starting the divide-and-conquer process let us assume that the points are sorted by increasing x-coordinate. Divide the points into two equal halves based on median of x- coordinates. That means problem is divided into that of finding the closest pair in each of the two halves. For simplicity let us consider the following algorithm and then we try to understand the process.



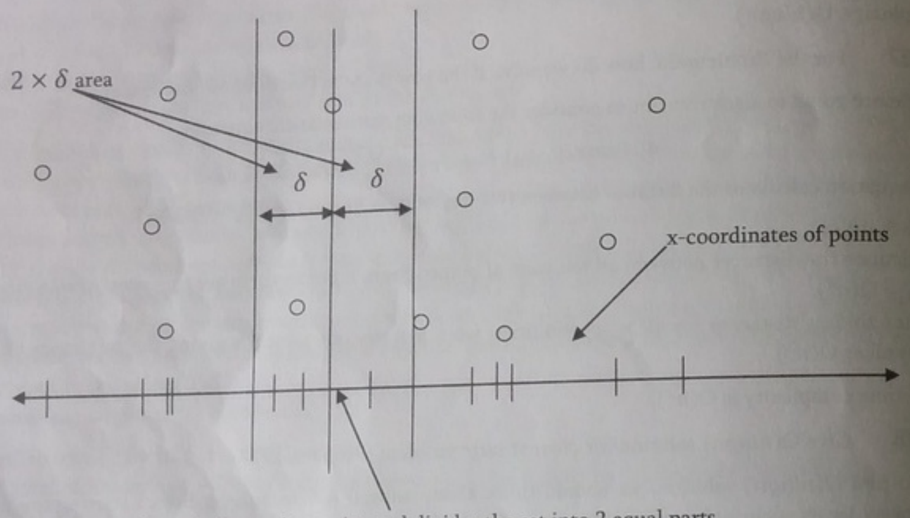
Line l passing through the median point and divides the set into 2 equal parts

Algorithm:

1) Sort the given points in S (given set of points) based on their x –coordinates. Partition S into two subsets, S_1 and S_2, about the line l through median of S. This step we can treat as *Divide* part of the D & C technique.

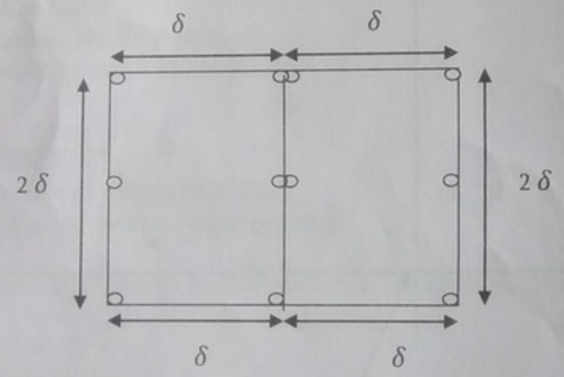
2) Find the closest-pairs in S_1 and S_2 and call them L and R recursively.
3) Now, steps-4 to 8 forms the Combining component of the D & C technique.
4) Let us assume that $\delta = min\,(L, R)$.
5) Eliminate points that are farther than δ apart from l.
6) Consider the remaining points and sort based on their y-coordinates.
7) Scan the remaining points in the y order and compute the distances of each point to all of its neighbors that are distanced no more than $2 \times \delta$ (that's the reason for sorting according to y).
8) If any of these distances is less than δ then update δ.

Combining the results in linear time



Line l passing through the median point and divides the set into 2 equal parts

Let $\delta = min(L, R)$, where L is the solution to first subproblem R is the solution to second subproblem. The possible candidates for closest-pair, which are across the dividing line, are those which are less than δ distance from the line. So we need to only the points which are inside the $2 \times \delta$ area across the dividing line as shown in figure. Now, to check all points within distance δ from the line consider figure.



From the above diagram we can seen that maximum of 12 points can be placed inside the square with a distance not less than δ. That means, we need to check only the distances which are within 11 positions in sorted list. This is similar to above one, but with the difference that in the above combining of subproblems, there are no vertical bounds. So we can apply the 12-point box tactic over all the possible boxes in the $2 \times \delta$ area with dividing line as

middle line. As there can be a maximum of n such boxes in the area, the total time for finding the closest pair in the corridor is $O(n)$.

Analysis:

1) Step-1 and Step-2 take $O(nlogn)$ for sorting and recursively finding the minimum.
2) Step-4 takes $O(1)$.
3) Step-5 takes $O(n)$ for scanning and eliminating.
4) Step-6 takes $O(nlogn)$ for sorting.
5) Step-7 takes $O(n)$ for scanning.

The total complexity, $T(n) = O(nlogn) + O(1) + O(n) + O(n) + O(n) \approx O(nlogn)$

Chapter-19

DYNAMIC PROGRAMMING

19.1 Introduction

In this chapter we will try to solve the problems for which we failed to get the optimal solutions using other techniques (say, divide & conquer and greedy methods). Dynamic Programming (DP) is a simple technique but it can be difficult to become a master on it. One easy way to identify and solve DP problems is by solving as many problems as possible. The term *Programming* is not related to coding but it is from literature which has the meaning of filling tables (similar to Linear Programming).

19.2 What is Dynamic Programming Strategy?

Dynamic programming and memoization work together. The main difference between dynamic programming and divide & conquer is that in-case of divide & conquer, subproblems are independent, where as in DP overlap of subproblems occur. By using memoization [maintaining a table of subproblems already solved], dynamic programming reduces the exponential complexity to polynomial complexity ($O(n^2)$, $O(n^3)$, etc..) for many problems. The major components of DP are:

- Recursion: Solves subproblems recursively.
- Memoization: Stores already computed values in table (*Memoization* means caching).

$$Dynamic\ Programming = Recursion + Memoization$$

19.3 Properties of Dynamic Programming Strategy?

The two dynamic programming properties which tells whether it can solve the given problem or not are:

- *Optimal substructure*: an optimal solution to a problem contains optimal solutions to subproblems.
- *Overlapping subproblems*: a recursive solution contains a small number of distinct subproblems repeated many times.

19.4 Can Dynamic Programming Solve Any Problem?

Like greedy and divide and conquer techniques, DP cannot solve every problem. There are problems which cannot be solved by any algorithmic technique [greedy, divide & conquer and dynamic programming].

The difference between the dynamic programming and straightforward recursion is in memoization of recursive calls. If the subproblems are independent and there is no repetition then memoization does not help, so dynamic programming is not a solution for all problems.

19.5 Dynamic Programming Approaches

Basically there are two approaches for solving DP problems:

- Bottom-up dynamic programming
- Top-down dynamic programming

Bottom-up Dynamic Programming

In this method, we evaluate the function starting with smallest possible input argument value and then we step through possible values, slowly increase input argument value. While computing the values we store all computed values in a table (memory). As larger arguments evaluated, precomputed values for smaller arguments can be used.

Top-down Dynamic Programming

In this method, the problem is broken into subproblems, and these subproblems are solved and the solutions remembered, in case they need to be solved. Also, we save the each computed value as final action of recursive function and as the first action we check if pre-computed value exists.

Bottom-up versus Top-down Programming

In bottom-up programming, programmer has to do the thinking by selecting values to calculate and order of calculation. In this case, all subproblems that might be needed are solved in advance and then used to build up solutions to larger problems. In top-down programming, recursive structure of original code is preserved, but unnecessary recalculation is avoided. The problem is broken into subproblems, and these subproblems are solved and the solutions remembered, in case they need to be solved again.

Note: Some problems can be solved with both the techniques and we will see such examples in next section.

19.6 Examples of Dynamic Programming Algorithms

- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.
- Algorithms on graphs can be solved efficiently: Bellman-Ford algorithm for finding the shortest distance in a graph, Floyd's All-Pairs shortest path algorithm etc...
- Chain matrix multiplication
- Subset Sum
- 0/1 Knapsack
- Travelling salesman problem and many more

19.7 Understanding Dynamic Programming

Before jumping to problems, let us try to understand how DP works through examples.

Fibonacci Series

In Fibonacci series, the current number is the sum of previous two numbers. The Fibonacci series is defined as follows:

$$\begin{aligned} Fib(n) &= 0, && for\ n = 0 \\ &= 1, && for\ n = 1 \\ &= Fib(n-1) + Fib(n-2), && for\ n > 1 \end{aligned}$$

The recursive implementation can be given as:

```
int RecursiveFibonacci(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    return RecursiveFibonacci(n -1) + RecursiveFibonacci(n -2);
}
```

Solving the above recurrence gives,

$$T(n) = T(n-1) + T(n-2) + 1 \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 2^n = O(2^n)$$

Note: For proof, refer *Introduction* chapter.

How Memoization Helps?

Calling $fib(5)$, produces a call tree that calls the function on the same value many times:

$fib(5)$
$fib(4) + fib(3)$

$(fib(3) + fib(2)) + (fib(2) + fib(1))$
$((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))$
$(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))$

In the above example, $fib(2)$ was calculated three times (over-lapping of subproblems). If the n is big then many more values of fib (subproblems) are recalculated which leads to an exponential time algorithm. Instead of solving the same subproblems again and again if we can store the previous calculated values then it reduces the complexity.

Memorization works like this: start with a recursive function and add a table that maps the functions parameter values to the results computed by the function. Then if this function is called twice with the same parameters, we simply look up the answer in the table.

Improving: Now, we see how DP reduces this problem complexity from exponential to polynomial. As discussed earlier, there are two ways of doing this. One approach is bottom-up: these methods starts with lower values of input and keep building the solutions for higher values.

```
int fib[n];
int fib(int n) {
        // Check for base cases
        if(n == 0 || n == 1) return 1;
        fib[0] = 1;
        fib[1] = 1;
        for (int i = 2; i < n; i++)
                fib[i] = fib[i - 1] + fib[i - 2];
        return fib[n - 1];
}
```

Other approach is top-down. In this method, we preserve the recursive calls and use the values if they are already computed. The implementation for this is given as:

```
int fib[n];
int fibonacci( int n ) {
        if(n == 1) return 1;
        if(n == 2) return 1;
        if( fib[n] != 0) return fib[n] ;
        return fib[n] = fibonacci(n-1) + fibonacci(n -2) ;
}
```

Note: For all problems, it may not be possible to find both top-down and bottom-up programming solutions.

Both the versions of Fibonacci series implementations clearly reduce the problem complexity to $O(n)$. This is because if a value is already computed then we are not calling the subproblems again. Instead, we are directly taking its value from table.

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for table.

Further Improving: One more observation from the Fibonacci series is: the current value is just the sum of previous two calculations only. This gives us the indication that we don't have to store all the previous values. Instead if we store last two values, we can calculate the current value. Implementation for this is given below:

```
int fibonacci(int n) {
        int a = 0, b = 1, sum, i;
        for (i=0;i < n;i++) {
                sum = a + b;
                a = b;
                b = sum;
        }
        return sum;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Note: This method may not be applicable (available) for all problems.

Observations

While solving the problems using DP, try to figure out the following:

- See how problems are defined in terms of subproblems recursively.
- See if we can use some table [memoization] to avoid the repeated calculations.

Factorial of a Number

As another example consider the factorial problem: $n!$ is the product of all integers between n and 1. Definition of recursive factorial can be given as:

$$n! = n * (n-1)!$$
$$1! = 1$$
$$0! = 1$$

This definition can easily be converted to implementation. Here the problem is finding the value of $n!$, and sub problem is finding the value of $(n-1)!$. In the recursive case, when n is greater than 1, function calls itself to find the value of $(n-1)!$ and multiplies that with n. In the base case, when n is 0 or 1, the function simply returns 1.

```
int fact(int n) {
        if(n == 1) return 1;
        else if(n == 0) return 1;
        else    // recursive case: multiply n by (n -1) factorial
                return n *fact(n -1);
}
```

The recurrence for the above implementation can be given as: $T(n) = n \times T(n-1) \approx O(n)$
Time Complexity: $O(n)$. Space Complexity: $O(n)$, recursive calls need a stack of size n.

In the above recurrence relation and implementation, for any n value, there are no repetitive calculations (*no* over-lapping of subproblems) and the factorial function is not getting any benefits with dynamic programming. Now, let us say we want to compute a series of $m!$ for some arbitrary value m. Using the above algorithm, for each such call we can compute it in $O(m)$. For example to find both $n!$ and $m!$, then using the above approach, the total complexity for finding $n!$ and $m!$ is $O(m+n)$.

Time Complexity: $O(n+m)$. Space Complexity: $O(\max(m,n))$, recursive calls need a stack of size equal to the maximum of m and n.

Improving: Now let us see how DP improves the complexity. From the above recursive definition it can be seen that $fact(n)$ is calculated from $fact(n-1)$ and n and nothing else. Instead of calling $fact(n)$ every time we can store the previous calculated values in some table and use them to calculate new value. This implementation can be given as:

```
int facto[n];
int fact(int n) {
        if(n == 1) return 1;
        else if(n == 0) return 1;
        //Already calculated case
        else if(facto[n]!=0) return facto[n];
        else    // recursive case: multiply n by (n -1) factorial
                return facto[n]= n *fact(n -1);
}
```

For simplicity, let us assume that we have already calculated $n!$ and want to find $m!$. For finding $m!$, we just need to see the table and use the existing entries if they are already computed. If $m < n$ then we do not have to recalculate the $m!$. If $m > n$ then we can use $n!$ and call the factorial on remaining numbers only.

The above implementation clearly reduces the complexity to $O(max(m,n))$. This is because if the $fact(n)$ is already there then we are not recalculating the value again. If we fill these newly computed values then the subsequent calls further reduces the complexity.

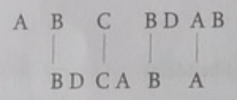
Time Complexity: $O(max(m,n))$. Space Complexity: $O(max(m,n))$ for table.

Longest Common Subsequence

Given two strings: string X of length m $[X(1..m)]$, and string Y of length n $[Y(1..n)]$, find longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings. For example, if X = "ABCBDAB" and Y = "BDCABA", the $LCS(X, Y)$ = {"BCBA", "BDAB", "BCAB"}. As we can see there are several optimal solutions.

Brute Force Approach: One simple idea is to check every subequence of $X[1..m]$ (m is the length of sequence X) to see if it is also a subsequence of $Y[1..n]$ (n is the length of sequence Y). Checking takes $O(n)$ time, and there are 2^m subsequences of X. The running time thus is exponential $O(n.2^m)$ and is not good for large sequences.

Recursive Solution: Before going to DP solution, let us form the recusive solution for this and later we can add memoization to reduce the complexity. Let's start with some simple observations about the LCS problem. If we have two strings, say "ABCBDAB" and "BDCABA", if we draw lines from the letters in the first string to the corresponding letters in the second, no two lines cross:



From the above observation, we can see that current characters of X and Y may or may not match. That means, suppose that the two first characters differ. Then it is not possible for both of them to be part of a common subsequence - one or the other (or maybe both) will have to be removed. Finally, observe that once we have decided what to do with the first characters of the strings, the remaining subproblem is again a LCS problem, on two shorter strings. Therefore we can solve it recursively.

Solution to LCS should find two sequences in X and Y and let us say the starting index of sequence in X is i and starting index of sequence in T is j. Also, assume that $X[i \ldots m]$ is a substring of X starting at character i and going until the end of X and $Y[j \ldots n]$ is a substring of Y starting at character j and going until the end of Y.

Based on the above discussion, here we get the possibilities as described below:

1) If $X[i] == Y[j] : 1 + LCS(i+1, j+1)$
2) If $X[i] \neq Y[j]$: $LCS(i, j+1)$ // skipping j^{th} character of Y
3) If $X[i] \neq Y[j]$: $LCS(i+1, j)$ // skipping i^{th} character of X

In the first case, if $X[i]$ is equal to $Y[j]$, we get a matching pair and can count it towards the total length of the LCS. Otherwise, we need to skip either i^{th} character of X or j^{th} character of Y and find the longest common subsequence. Now, $LCS(i, j)$ can be defined as:

$$LCS(i,j) = \begin{cases} 0, & if\ i = m\ or\ j = n \\ Max\{LCS(i, j+1), LCS(i+1, j)\}, & if\ X[i] \neq Y[j] \\ 1 + LCS[i+1, j+1], & if\ X[i] == Y[j] \end{cases}$$

LCS has many applications. In web searching, if we find the smallest number of changes that are needed to change one word into another. A *change* here is an insertion, deletion or replacement of a single character.

```
//Initial Call: LCSLength(X, 0, m-1, Y, 0, n-1);
int LCSLength( int S], int i, int m, int T[], int j, int n ) {
        if (i == m || j == n)
```



```
                return 0;
        else if (X[i] == Y[j]) return 1 + LCSLength(X, i+1, m Y, j+1, n);
        else return max( LCSLength(X, i+1, m Y, j, n), LCSLength(X, i, m Y, j+1, n));
}
```

This is a correct solution but it's very time consuming. For example, if the two strings have no matching characters, so the last line always gets executed which give (if $m == n$) are close to $O(2^n)$.

DP Solution: Adding Memoization: The problem with the recursive solution is that the same subproblems get called many different times. A subproblem consists of a call to LCSLength, with the arguments being two suffixes of X and Y, so there are exactly $(i+1)(j+1)$ possible subproblems (a relatively small number). If there are nearly 2^n recursive calls, some of these subproblems must be being solved over and over.

The DP solution is to check whenever we want to solve a subproblem, whether we've already done it before. If so we look up the solution instead of solving it again. Implemented in the most direct way, we just add some code to our recursive solution to do this look up and the code for this can be given as:

```
int LCS[1024][1024];
int LCSLength( int X], int m, int Y[], int n ) {
        for( int i = 0; i <= m; i++ )
                LCS[i][n] = 0;
        for( int j = 0; j <= n; j++ )
                LCS[m][j] = 0;
        for( int i = m - 1; i >= 0; i) {
                for( int j = n - 1; j >= 0; j) {
                        LCS[i][j] = LCS[i + 1][j + 1]; // matching X[i] to Y[j]

                        if( X[i] == Y[j] )
                                LCS[i][j]++; // we get a matching pair

                        // the other two cases – inserting a gap
                        if(LCS[i][j + 1] > LCS[i][j] )
                                LCS[i][j] = LCS[i][j + 1];
                        if(LCS[i + 1][j] > LCS[i][j] )
                                LCS[i][j] = LCS[i + 1][j];
                }
        }
        return LCS[0][0];
}
```

First, take care of the base cases. We have created LCS table with one row and one column larger than the lengths of the two strings. Then run the iterative DP loops to fill each cell in the table. This is like doing recursion backwards, or bottom up.