CHAPTER

16

ALGORITHMS DESIGN TECHNIQUES

16.1 Introduction

In the previous chapters, we have seen many algorithms for solving different kinds of problems. Before solving a new problem, the general tendency is to look for the similarity of the current problem to 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 we will focus on a few of them (Greedy, Divide and Conquer, Dynamic Programming).

16.2 Classification

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

- Implementation Method
- 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 and queues to solve the problems.

Some problems are suited for recursive and others are suited for iterative. For example, the *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 the exact steps to get the result. For example, SQL is more declarative than procedural, because the queries don't specify the steps to produce the result. Examples of procedural languages include: C, PHP, and PERL.

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 take 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.

If the parallel algorithms are distributed on to different machines then we call such algorithms *distributed* algorithms.

Deterministic or Non-Deterministic

Deterministic algorithms solve the problem with a predefined process, whereas $non-deterministic$ algorithms guess 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 the optimal solution, we give approximation algorithms.

Approximation algorithms are generally associated with NP-hard problems (refer to the *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 the local best selection also makes for 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 the case of the latter there is no dependency among the sub problems, whereas in DP there will be an overlap of sub-problems. By using memoization [maintaining a table for already solved sub problems], 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 the memoization of recursive calls. When sub problems 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 sub problems already solved], dynamic programming reduces the complexity from exponential to polynomial.

Linear Programming

In linear programming, there are inequalities in terms of inputs and *maximizing* (or *minimizing*) 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 a 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, the 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 *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, 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 take exponential time, and some never halt. Note that some problems may have multiple algorithms with different complexities.

Randomized Algorithms

A 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 the Backtracking method refer to the *Recusion and Backtracking* chapter.

Note: In the next few chapters we discuss the Greedy, Divide and Conquer, and Dynamic Programming] design methods. These methods are emphasized because they are used more often than other methods to solve problems.

CHAPTER 17

GREEDY ALGORITHMS

17.1 Introduction

Let us start our discussion with simple theory that will give us an understanding of the Greedy technique. In the game of *Chess*, every time we make a decision about a move, we have to also think about the future consequences. Whereas, in the game of *Tennis* (or *Volleyball*), our action is based on the immediate situation. This means that in some cases making a decision that looks right at that moment gives the best solution (*Greedy*), but in other cases it doesn't. The Greedy technique is best suited for looking at the immediate situation.

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 that a local good selection makes for a 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 the 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 the future. It iteratively makes one Greedy choice after another and reduces the given problem to 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 Always Work?

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

17.5 Advantages and Disadvantages of Greedy Method

The main advantage of the Greedy method is that it is straightforward, easy to understand and easy to code. In Greedy algorithms, once we make a decision, we do not have to spend time re-examining the 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 an approximation algorithm for complex problems

17.7 Understanding Greedy Technique

For better understanding let us go through an example.

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 the sum of the lengths of all character codes should be minimum [the sum of each character's frequency multiplied by the number of bits in the representation].

The basic idea behind the Huffman coding algorithm is to use fewer bits for more frequently occurring characters. The 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, the system generally 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 instead use a 7 bit code for e and a 9 bit code for q 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 on 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.

An Example

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

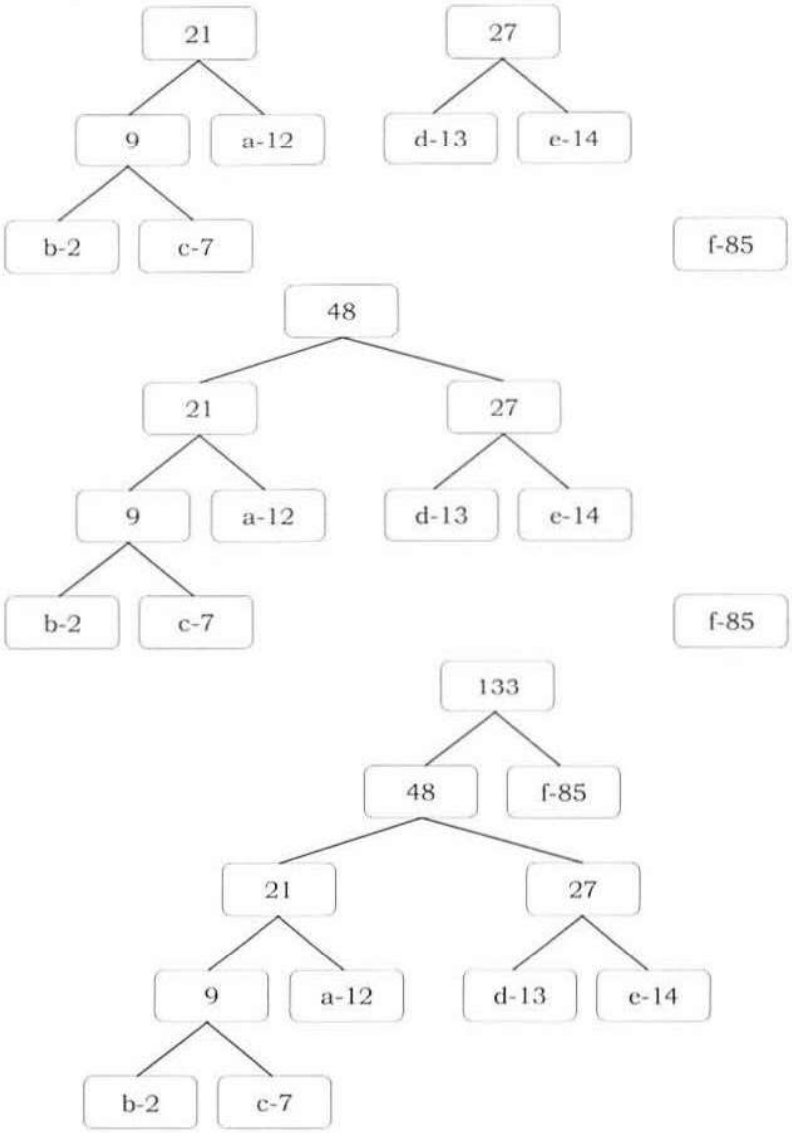
Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

Given 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: In the list, find the two binary trees 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 this:

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 to the 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 the number of bits that 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 by 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% of the storage space.

```
from heapq import heappush, heappop, heapify
from collections import defaultdict

def HuffmanEncode(characterFrequency):
    heap = [[freq, [sym, ""]] for sym, freq in characterFrequency.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

inputText = "this is an example for huffman encoding"
characterFrequency = defaultdict(int)
for character in inputText:
    characterFrequency[character] += 1

huffCodes = HuffmanEncode(characterFrequency)
print "Symbol\tFrequency\tHuffman Code"

for p in huffCodes:
    print "%s\t\t%s\t\t%s" % (p[0], characterFrequency[p[0]], p[1])
```

Time Complexity: $O(n \log n)$, 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 to the *Priority Queues* chapter for details.

17.8 Greedy Algorithms: Problems & Solutions

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 the previous merge and merge with the 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 a 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 list below, 15 indicates the cost of merging two files with sizes 10 and 5.

$$\{15,100,50,20,15\}$$

Similarly, merging 15 with the 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 give the 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 K-way merging.

Solution: This algorithm will not produce the optimal solution and consider the previous example for a counter example. As per the above algorithm, we need to merge the first pair of files (10 and 5 size files), the second pair of files (100 and 50) and the 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 [below, the third element does not have a pair element, so keep it the 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 In Problem-1, what is the best way to merge *all the files* into a single file?

Solution: Using the 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 the 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 the previous example. The given array is:

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

As per the above algorithm, after 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 list below, 15 indicates the cost of merging two files with sizes 10 and 5.

{15,15,20,50,100}

Similarly, merging the 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.

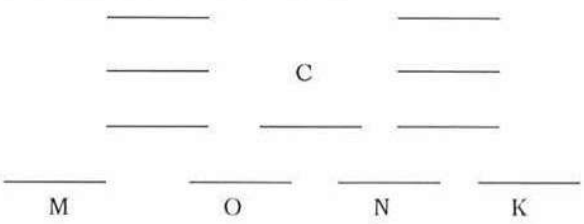
Time Complexity: O($nlogn$) time using heaps to find best merging pattern plus the optimal cost of merging the files.

Problem-4 **Interval Scheduling Algorithm:** Given a set of n intervals $S = \{(start_i, end_i) | 1 \leq i \leq n\}$. 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) {

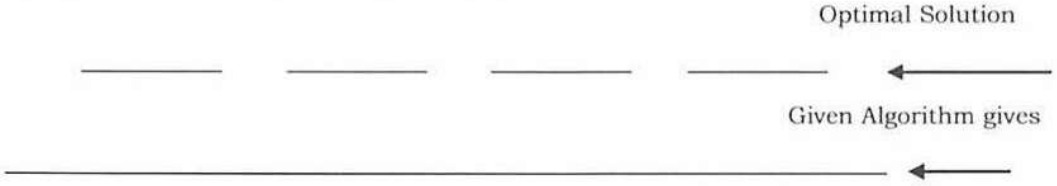
Select the interval I that overlaps the least number of other intervals.
Add I to final solution set S'.
Remove all intervals from S that overlap with I.
}

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.



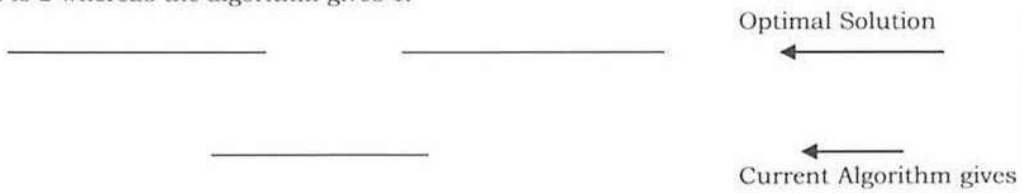
Problem-5 In Problem-4, if we select the interval that starts earliest (also not overlapping with already chosen intervals), does it give the optimal solution?

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



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

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



Problem-7 For 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.

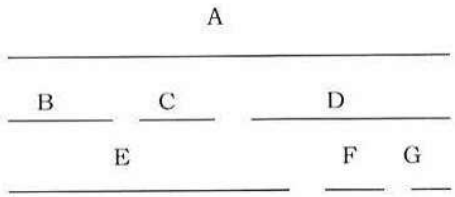
Input: $S = \{(start_i, end_i) | 1 \leq i \leq n\}$ of intervals. The interval $(start_i, end_i)$ we can treat as a request for a room for a class with time $start_i$ to time end_i.

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 the 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 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, so it must then be 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 to Problem-7 and for code refer to 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, and whenever we encounter a class that doesn't conflict with the latest class so far, then we take that class.

```
def LargestTasks(Start, n,  Finish):
    sort Finish[]
    rearrange Start[] to match
    count = 1
    X[count]  = 1
    for i in range(2,n):
            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.
This algorithm clearly runs in O($nlogn$) time due to sorting.

Problem-11 Consider the making change problem in the country of 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 the 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 algorithm 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 the optimal solution, refer to the *Dynamic Programming* chapter.

Problem-12 Let us assume that we are going for a 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 on our route. Assume that our car's tanks can hold petrol for n miles. Assume that the value n is given. Suppose we stop at every point. Does it give the 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 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 from A with a full tank. We check our map to determine the farthest petrol station on our route within n miles. We stop at that petrol station, fill up our tank and check our map again to determine the farthest petrol station on our route within n miles from this stop. Repeat the process until we get to B.

Note: For code, refer to *Dynamic Programming* 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 its 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 railway station, we have a time-table with the trains' arrivals and departures. We need to find the minimum number of platforms so that all the trains can be accommodated as per their schedule.

Example: The timetable is as given below, the answer is 3. Otherwise, the railway 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 done by determining the maximum number of trains at the railway station at any time.

First, sort all the arrival(A) and departure(D) times in an array. Then, save the corresponding arrivals anddepartures 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 this:

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

Finally make a cumulative array out of this:

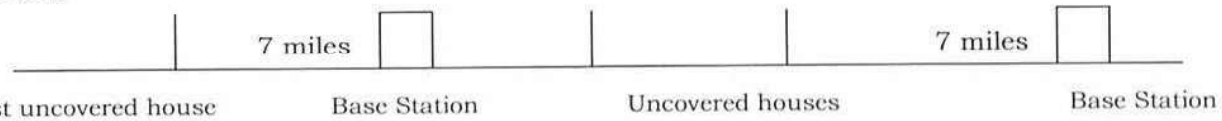
1	2	1	2	3	2	1	0

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 the same time, then put the departure time first in the 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 phones. We want to place cell phone towers along the road, and each cell phone tower covers a range of 7 kilometers. Create an efficient algorithm that allow for the fewest 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 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 these on a tape. In the future, users will want to read those songs from the tape. Reading a song from a tape is not like reading from a 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, with the result that 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. We should store the songs in the order from shortest to longest. Storing the short songs at the beginning reduces the forwarding times for the 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 $\mathbf{P[i]}$ rupees ($\mathbf{P[i]} > 0$) 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 be 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 $\mathbf{g_i}$ used as the key.
- A *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 of which takes O($logn$) time.

Problem-19 Let us consider a customer-care server (say, mobile customer-care) with n customers to be served 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 time then we can reduce the total waiting time.

Time Complexity: O($nlogn$).

CHAPTER

18

Divide and Conquer Algorithms

18.1 Introduction

In the *Greedy* chapter, we have seen that for many problems the Greedy strategy failed to provide optimal solutions. Among those problems, there are some that can be easily solved by using the *Divide and Conquer* (D & *C*) technique. Divide and Conquer is an important algorithm design technique based on recursion.

The *D* & *C* algorithm works by recursively breaking down a problem into two or more sub problems of the same type, until they become simple enough to be solved directly. The solutions to the sub problems 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 Always Work?

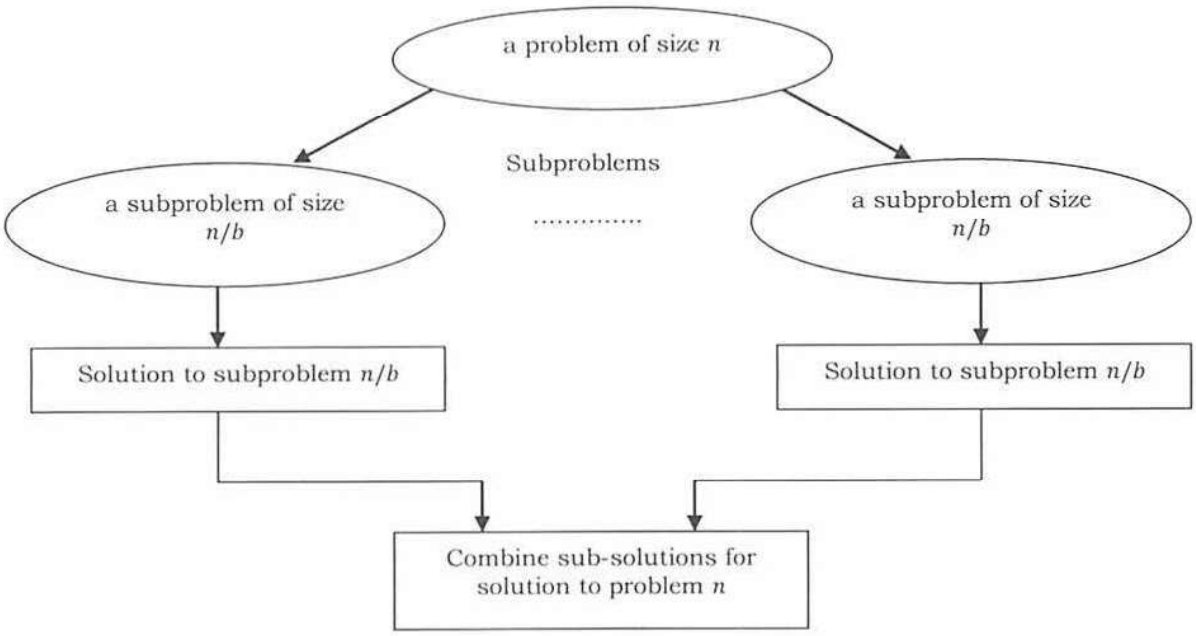
It's not possible to solve all the problems with the Divide & Conquer technique. As per the definition of D & C, the recursion solves the subproblems which are of the same type. For all problems it is not possible to find the subproblems which are the same size and *D* & *C* is not a choice for all problems.

18.4 Divide and Conquer Visualization

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

```
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 a clear understanding of D & C, let us consider a story. There was an old man who was 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 to those topics to get an idea of how D & C works. Below are a 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 the original problem.

- Looking for a name in a phone book: We have a phone book with names in alphabetical order. Given a name, how do we find whether that name is there in the phone book or not?
- Breaking a stone into dust: We want to convert a stone into dust (very small stones).
- Finding the exit in a hotel: We are at the end of a very long hotel lobby with a long series of doors, with one door next to us. We are looking for the door that leads to the exit.
- 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 the Tower of Hanoi problem. This requires breaking the problem into subproblems, solving the trivial cases and combining the subproblems to solve the original problem. Dividing the problem into subproblems so that subproblems can be combined again is a major difficulty in designing a new algorithm. For many such problems D & C provides a simple solution.

Parallelism: Since D & C allows us to solve the subproblems independently, this allows for 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 the $D\ \&\ C$ approach is that recursion is slow. This is because of the overhead of the repeated subproblem calls. Also, the $D\ \&\ C$ approach needs 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 $D\ \&\ C$ approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums.

18.8 Master Theorem

As stated above, in the $D\ \&\ C$ 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 to the *Introduction to Analysis of Algorithms* chapter. Just for continuity, let us reconsider 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 $T(n) = \Theta(n^{log_b^a})$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$
   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$
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 and Quick Sort
- Median Finding
- Min and Max Finding
- Matrix Multiplication
- Closest Pair problem

18.10 Divide and Conquer: Problems & Solutions

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, the algorithm divides the problem into 5 sub problems with each of size $\frac{n}{2}$. So we need to solve $5T(\frac{n}{2})$ subproblems. After solving these sub problems, the given array (linear time) is scanned 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 the 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 the input size is n and $T(n)$ defines the solution to the given problem. As per the description of algorithm we divide the problem into 2 sub problems with each of size $n - 1$. So we have to solve $2T(n - 1)$ sub problems. After solving these sub problems, the algorithm takes 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 $O\left(n^0 2^{\frac{n}{i}}\right) = O(2^n)$. (Refer to *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 divide the problem into 9 sub problems with each of size $\frac{n}{3}$. So we need to solve $9T(\frac{n}{3})$ sub problems. After solving the sub problems, the algorithm takes 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.

```
def function(n):
    if(n > 1):
        print("*")
        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 into 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. The elements are in sorted order. In binary search we take the middle element and check whether the element to be searched 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 right sub-array for finding the element and discard the left sub-array. Similarly, if the element to be searched is less than the middle element then we consider the left sub-array for finding the element and discard the right 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 halves. As per the above discussion every time we divide the problem into 2 sub problems with each of size $\frac{n}{2}$ and solve one $T(\frac{n}{2})$ sub problem. 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 2 equal parts. Write the recurrence for this ternary search and find its complexity.

Solution: From the discussion on Problem-5, binary search has the recurrence relation: $T(n) = T\left(\frac{n}{2}\right) + O(1)$. Similar to the Problem-5 discussion, instead of 2 in the recurrence relation we use "3". That indicates that we are dividing the array into 3 sub-arrays with equal size 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 In 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 to *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 it gives O(nlogn) complexity.

Problem-10 Discuss Quick Sort and its complexity.

Solution: Refer to *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 number of elements is greater than 1 then they are divided into two equal subsets, and 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 are already in sorted order. But, we need to scan the complete elements to partition the elements. The recurrence equation of *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 sub problems the algorithm takes 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 to *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 the list. Our first problem is to find the end of the list. To do that, we can start at the first element and keep searching with doubled index. That means we first search at index 1 then, 2, 4, 8 ...

```
def findInInfiniteSeries(A):
    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 have 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 implement 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: We use the Merge Sort process. Use *merge* procedure of merge sort (refer to *Sorting* chapter). Keep track of the count while comparing elements of two arrays. If the 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 the algorithm if the size of the two lists are not the same?

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

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

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

Solution: Yes, using the 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 $m2$ are equal then return $m1$ (or $m2$).
3. If $m1$ is greater than $m2$, then the final median will be 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 two sub arrays below.
7. From $m1$ to last element of $L1$.
8. From first element of $L2$ to $m2$.
9. Repeat the above process until the size of both the sub arrays becomes 2.
10. If size of the two arrays is 2, then use the formula below 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 to *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 the 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 the stock price will not be the same on all the days. In the input stock prices there may be dates where the stock is high when we can sell the current holdings, and there may be days when 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.

Solution: As given in the 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 the buy date should be before sell date. One simple approach is to look at all possible buy and sell dates.

```
def calculateProfitWhenBuyingNow(A, index):
    buyingPrice = A[index]
    maxProfit = 0
    sellAt = index
    for i in range(index+1, len(A)):
        sellingPrice = A[i]
        profit = sellingPrice - buyingPrice
        if profit > maxProfit:
            maxProfit = profit
            sellAt = i
    return maxProfit, sellAt
```

```
# check all possible buying times
def StockStrategyBruteForce(A):
    maxProfit = None
    buy = None
    sell = None
    for index, item in enumerate(A):
        profit, sellAt = calculateProfitWhenBuyingNow(A, index)
        if (maxProfit is None) or (profit > maxProfit):
            maxProfit = profit
            buy = index
            sell = sellAt
    return maxProfit, buy, sell
```

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

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

Solution: Yes, by opting for the Divide-and-Conquer $\Theta(nlogn)$ solution. Divide the input list into two parts and recursively find the solution in both the parts. Here, 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.

```
def StockStrategy(A, start, stop):
    n = stop - start
    # edge case 1: start == stop: buy and sell immediately = no profit at all
    if n == 0:
        return 0, start, start
    if n == 1:
        return A[stop] - A[start], start, stop
    mid = start + n/2
    # the "divide" part in Divide & Conquer: try both halfs of the array
    maxProfit1, buy1, sell1 = StockStrategy(A, start, mid-1)
    maxProfit2, buy2, sell2 = StockStrategy(A, mid, stop)
    maxProfitBuyIndex = start
    maxProfitBuyValue = A[start]
    for k in range(start+1, mid):
        if A[k] < maxProfitBuyValue:
            maxProfitBuyValue = A[k]
            maxProfitBuyIndex = k
    maxProfitSellIndex = mid
    maxProfitSellValue = A[mid]
    for k in range(mid+1, stop+1):
        if A[k] > maxProfitSellValue:
            maxProfitSellValue = A[k]
            maxProfitSellIndex = k
    # those two points generate the maximum cross border profit
    maxProfitCrossBorder = maxProfitSellValue - maxProfitBuyValue
    # and now compare our three options and find the best one
    if maxProfit2 > maxProfit1:
        if maxProfitCrossBorder > maxProfit2:
            return maxProfitCrossBorder, maxProfitBuyIndex, maxProfitSellIndex
        else:
            return maxProfit2, buy2, sell2
    else:
        if maxProfitCrossBorder > maxProfit1:
            return maxProfitCrossBorder, maxProfitBuyIndex, maxProfitSellIndex
        else:
```

```
        return maxProfit1, buy1, sell1
def StockStrategyWithDivideAndConquer(A):
    return StockStrategy(A, 0, len(A)-1)
```

Algorithm *StockStrategy* is used recursively on two problems of half the size of the input, and in addition $\Theta(n)$ time is spent 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 breaking it (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 it recursively. Let us take an example for understanding the scenario. Let us say 14 is the answer. That means we need 14 drops to find the answer. First we drop from height 14, and if it breaks we try all floors from 1 to 13. If it doesn't break then we are left 13 drops, so we will drop it from $14 + 13 + 1 = 28^{th}$ floor. The reason being if it breaks at the 28^{th} floor we can try all the floors from 15 to 27 in 12 drops (total of 14 drops). If it did not break, then we are left with 11 drops and we can try to 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 that the 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 to *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 to *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 a specific nature. For example you can calculate the new position of the element using the value of the element itself. This is a hashing technique.

```
def shuffleArray(A, l, r):
    #Array center
    c = l + (r-l)/2
    q = 1 + l + (c-l)//2
    if(l == r):                    # Base case when the array has only one element
            return
    k = 1
    I = q
    while(i<=c):
```

```
            # Swap elements around the center
            tmp = A[i]
            A[i] = A[c + k]
            A[c + k] = tmp
            i += 1
            k += 1
    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 to *Sorting* chapter.

Problem-25 **Maximum Value Contiguous Subsequence:** Given a sequence of n numbers $A(1)...A(n)$, give an algorithm for finding a contiguous subsequence $A(i)...A(j)$ 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 with 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 the 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.

```
def maxSumWithDivideAndConquer(A, low, hi):
    #run MCS algorithm on condensed list
    if low is hi:
            return (low, low, A[low][2])
    else:
            pivot = (low + hi) / 2
            #max subsequence exclusively in left half
            left = maxSumWithDivideAndConquer(A, low, pivot)
            #max subsequence exclusively in right half
            right = maxSub(A, pivot + 1, hi)
            #calculate max sequence left from mid
            leftSum = A[pivot][2]
            temp = 0
            for i in xrange(pivot, low - 1, -1):
                    temp += A[i][2]
                    if temp >= leftSum:
                            l = i
                            leftSum = temp
            #calculate max sequence right from mid
            rightSum = A[pivot + 1][2]
            temp = 0
            for i in xrange(pivot + 1, hi + 1):
                    temp += A[i][2]
                    if temp >= rightSum:
                            r = i
                            rightSum = temp
            #combine to find max subsequence crossing mid
```

```
        mid = (l, r, leftSum + rightSum)
        if left[2] > mid[2] and left[2] > right[2]:
                return left
        elif right[2] > mid[2] and right[2] > left[2]:
                return right
        else:
                return mid
list = [100, -4, -3, -10, -5, -1, -2, -2, -0, -15, -3, -5, -2, 70]
print maxSumWithDivideAndConquer(list, 0, len(list) - 1)
```

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 an efficient solution refer to the *Dynamic Programming* chapter.

Problem-26 **Closest-Pair of Points:** Given a set of n points, $S = \{p_1, p_2, p_3, ..., p_n\}$, where $p_i = (x_i, y_i)$. Find the pair of points having the 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 a line after we sort them (either on X-axis or Y-axis). The complexity of sorting is $O(nlogn)$. After sorting we can go through them to find the consecutive points with the least difference. So the problem in one dimension is solved in $O(nlogn)$ time which is mainly dominated by sorting time.

Time Complexity: $O(nlogn)$.

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

Solution: Before going to the 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 $O(n^2)$.

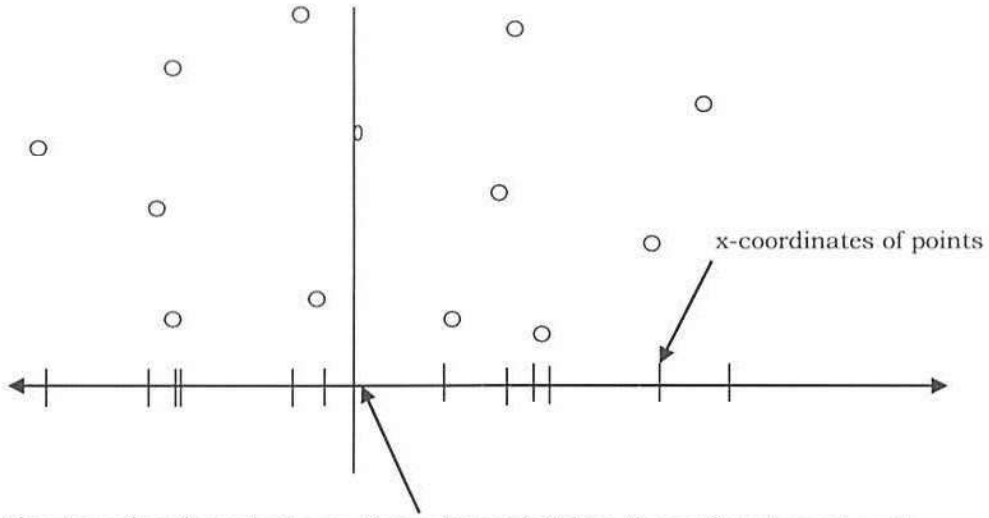
```
from math import sqrt, pow
def distance(a, b):
  return sqrt(pow(a[0] - b[0],2) + pow(a[1] - b[1],2))

def bruteMin(points, current=float("inf")):
  if len(points) < 2:
     return current
  else:
    head = points[0]
    del points[0]
    newMin = min([distance(head, x) for x in points])
    newCurrent = min([newMin, current])
    return bruteMin(points, newCurrent)

A = [(12,30), (40, 50), (5, 1), (12, 10), (3,4)]
print bruteMin(A)
```

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

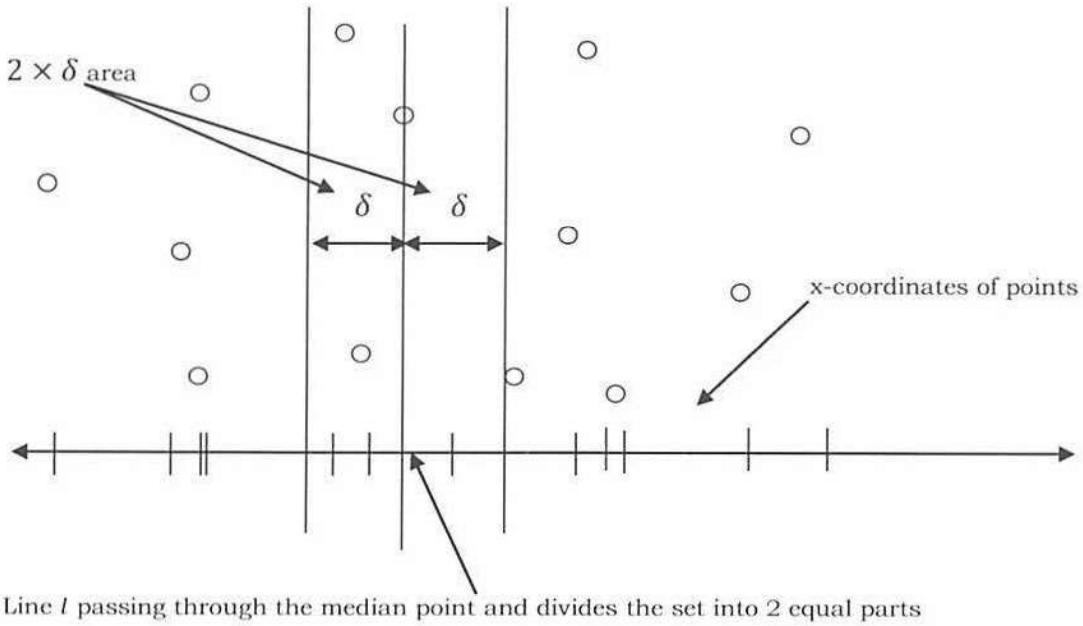
Solution: To find $O(nlogn)$ solution, we can use the D & C technique. Before starting the divide-and-conquer process let us assume that the points are sorted by increasing x-coordinate. Divide the points into two equal halves based on median of x-coordinates. That means the problem is divided into that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

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 is the *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 form 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 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.

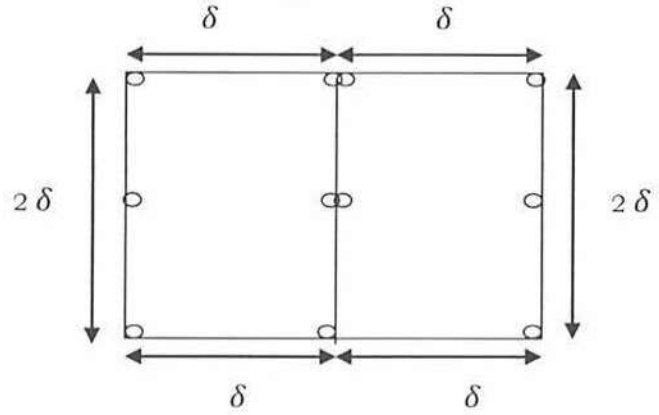


Combining the results in linear time



Let $\delta = min(L, R)$, where L is the solution to first sub problem and R is the solution to second sub problem. 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 only the points which are inside the $2 \times \delta$ area across the dividing line as shown in the figure. Now, to check all points within distance δ from the line, consider the following figure.



From the above diagram we can see that a 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 the sorted list. This is similar to the one above, 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 the dividing line as the 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: $T(n) = O(nlogn) + O(1) + O(n) + O(n) + O(n) \approx O(nlogn)$.

```
import operator

class Point():
    def __init__(self, x, y):
        """Init"""
        self.x = x
        self.y = y
    def __repr__(self):
        return '<{0}, {1}>'.format(self.x, self.y)

def distance(a, b):
    return abs((a.x - b.x) ** 2 + (a.y - b.y) ** 2) ** 0.5

def closestPoints(points):
    """Time complexity: O(nlogn)"""
    n = len(points)
    if n <= 1:
        print 'Invalid input'
        raise Exception
    elif n == 2:
        return (points[0], points[1])
    elif n == 3:
        # Calc directly
        (a, b, c) = points
        ret = (a, b) if distance(a, b) < distance(a, c) else (a, c)
        ret = (ret[0], ret[1]) if distance(ret[0], ret[1]) < distance(b, c) else (b, c)
        return ret
    else:
        points = sorted(points, key=operator.attrgetter('x'))
        leftPoints = points[ : n / 2]
        rightPoints = points[n / 2 : ]
        # Devide and conquer.
```

```
        (left_a, left_b) = closestPoints(leftPoints)
        (right_a, right_b) = closestPoints(rightPoints)
        # Find the min distance for leftPoints part and rightPoints part.
        d = min(distance(left_a, left_b), distance(right_a, right_b))
        # Cut the point set into two.
        mid = (points[n / 2].x + points[n / 2 + 1].x) / 2
        # Find all points fall in [mid - d, mid + d]
        midRange = filter(lambda pt : pt.x >= mid - d and pt.x <= mid + d, points)
        # Sort by y axis.
        midRange = sorted(midRange, key=operator.attrgetter('y'))
        ret = None
        localMin = None
        # Brutal force, for each point, find another point and delta y less than d.
        # Calc the distance and update the global var if hits the condition.
        for i in xrange(len(midRange)):
            a = midRange[i]
            for j in xrange(i + 1, len(midRange)):
                b = midRange[j]
                if (not ret) or (abs(a.y - b.y) <= d and distance(a, b) < localMin):
                        ret = (a, b)
                        localMin = distance(a, b)
        return ret
    points = [ Point(1, 2), Point(0, 0), Point(3, 6), Point(4, 7), Point(5, 5),
        Point(8, 4), Point(2, 9), Point(4, 5), Point(8, 1), Point(4, 3),
        Point(3, 3)]
    print closestPoints(points)
```

Problem-29 To calculate k^n, give algorithm and discuss its complexity.

Solution: The naive algorithm to compute k^n is: start with 1 and multiply by k until reaching k^n. For this approach; there are $n-1$ multiplications and each takes constant time giving a $\Theta(n)$ algorithm.

But there is a faster way to compute k^n. For example,

$$9^{24} = (9^{12})^2 = ((9^6)^2)^2 = (((9^3)^2)^2)^2 = (((9^2.9)^2)^2)^2$$

Note that taking the square of a number needs only one multiplication; this way, to compute 9^{24} we need only 5 multiplications instead of 23.

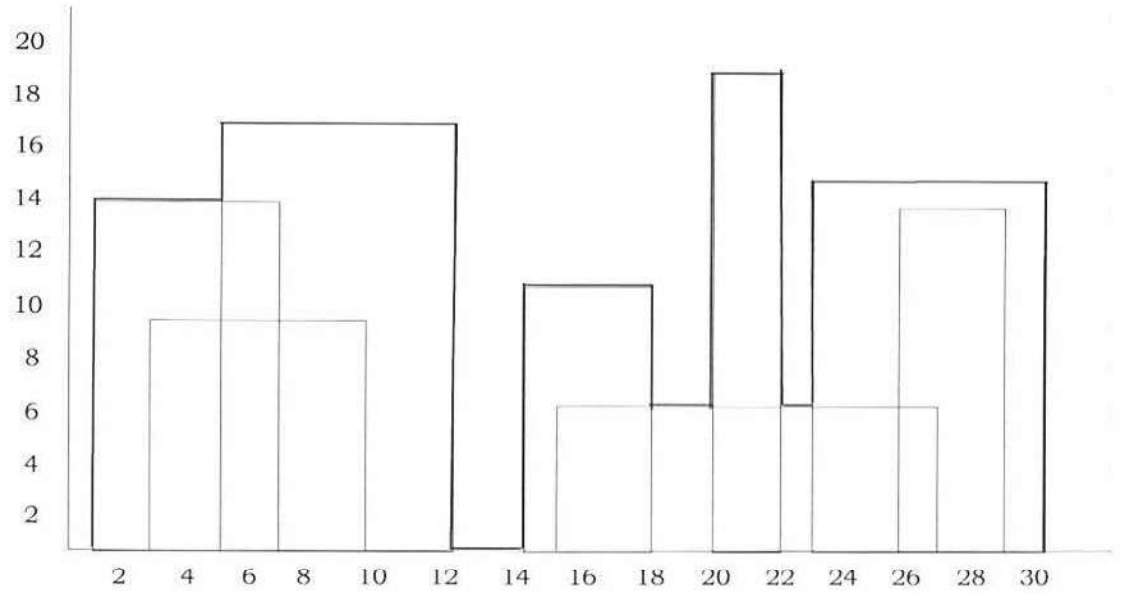
```
def powerBruteForce(k, n):
    """linear power algorithm"""
    x = k;
    for i in range(1, n):
        x *= k
    return x
def power(k, n):
    if n == 0: return 1
    x = power(k, math.floor(n/2))
    if n % 2 == 0: return pow(x, 2)
    else: return k * pow(x, 2)
```

Let T(n) be the number of multiplications required to compute k^n. For simplicity, assume $k = 2^i$ for some $i \geq 1$.

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

Using master theorem we get T(n) = O($logn$).

Problem-30 **The Skyline Problem**: Given the exact locations and shapes of n rectangular buildings in a 2-dimensional city. There is no particular order for these rectangular buildings. Assume that the bottom of all buildings lie on a fixed horizontal line (bottom edges are collinear). The input is a list of triples; one per building. A building B_i is represented by the triple (l_i, h_i, r_i) where l_i denote the x-position of the left edge and r_i denote the x-position of the right edge, and h_i denotes the building's height. Give an algorithm that computes the skyline (in 2 dimensions) of these buildings, eliminating hidden lines. In the diagram below there are 8 buildings, represented from left to right by the triplets (1, 14, 7), (3, 9, 10), (5, 17, 12), (14, 11, 18), (15, 6, 27), (20, 19, 22), (23, 15, 30) and (26, 14, 29).



The output is a collection of points which describe the path of the skyline. In some versions of the problem this collection of points is represented by a sequence of numbers p_1, p_2, ..., p_n, such that the point p_i represents a horizontal line drawn at height p_i if i is even, and it represents a vertical line drawn at position p_i if i is odd. In our case the collection of points will be a sequence of p_1, p_2, ..., p_n pairs of (x_i, h_i) where $p_i(x_i, h_i)$ represents the h_i height of the skyline at position x_i. In the diagram above the skyline is drawn with a thick line around the buildings and it is represented by the sequence of position-height pairs (1, 14), (5, 17), (12, 0), (14, 11), (18, 6), (20, 19), (22, 6), (23, 15) and (30, 0). Also, assume that R_i of the right most building can be maximum of 1000. That means, the L_i co-ordinate of left building can be minimum of 1 and R_i of the right most building can be maximum of 1000.

Solution: The most important piece of information is that we know that the left and right coordinates of each and every building are non-negative integers less than 1000. Now why is this important? Because we can assign a height-value to every distinct x_i coordinate where i is between 0 and 9,999.

Algorithm:

- Allocate an array for 1000 elements and initialize all of the elements to 0. Let's call this array *auxHeights*.
- Iterate over all of the buildings and for every B_i building iterate on the range of $[l_i .. r_i)$ where l_i is the left, r_i is the right coordinate of the building B_i.
- For every x_j element of this range check if h_i>*auxHeights*[xj], that is if building B_i is taller than the current height-value at position x_j. If so, replace *auxHeights*[x_j] with h_i.

Once we checked all the buildings, the *auxHeights* array stores the heights of the tallest buildings at every position. There is one more thing to do: convert the *auxHeights* array to the expected output format, that is to a sequence of position-height pairs. It's also easy: just map each and every i index to an (i, *auxHeights*[i]) pair.

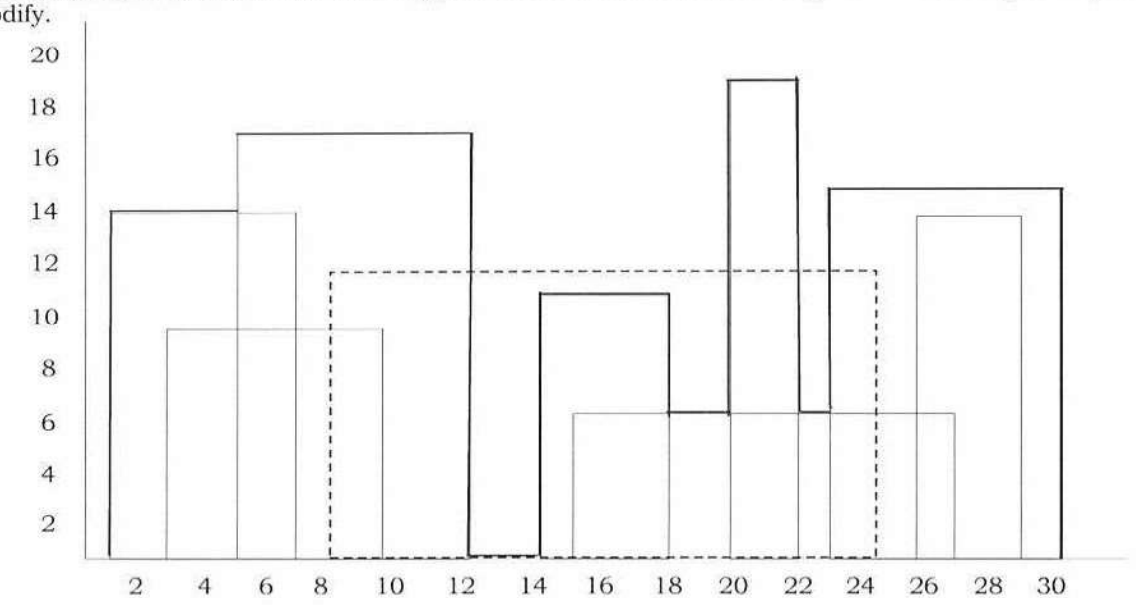
```
def SkyLineBruteForce():
    auxHeights = [0]*1000
    rightMostBuildingRi=0
    p = raw_input("Enter three values: ") # raw_input() function
    inputValues = p.split()
    inputCount = len(inputValues)
    while inputCount==3:
        left = int(inputValues[0])
        h = int(inputValues[1])
        right = int(inputValues[2])
        for i in range(left,right-1):
            if(auxHeights[i]<h):
                auxHeights[i]=h;
        if(rightMostBuildingRi<right):
                rightMostBuildingRi=right
        p = raw_input("Enter three values: ") # raw_input() function
        inputValues = p.split()
```

```
    inputCount = len(inputValues)
  prev = 0
  for i in range(1,rightMostBuildingRi-1):
    if prev!=auxHeights[i]:
      print i, " ", auxHeights[i]
    prev=auxHeights[i]
  print rightMostBuildingRi, " ", auxHeights[rightMostBuildingRi]
SkyLineBruteForce()
```

Let's have a look at the time complexity of this algorithm. Assume that, n indicates the number of buildings in the input sequence and m indicates the maximum coordinate (right most building r_i). From the above code, it is clear that for every new input building, we are traversing from *left* (l_i) to *right* (r_i) to update the heights. In the worst case, with n equal-size buildings, each having $l = 0$ left and $r = m - 1$ right coordinates, that is every building spans over the whole $[0..m)$ interval. Thus the running time of setting the height of every position is $O(n \times m)$. The overall time-complexity is $O(n \times m)$, which is a lot larger than $O(n^2)$ if $m > n$.

Problem-31 Can we improve the solution of the Problem-30?

Solution: It would be a huge speed-up if somehow we could determine the skyline by calculating the height for those coordinates only where it matters, wouldn't it? Intuition tells us that if we can insert a building into an *existing skyline* then instead of all the coordinates the building spans over we only need to check the height at the left and right coordinates of the building plus those coordinates of the skyline the building overlaps with and may modify.



Is merging two skylines substantially different from merging a building with a skyline? The answer is, of course, No. This suggests that we use divide-and-conquer. Divide the input of n buildings into two equal sets. Compute (recursively) the skyline for each set then merge the two skylines. Inserting the buildings one after the other is not the fastest way to solve this problem as we've seen it above. If, however, we first merge pairs of buildings into skylines, then we merge pairs of these skylines into bigger skylines (and not two sets of buildings), and then merge pairs of these bigger skylines into even bigger ones, then - since the problem size is halved in every step - after $logn$ steps we can compute the final skyline.

```
class SkyLinesDivideandConquer:
  # @param {integer[][]} buildings
  # @return {integer[][]}
  def getSkylines(self, buildings):
    result = []
    if len(buildings) == 0:
      return result
    if len(buildings) == 1:
      result.append([buildings[0][0], buildings[0][2]])
      result.append([buildings[0][1], 0])
      return result
    mid = (len(buildings) - 1) / 2
    leftSkyline = self.getSkyline(0, mid, buildings)
```

```
        rightSkyline = self.getSkyline(mid + 1, len(buildings)-1, buildings)
        result = self.mergeSkylines(leftSkyline, rightSkyline)
        return result
    def getSkyline(self, start, end, buildings):
        result = []
        if start == end:
            result.append([buildings[start][0], buildings[start][2]])
            result.append([buildings[start][1], 0])
            return result
        mid = (start + end) / 2
        leftSkyline = self.getSkyline(start, mid, buildings)
        rightSkyline = self.getSkyline(mid+1, end, buildings)
        result = self.mergeSkylines(leftSkyline, rightSkyline)
        return result
    def mergeSkylines(self, leftSkyline, rightSkyline):
        result = []
        i, j, h1, h2, maxH = 0, 0, 0, 0, 0
        while i < len(leftSkyline) and j < len(rightSkyline):
            if leftSkyline[i][0] < rightSkyline[j][0]:
                h1 = leftSkyline[i][1]
                if maxH != max(h1, h2):
                    result.append([leftSkyline[i][0], max(h1, h2)])
                maxH = max(h1, h2)
                i += 1
            elif leftSkyline[i][0] > rightSkyline[j][0]:
                h2 = rightSkyline[j][1]
                if maxH != max(h1, h2):
                    result.append([rightSkyline[j][0], max(h1, h2)])
                maxH = max(h1, h2)
                j += 1
            else:
                h1 = leftSkyline[i][1]
                h2 = rightSkyline[j][1]
                if maxH != max(h1, h2):
                    result.append([rightSkyline[j][0], max(h1, h2)])
                maxH = max(h1, h2)
                i += 1
                j += 1
        while i < len(leftSkyline):
            result.append(leftSkyline[i])
            i += 1
        while j < len(rightSkyline):
            result.append(rightSkyline[j])
            j += 1
        return result
```

For example, given two skylines A=(a_1, ha_1, a_2, ha_2, ..., a_n, 0) and B=(b_1, hb_1, b_2, hb_2, ..., b_m, 0), we merge these lists as the new list: (c_1, hc_1, c_2, hc_2, ..., c_{n+m}, 0). Clearly, we merge the list of a's and b's just like in the standard Merge algorithm. But, in addition to that, we have to decide on the correct height in between these boundary values. We use two variables *currentHeight*1 and *currentHeight*2 (note that these are the heights prior to encountering the heads of the lists) to store the current height of the first and the second skyline, respectively. When comparing the head entries (*currentHeight*1, *currentHeight*2) of the two skylines, we introduce a new strip (and append to the output skyline) whose x-coordinate is the minimum of the entries' x-coordinates and whose height is the maximum of *currentHeight*1 and *currentHeight*2. This algorithm has a structure similar to Mergesort. So the overall running time of the divide and conquer approach will be 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 master. 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, and means 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 and conquer is that in the case of the latter, sub problems are independent, whereas in DP there can be an overlap of sub problems. By using memoization [maintaining a table of sub problems 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 sub problems 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 can tell whether it can solve the given problem or not are:

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

19.4 Can Dynamic Programming Solve All Problems?

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 and Conquer and Dynamic Programming].

The difference between Dynamic Programming and straightforward recursion is in memoization of recursive calls. If the sub problems 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 the smallest possible input argument value and then we step through possible values, slowly increasing the input argument value. While computing the values we store all computed values in a table (memory). As larger arguments are evaluated, pre-computed values for smaller arguments can be used.

Top-down Dynamic Programming

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

Bottom-up versus Top-down Programming

In bottom-up programming, the programmer has to select values to calculate and decide the order of calculation. In this case, all sub problems that might be needed are solved in advance and then used to build up solutions to larger problems. In top-down programming, the recursive structure of the original code is preserved, but unnecessary recalculation is avoided. The problem is broken into sub problems, these sub problems 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 examples in the 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 going to problems, let us 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:

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

The recursive implementation can be given as:

```
def Fibo(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return Fibo(n-1)+Fibo(n-2)
print (Fibo(10))
```

Solving the above recurrence gives:

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

Note: For proof, refer to *Introduction* chapter.

How does Memoization help?

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 (overlapping of subproblems). If n is big, then many more values of fib (sub problems) are recalculated, which leads to an exponential time algorithm. Instead of solving the same sub problems again and again we can store the previous calculated values and reduce the complexity.

Memoization works like this: Start with a recursive function and add a table that maps the function's 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 start with lower values of input and keep building the solutions for higher values.

```
def Fibo(n):
  fibTable = [0,1]
  for i in range(2,n+1):
    fibTable.append(fibTable[i-1] + fibTable[i-2])
  return fibTable[n]
print(Fibo(10))
```

The 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:

```
fibTable = {1:1, 2:1}
def Fibo(n):
  if n <= 2:
    return 1
  if n in fibTable:
    return fibTable[n]
  else:
    fibTable[n] = Fibo(n-1) + Fibo(n-2)
    return fibTable[n]
print(Fibo(10))
```

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

Both versions of the 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 the 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 the sum of the previous two calculations only. This indicates that we don't have to store all the previous values. Instead, if we store just the last two values, we can calculate the current value. The implementation for this is given below:

```
def Fibo(n):
 a, b = 0, 1
 for i in range(n):
        a, b = b, a + b
 return a
print(Fibo(10))
```

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 the 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. The definition of recursive factorial can be given as:

$$\begin{aligned} n! &= n * (n-1)! \\ 1! &= 1 \\ 0! &= 1 \end{aligned}$$

This definition can easily be converted to implementation. Here the problem is finding the value of $n!$, and the sub-problem is finding the value of $(n-1)!$. In the recursive case, when n is greater than 1, the 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.

```
def factorial(n):
 if n == 0: return 1
 return n*factorial(n-1)
print(factorial(6))
```

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 overlapping of sub problems) 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 $m!$ we can use the above approach, wherein 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 reduces 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 a table and use these values to calculate a new value. This implementation can be given as:

```
factTable = {}
def factorial(n):
    try:
        return factTable[n]
    except KeyError:
        if n == 0:
            factTable[0] = 1
            return 1
        else:
            factTable[n] = n * factorial(n-1)
            return factTable[n]
print(factorial(10))
```

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 $m!$. If $m > n$ then we can use $n!$ and call the factorial on the 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 reduce the complexity.

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

19.8 Longest Common Subsequence

Given two strings: string X of length m $[X(1..m)]$, and string Y of length n $[Y(1..n)]$, find the 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"}. We can see there are several optimal solutions.

Brute Force Approach: One simple idea is to check every subsequence 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 recursive 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", and 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 the 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 sub problem is again a LCS problem, on two shorter strings. Therefore we can solve it recursively.

The 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 the starting index of sequence in Y 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 that $Y[j \ldots n]$ 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:

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

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.

```
def LCSLength(X, Y):
    if not X or not Y:
        return ""
    x, m, y, n = X[0], X[1:], Y[0], Y[1:]
    if x == y:
        return x + LCSLength(m, n)
    else:
        return max(LCSLength(X, n), LCSLength(m, Y), key=len)
print (LCSLength('thisisatest', 'testingLCS123testing'))
```

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

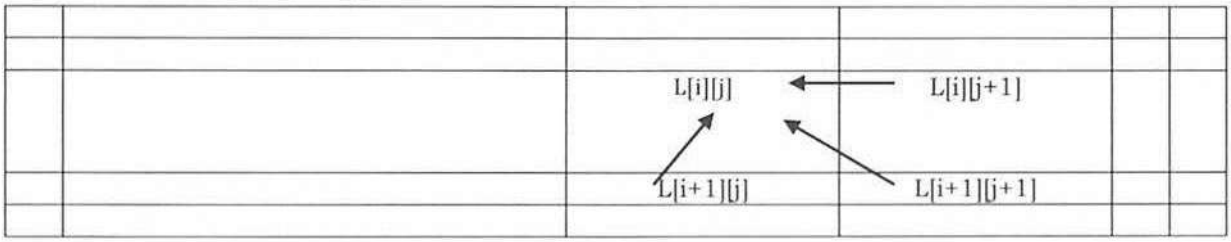
DP Solution: Adding Memoization: The problem with the recursive solution is that the same subproblems get called many different times. A subproblem consists of a call to LCSLength, with the arguments being two suffixes of X and Y, so there are exactly $(i+1)(j+1)$ possible subproblems (a relatively small 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 sub problem, whether we've already done it before. 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 the code. This can be given as:

```
def LCSLength(X, Y):
    Table = [[0 for j in range(len(Y)+1)] for i in range(len(X)+1)]
    # row 0 and column 0 are initialized to 0 already
    for i, x in enumerate(X):
```

```
        for j, y in enumerate(Y):
            if x == y:
                Table[i+1][j+1] = Table[i][j] + 1
            else:
                Table[i+1][j+1] = \
                    max(Table[i+1][j], Table[i][j+1])
    # read the substring out from the matrix
    result = ""
    x, y = len(X), len(Y)
    while x != 0 and y != 0:
        if Table[x][y] == Table[x-1][y]:
            x -= 1
        elif Table[x][y] == Table[x][y-1]:
            y -= 1
        else:
            assert X[x-1] == Y[y-1]
            result = X[x-1] + result
            x -= 1
            y -= 1
    return result
print (LCSLength('thisisatest', 'testingLCS123testing'))
```

First, take care of the base cases. We have created an *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.



The value of $LCS[i][j]$ depends on 3 other values ($LCS[i+1][j+1]$, $LCS[i][j+1]$ and $LCS[i+1][j]$), all of which have larger values of i or j. They go through the table in the order of decreasing i and j values. This will guarantee that when we need to fill in the value of $LCS[i][j]$, we already know the values of all the cells on which it depends.

Time Complexity: $O(mn)$, since i takes values from 1 to m and and j takes values from 1 to n.
Space Complexity: $O(mn)$.

Note: In the above discussion, we have assumed $LCS(i, j)$ is the length of the LCS with $X[i \ldots m]$ and $Y[j \ldots n]$. We can solve the problem by changing the definition as $LCS(i, j)$ is the length of the LCS with $X[1 \ldots i]$ and $Y[1 \ldots j]$.

Printing the subsequence: The above algorithm can find the length of the longest common subsequence but cannot give the actual longest subsequence. To get the sequence, we trace it through the table. Start at cell $(0, 0)$. We know that the value of $LCS[0][0]$ was the maximum of 3 values of the neighboring cells. So we simply recompute $LCS[0][0]$ and note which cell gave the maximum value. Then we move to that cell (it will be one of $(1, 1)$, $(0, 1)$ or $(1, 0)$) and repeat this until we hit the boundary of the table. Every time we pass through a cell (i, j) where $X[i] == Y[j]$, we have a matching pair and print $X[i]$. At the end, we will have printed the longest common subsequence in $O(mn)$ time.

An alternative way of getting path is to keep a separate table for each cell. This will tell us which direction we came from when computing the value of that cell. At the end, we again start at cell $(0, 0)$ and follow these directions until the opposite corner of the table.

From the above examples, I hope you understood the idea behind DP. Now let us see more problems which can be easily solved using the DP technique.

Note: As we have seen above, in DP the main component is recursion. If we know the recurrence then converting that to code is a minimal task. For the problems below, we concentrate on getting the recurrence.

19.9 Dynamic Programming: Problems & Solutions

Problem-1 Convert the following recurrence to code.

$T(0) = T(1) = 2$

$$T(n) = \sum_{i=1}^{n-1} 2 \times T(i) \times T(i-1), \text{for } n > 1$$

Solution: The code for the given recursive formula can be given as:

```
def f(n) :
    sum = 0
    if(n==0 or n==1):
        return 2
    # Recursive case
    for i in range(1, n):
        sum += 2 * f(i) * f(i-1)
    return sum
```

Problem-2 Can we improve the solution to Problem-1 using memoization of DP?

Solution: Yes. Before finding a solution, let us see how the values are calculated.

$$T(0) = T(1) = 2$$
$$T(2) = 2 * T(1) * T(0)$$
$$T(3) = 2 * T(1) * T(0) + 2 * T(2) * T(1)$$
$$T(4) = 2 * T(1) * T(0) + 2 * T(2) * T(1) + 2 * T(3) * T(2)$$

From the above calculations it is clear that there are lots of repeated calculations with the same input values. Let us use a table for avoiding these repeated calculations, and the implementation can be given as:

```
def f2(n) :
    T = [0] * (n+1)
    T[0] = T[1] = 2
    for i in range(2, n+1):
        T[i] = 0
        for j in range(1, i):
            T[i] +=2 * T[j] * T[j-1]
    return T[n]
print f2(4)
```

Time Complexity: $O(n^2)$, two *for* loops. Space Complexity: $O(n)$, for table.

Problem-3 Can we further improve the complexity of Problem-2?

Solution: Yes, since all sub problem calculations are dependent only on previous calculations, code can be modified as:

```
def f(n):
    T = [0] * (n+1)
    T[0] = T[1] = 2
    T[2] = 2 * T[0] * T[1]
    for i in range(3, n+1):
        T[i]=T[i-1] + 2 * T[i-1] * T[i-2]
    return T[n]
print f(4)
```

Time Complexity: $O(n)$, since only one *for* loop. Space Complexity: $O(n)$.

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

Solution:

Input: Array $A(1)$ ... $A(n)$ of n numbers.

Goal: If there are no negative numbers, then the solution is just the sum of all elements in the given array. If negative numbers are there, then our aim is to maximize the sum [there can be a negative number in the contiguous sum].

One simple and brute force approach is to see all possible sums and select the one which has maximum value.

```
def MaxContigousSum(A):
    maxSum = 0
```

```
    n = len(A)
    for i in range(1, n):
        for j in range(i, n):
            currentSum = 0
            for k in range(i, j+1):
                currentSum += A[k]
                if(currentSum > maxSum):
                    maxSum = currentSum
    return maxSum;
A = [-2, 3, -16, 100, -4, 5]
print MaxContigousSum(A)
```

Time Complexity: $O(n^3)$. Space Complexity: $O(1)$.

Problem-5 Can we improve the complexity of Problem-4?

Solution: Yes. One important observation is that, if we have already calculated the sum for the subsequence $i, \ldots, j-1$, then we need only one more addition to get the sum for the subsequence $i, \ldots, j$. But, the Problem-4 algorithm ignores this information. If we use this fact, we can get an improved algorithm with the running time $O(n^2)$.

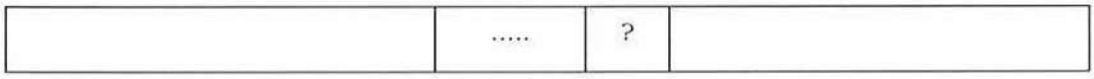
```
def MaxContigousSum(A):
    maxSum = 0
    n = len(A)
    for i in range(1, n):
        currentSum = 0
        for j in range(i, n):
            currentSum += A[j]
            if(currentSum > maxSum):
                maxSum = currentSum
    return maxSum;
A = [-2, 3, -16, 100, -4, 5]
print MaxContigousSum(A)
```

Time Complexity: $O(n^2)$. Space Complexity: $O(1)$.

Problem-6 Can we solve Problem-4 using Dynamic Programming?

Solution: Yes. For simplicity, let us say, $M(i)$ indicates maximum sum over all windows ending at i.

Given Array, A: recursive formula considers the case of selecting i^{th} element



$A[i]$

To find maximum sum we have to do one of the following and select maximum among them.

- Either extend the old sum by adding $A[i]$
- or start new window starting with one element $A[i]$

$$M(i) = Max\begin{cases} M(i-1) + A[i] \\ 0 \end{cases}$$

Where, $M(i-1) + A[i]$ indicates the case of extending the previous sum by adding $A[i]$ and 0 indicates the new window starting at $A[i]$.

```
def MaxContigousSum(A):
    maxSum = 0
    n = len(A)
    M = [0] * (n+1)
    if(A[0] > 0):
        M[0] = A[0]
    else: M[0] = 0
    for i in range(1, n):
        if( M[i-1] + A[i] > 0):
            M[i] = M[i-1] + A[i]
        else:   M[i] = 0
```

```
    for i in range(0, n):
            if(M[i]  > maxSum):
                    maxSum = M[i]
    return maxSum
A = [-2, 3, -16, 100, -4, 5]
print MaxContigousSum(A)
```

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

Problem-7 Is there any other way of solving Problem-4?

Solution: Yes. We can solve this problem without DP too (without memory). The algorithm is a little tricky. One simple way is to look for all positive contiguous segments of the array (*sumEndingHere*) and keep track of the maximum sum contiguous segment among all positive segments (*sumSoFar*). Each time we get a positive sum compare it (*sumEndingHere*) with *sumSoFar* and update *sumSoFar* if it is greater than *sumSoFar*. Let us consider the following code for the above observation.

```
def MaxContigousSum(A):
    sumSoFar = sumEndingHere = 0
    n = len(A)
    for i in range(0, n) :
            sumEndingHere = sumEndingHere + A[i]
            if(sumEndingHere < 0):
                    sumEndingHere = 0
                    continue
            if(sumSoFar < sumEndingHere):
                    sumSoFar = sumEndingHere
    return sumSoFar
A = [-2, 3, -16, 100, -4, 5]
print MaxContigousSum(A)
```

Note: The algorithm doesn't work if the input contains all negative numbers. It returns 0 if all numbers are negative. To overcome this, we can add an extra check before the actual implementation. The phase will look if all numbers are negative, and if they are it will return maximum of them (or smallest in terms of absolute value).

Time Complexity: $O(n)$, because we are doing only one scan. Space Complexity: O(1), for table.

Problem-8 In Problem-7 solution, we have assumed that $M(i)$ indicates maximum sum over all windows ending at i. Can we assume $M(i)$ indicates maximum sum over all windows starting at i and ending at n?

Solution: Yes. For simplicity, let us say, $M(i)$ indicates maximum sum over all windows starting at i.

Given Array, A: recursive formula considers the case of selecting i^{th} element

$A[i]$

To find maximum window we have to do one of the following and select maximum among them.

- Either extend the old sum by adding $A[i]$
- Or start new window starting with one element $A[i]$

$$M(i) = Max\begin{cases} M(i+1) + A[i], & if\ M(i+1) + A[i] > 0 \\ 0 \quad , & if\ M(i+1) + A[i] <= 0 \end{cases}$$

Where, $M(i+1) + A[i]$ indicates the case of extending the previous sum by adding $A[i]$, and 0 indicates the new window starting at $A[i]$.

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

Note: For $O(nlogn)$ solution, refer to the *Divide and Conquer* chapter.

Problem-9 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. Here the condition is we should not select *two* contiguous numbers.

Solution: Let us see how DP solves this problem. Assume that $M(i)$ represents the maximum sum from 1 to i numbers without selecting two contiguous numbers. While computing $M(i)$, the decision we have to make is,

whether to select the i^{th} element or not. This gives us two possibilities and based on this we can write the recursive formula as:

$$M(i) = \begin{cases} Max\{A[i] + M(i-2), M(i-1)\}, & if\ i > 2 \\ A[1], & if\ i = 1 \\ Max\{A[1], A[2]\}, & if\ i = 2 \end{cases}$$

- The first case indicates whether we are selecting the i^{th} element or not. If we don't select the i^{th} element then we have to maximize the sum using the elements 1 to $i-1$. If i^{th} element is selected then we should not select $i-1^{th}$ element and need to maximize the sum using 1 to $i-2$ elements.
- In the above representation, the last two cases indicate the base cases.

Given Array, A: recursive formula considers the case of selecting i^{th} element


```
def maxSumWithNoTwoContinuousNumbers(A):
    n = len(A)
    M = [0] * (n+1)
    M[0] = A[0]
    if(A[0]>A[1]):
            M[0] = A[0]
    else: M[0] = A[1]
    for i in range(2, n):
            if( M[i-1]>M[i-2]+A[i]):
                    M[i] = M[i-1]
            else:   M[i] = M[i-2]+A[i]
    return M[n-1]
A = [-2, 3, -16, 100, -4, 5]
print maxSumWithNoTwoContinuousNumbers(A)
```

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

Problem-10 In Problem-9, we assumed that $M(i)$ represents the maximum sum from 1 to i numbers without selecting two contiguous numbers. Can we solve the same problem by changing the definition as: $M(i)$ represents the maximum sum from i to n numbers without selecting two contiguous numbers?

Solution: Yes. Let us assume that $M(i)$ represents the maximum sum from i to n numbers without selecting two contiguous numbers:

$$M(i) = \begin{cases} Max\{A[i] + M(i+2), M(i+1)\}, & if\ i > 2 \\ A[1], & if\ i = 1 \\ Max\{A[1], A[2]\}, & if\ i = 2 \end{cases}$$

Given Array, A: recursive formula considers the case of selecting i^{th} element

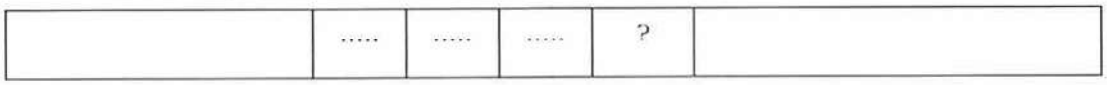
- The first case indicates whether we are selecting the i^{th} element or not. If we don't select the i^{th} element then we have to maximize the sum using the elements $i+1$ to n. If i^{th} element is selected then we should not select $i+1^{th}$ element need to maximize the sum using $i+2$ to n elements.
- In the above representation, the last two cases indicate the base cases.

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

Problem-11 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. Here the condition is we should not select *three* continuous numbers.

Solution: Input: Array $A(1) \ldots A(n)$ of n numbers.

Given Array, A: recursive formula considers the case of selecting i^{th} element



Assume that $M(i)$ represents the maximum sum from 1 to i numbers without selecting three contiguous numbers. While computing $M(i)$, the decision we have to make is, whether to select i^{th} element or not. This gives us the following possibilities:

$$M(i) = Max\begin{cases} A[i] + A[i-1] + M(i-3) \\ A[i] + M(i-2) \\ M(i-1) \end{cases}$$

- In the given problem the restriction is not to select three continuous numbers, but we can select two elements continuously and skip the third one. That is what the first case says in the above recursive formula. That means we are skipping $A[i-2]$.
- The other possibility is, selecting i^{th} element and skipping second $i-1^{th}$ element. This is the second case (skipping $A[i-1]$).
- The third term defines the case of not selecting i^{th} element and as a result we should solve the problem with $i-1$ elements.

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

Problem-12 In Problem-11, we assumed that $M(i)$ represents the maximum sum from 1 to i numbers without selecting three contiguous numbers. Can we solve the same problem by changing the definition as: $M(i)$ represents the maximum sum from i to n numbers without selecting three contiguous numbers?

Solution: Yes. The reasoning is very much similar. Let us see how DP solves this problem. Assume that $M(i)$ represents the maximum sum from i to n numbers without selecting three contiguous numbers.

Given Array, A: recursive formula considers the case of selecting i^{th} element

While computing $M(i)$, the decision we have to make is, whether to select i^{th} element or not. This gives us the following possibilities:

$$M(i) = Max\begin{cases} A[i] + A[i+1] + M(i+3) \\ A[i] + M(i+2) \\ M(i+1) \end{cases}$$

- In the given problem the restriction is to not select three continuous numbers, but we can select two elements continuously and skip the third one. That is what the first case says in the above recursive formula. That means we are skipping $A[i+2]$.
- The other possibility is, selecting i^{th} element and skipping second $i-1^{th}$ element. This is the second case (skipping $A[i+1]$).
- And the third case is not selecting i^{th} element and as a result we should solve the problem with $i+1$ elements.

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

Problem-13 There are n petrol stations along a circular route, where the amount of petrol at station i is petrol[i]. You have a car with an unlimited petrol tank and it costs cost[i] of petrol to travel from station i to its next station $(i+1)$. You begin the journey with an empty tank at one of the petrol stations. Return the starting petrol station's index if you can travel around the circuit once, otherwise return -1.

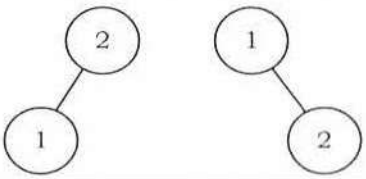
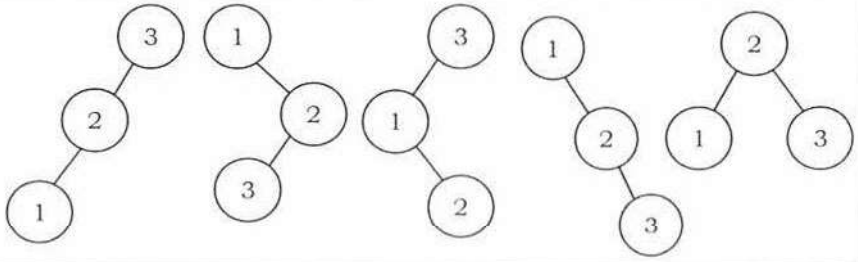
Solution: This is just alternative way of asking the Problem-4. We need to make sure that the value should never go less than zero.

```
def canCompleteTour(self, petrol, cost):
    minVal = float("inf")
    minPos = -1
    petrolTillNow = 0
    for i in range(0, len(petrol)):
```

```
        petrolTillNow += petrol[i] - cost[i]
        if petrolTillNow < minVal:
            minVal = petrolTillNow
            minPos = i
    if petrolTillNow >=0:
        return (minPos + 1) % len(petrol)
    return -1
```

Problem-14 **Catalan Numbers:** How many binary search trees are there with n vertices?

Solution: Binary Search Tree (BST) is a tree where the left subtree elements are less than the root element, and the right subtree elements are greater than the root element. This property should be satisfied at every node in the tree. The number of BSTs with n nodes is called *Catalan Number* and is denoted by C_n. For example, there are 2 BSTs with 2 nodes (2 choices for the root) and 5 BSTs with 3 nodes.

Number of nodes, n	Number of Trees
1	
2	
3	

Let us assume that the nodes of the tree are numbered from 1 to n. Among the nodes, we have to select some node as root, and then divide the nodes which are less than root node into left sub tree, and elements greater than root node into right sub tree. Since we have already numbered the vertices, let us assume that the root element we selected is i^{th} element.

If we select i^{th} element as root then we get $i-1$ elements on left sub-tree and $n-i$ elements on right sub tree. Since C_n is the Catalan number for n elements, C_{i-1} represents the Catalan number for left sub tree elements ($i-1$ elements) and C_{n-i} represents the Catalan number for right sub tree elements. The two sub trees are independent of each other, so we simply multiply the two numbers. That means, the Catalan number for a fixed i value is $C_{i-1} \times C_{n-i}$.

Since there are n nodes, for i we will get n choices. The total Catalan number with n nodes can be given as:

$$C_n = \sum_{i=1}^{n} C_{i-1} \times C_{n-i}$$

```
def CatalanRecursive(n):
    if n == 0:
        return 1
    else:
        count = 0
        for i in range(n):
            count += CatalanRecursive(i) * CatalanRecursive(n - 1 - i)
        return count
print CatalanRecursive(4)
```

Time Complexity: $O(4^n)$. For proof, refer *Introduction* chapter.

Problem-15 Can we improve the time complexity of Problem-144 using DP?

Solution: The recursive call C_n depends only on the numbers C_0 to C_{n-1} and for any value of i, there are a lot of recalculations. We will keep a table of previously computed values of C_i. If the function *CatalanNumber*() is called with parameter i, and if it has already been computed before, then we can simply avoid recalculating the same subproblem.

```
def CatalanNumber(n):
    catalan=[1,1]+[0]*n
    for i in range(2,n+1):
        for j in range(n):
            catalan[i]+=catalan[j]*catalan[i-j-1]
    return catalan[n]
print CatalanNumber(4)
```

The time complexity of this implementation $O(n^2)$, because to compute *CatalanNumber*(n), we need to compute all of the *CatalanNumber*(i) values between 0 and $n-1$, and each one will be computed exactly once, in linear time.

In mathematics, Catalan Number can be represented by direct equation as: $\frac{(2n)!}{n!(n+1)!}$.

```
catalan=[]
#1st term is 1
catalan.append(1)
for i in range (1,1001):
    x=catalan[i-1]*(4*i-2)/(i+1)
    catalan.append(x)
def CatalanNumber(n):
    return catalan[n]
print CatalanNumber(4)
```

Problem-16 **Matrix Product Parenthesizations:** Given a series of matrices: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$with their dimensions, what is the best way to parenthesize them so that it produces the minimum number of total multiplications. Assume that we are using standard matrix and not Strassen's matrix multiplication algorithm.

Solution: Input: Sequence of matrices $A_1 \times A_2 \times A_3 \times \ldots \times A_n$, where A_i is a $P_{i-1} \times P_i$. The dimensions are given in an array P.

Goal: Parenthesize the given matrices in such a way that it produces the optimal number of multiplications needed to compute $A_1 \times A_2 \times A_3 \times \ldots \times A_n$.

For the matrix multiplication problem, there are many possibilities. This is because matrix multiplication is associative. It does not matter how we parenthesize the product, the result will be the same. As an example, for four matrices A, B, C, and D, the possibilities could be:

$$(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = ..$$

Multiplying $(p \times q)$ matrix with $(q \times r)$ matrix requires pqr multiplications. Each of the above possibilities produces a different number of products during multiplication. To select the best one, we can go through each possible parenthesization (brute force), but this requires $O(2^n)$ time and is very slow. Now let us use DP to improve this time complexity. Assume that, $M[i,j]$ represents the least number of multiplications needed to multiply $A_i \cdots A_j$.

$$M[i,j] = \begin{cases} 0 & ,if\ i = j \\ Min\{M[i,k] + M[k+1,j] + P_{i-1}P_kP_j\}, & if\ i < j \end{cases}$$

The above recursive formula says that we have to find point k such that it produces the minimum number of multiplications. After computing all possible values for k, we have to select the k value which gives minimum value. We can use one more table (say, $S[i,j]$) to reconstruct the optimal parenthesizations. Compute the $M[i,j]$ and $S[i,j]$ in a bottom-up fashion.

```
import sys, time
gk = lambda i,j:str(i)+','+str(j)
MAX = sys.maxint
def matrixMultiplicationWithDP(p):
    n = len(p)-1
    m = {}
    for i in xrange(1, n+1):
        for j in xrange (i, n+1):
```

```
            m[gk(i, j)] = MAX
    return lookup_chain(m, p, 1, n)
def lookup_chain(m, p, i, j):
    if m[gk(i, j)] < MAX:
        return m[gk(i, j)]
    if i == j:
        m[gk(i, j)] = 0
    else:
        for k in xrange(i, j):
            q = lookup_chain(m, p, i, k) + lookup_chain(m, p, k+1, j) + p[i-1]*p[k]*p[j]
            if q < m[gk(i, j)]:
                m[gk(i, j)] = q
    return m[gk(i, j)]
p = [30,35,15,5,10,20,25,5,16,34,28,19,66,34,78,55,23]
print matrixMultiplicationWithDP(p)
```

How many sub problems are there? In the above formula, i can range from 1 *to* n and j can range from 1 *to* n. So there are a total of n^2 subproblems, and also we are doing $n-1$ such operations [since the total number of operations we need for $A_1 \times A_2 \times A_3 \times \ldots \times A_n$ ise $n-1$]. So the time complexity is $O(n^3)$. Space Complexity: $O(n^2)$.

Problem-17 For the Problem-16, can we use greedy method?

Solution: *Greedy* method is not an optimal way of solving this problem. Let us go through some counter example for this. As we have seen already, *greedy* method makes the decision that is good locally and it does not consider the future optimal solutions. In this case, if we use *Greedy*, then we always do the cheapest multiplication first. Sometimes it returns a parenthesization that is not optimal.

Example: Consider $A_1 \times A_2 \times A_3$ with dimentions 3×100, 100×2 and 2×2. Based on *greedy* we parenthesize them as: $A_1 \times (A_2 \times A_3)$ with $100 \cdot 2 \cdot 2 + 3 \cdot 100 \cdot 2 = 1000$ multiplications. But the optimal solution to this problem is: $(A_1 \times A_2) \times A_3$ with $3 \cdot 100 \cdot 2 + 3 \cdot 2 \cdot 2 = 612$ multiplications. $\therefore$ we cannot use *greedy* for solving this problem.

Problem-18 **Integer Knapsack Problem [Duplicate Items Permitted]:** Given n types of items, where the i^{th} item type has an integer size s_i and a value v_i. We need to fill a knapsack of total capacity C with items of maximum value. We can add multiple items of the same type to the knapsack.
Note: For Fractional Knapsack problem refer to *Greedy Algorithms* chapter.

Solution: Input: n types of items where i^{th} type item has the size s_i and value v_i. Also, assume infinite number of items for each item type.

Goal: Fill the knapsack with capacity C by using n types of items and with maximum value.

One important note is that it's not compulsory to fill the knapsack completely. That means, filling the knapsack completely [of size C] if we get a value V and without filling the knapsack completely [let us say $C-1$] with value U and if $V < U$ then we consider the second one. In this case, we are basically filling the knapsack of size $C-1$. If we get the same situation for $C-1$ also, then we try to fill the knapsack with $C-2$ size and get the maximum value.

Let us say M(j) denotes the maximum value we can pack into a j size knapsack. We can express M(j) recursively in terms of solutions to sub problems as follows:

$$M(j) = \begin{cases} max\{M(j-1), max_{i=1\ to\ n}(M(j-s_i)) + v_i\}, & if\ j \geq 1 \\ 0, & if\ j \leq 0 \end{cases}$$

For this problem the decision depends on whether we select a particular i^{th} item or not for a knapsack of size j.

- If we select i^{th} item, then we add its value v_i to the optimal solution and decrease the size of the knapsack to be solved to $j - s_i$.
- If we do not select the item then check whether we can get a better solution for the knapsack of size $j-1$.

The value of $M(C)$ will contain the value of the optimal solution. We can find the list of items in the optimal solution by maintaining and following "back pointers".

Time Complexity: Finding each $M(j)$ value will require $\Theta(n)$ time, and we need to sequentially compute C such values. Therefore, total running time is $\Theta(nC)$.
Space Complexity: $\Theta(C)$.

Problem-19 0-1 Knapsack Problem: For Problem-18, how do we solve it if the items are not duplicated (not having an infinite number of items for each type, and each item is allowed to be used for 0 or 1 time)?

Real-time example: Suppose we are going by flight, and we know that there is a limitation on the luggage weight. Also, the items which we are carrying can be of different types (like laptops, etc.). In this case, our objective is to select the items with maximum value. That means, we need to tell the customs officer to select the items which have more weight and less value (profit).

Solution: Input is a set of n items with sizes s_i and values v_i and a Knapsack of size C which we need to fill with a subset of items from the given set. Let us try to find the recursive formula for this problem using DP. Let $M(i,j)$ represent the optimal value we can get for filling up a knapsack of size j with items $1 \dots i$. The recursive formula can be given as:

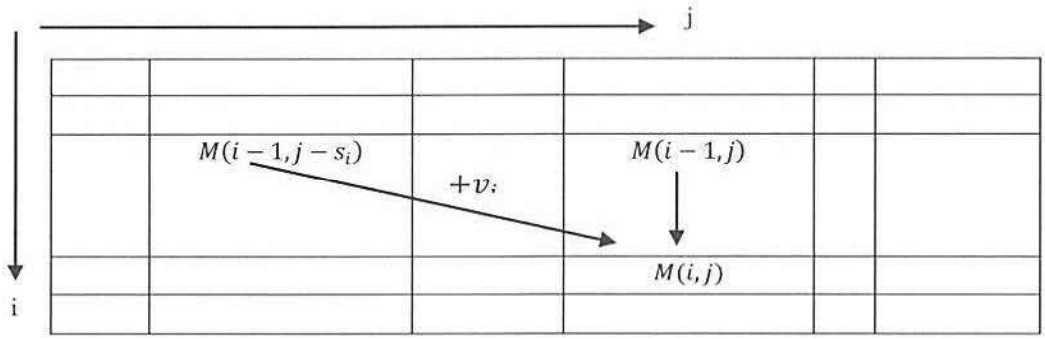
$$M(i,j) = Max\{M(i-1,j), M(i-1,j-s_i) + v_i\}$$

i^{th} item is not used

i^{th} item is used

Time Complexity: $O(nC)$, since there are nC subproblems to be solved and each of them takes O(1) to compute. Space Complexity: $O(nC)$, where as Integer Knapsack takes only $O(C)$.

Now let us consider the following diagram which helps us in reconstructing the optimal solution and also gives further understanding. Size of below matrix is M.



Since i takes values from $1 \dots n$ and j takes values from $1 \dots C$, there are a total of nC subproblems. Now let us see what the above formula says:

- $M(i-1,j)$: Indicates the case of not selecting the i^{th} item. In this case, since we are not adding any size to the knapsack we have to use the same knapsack size for subproblems but excluding the i^{th} item. The remaining items are $i-1$.
- $M(i-1,j-s_i) + v_i$ indicates the case where we have selected the i^{th} item. If we add the i^{th} item then we have to reduce the subproblem knapsack size to $j-s_i$ and at the same time we need to add the value v_i to the optimal solution. The remaining items are $i-1$.

Now, after finding all $M(i,j)$ values, the optimal objective value can be obtained as: $Max_j\{M(n,j)\}$
This is because we do not know what amount of capacity gives the best solution.

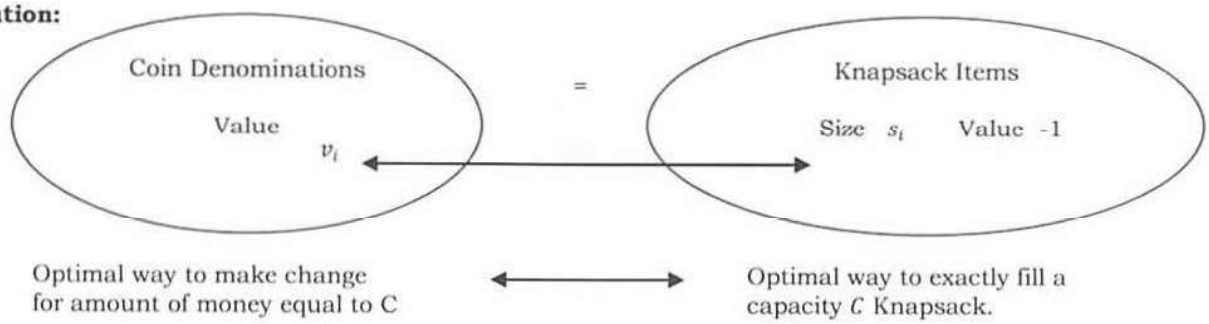
In order to compute some value $M(i,j)$, we take the maximum of $M(i-1,j)$ and $M(i-1,j-s_i)+v_i$. These two values ($M(i,j)$ and $M(i-1,j-s_i)$) appear in the previous row and also in some previous columns. So, $M(i,j)$ can be computed just by looking at two values in the previous row in the table.

```
def Knapsack(knapsackSize, itemsValue, itemsWeight):
    numItems = len(itemsValue)
    M =[[0 for x in range(knapsackSize+1)] for x in range(len(itemsValue))]
    for i in range(1, numItems):
        for j in range(knapsackSize+1):
            value = itemsValue[i]
            weight = itemsWeight[i]
            if weight > j:
                M[i][j] = M[i-1][j]
            else:
                M[i][j] = max(M[i-1][j], M[i-1][j-weight] + value)
    return M[numItems-1][knapsackSize]
```

```
print Knapsack(50, [60,100,120], [10,20,30])
```

Problem-20 **Making Change:** Given n types of coin denominations of values $v_1 < v_2 < \ldots < v_n$ (integers). Assume $v_1 = 1$, so that we can always make change for any amount of money C. Give an algorithm which makes change for an amount of money C with as few coins as possible.

Solution:



This problem is identical to the Integer Knapsack problem. In our problem, we have coin denominations, each of value v_i. We can construct an instance of a Knapsack problem for each item that has a size s_i, which is equal to the value of v_i coin denomination. In the Knapsack we can give the value of every item as -1.

Now it is easy to understand an optimal way to make money C with the fewest coins is completely equivalent to the optimal way to fill the Knapsack of size C. This is because since every value has a value of -1, and the Knapsack algorithm uses as few items as possible which correspond to as few coins as possible.

Let us try formulating the recurrence. Let $M(j)$ indicate the minimum number of coins required to make change for the amount of money equal to j.

$$M(j) = Min_i\{M(j - v_i)\} + 1$$

What this says is, if coin denomination i was the last denomination coin added to the solution, then the optimal way to finish the solution with that one is to optimally make change for the amount of money $j - v_i$ and then add one extra coin of value v_i.

```
def MakingChange(coins,change,minimumCoins,coinsUsed):
  for cents in range(change+1):
    coinCount = cents
    newCoin = 1
    for j in [c for c in coins if c <= cents]:
        if minimumCoins[cents-j] + 1 < coinCount:
          coinCount = minimumCoins[cents-j]+1
          newCoin = j
    minimumCoins[cents] = coinCount
    coinsUsed[cents] = newCoin
  return minimumCoins[change]
```

Time Complexity: $O(nC)$. Since we are solving C sub-problems and each of them requires minimization of n terms.
Space Complexity: $O(nC)$.

Problem-21 **Longest Increasing Subsequence:** Given a sequence of n numbers $A_1 \ldots A_n$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly increasing sequence.

Solution:

Input: Sequence of n numbers $A_1 \ldots A_n$.

Goal: To find a subsequence that is just a subset of elements and does not happen to be contiguous. But the elements in the subsequence should form a strictly increasing sequence and at the same time the subsequence should contain as many elements as possible.

For example, if the sequence is (5,6,2,3,4,1,9,9,8,9,5), then (5,6), (3,5), (1,8,9) are all increasing sub-sequences. The longest one of them is (2, 3, 4, 8, 9), and we want an algorithm for finding it.

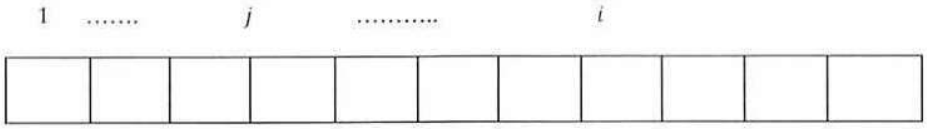
First, let us concentrate on the algorithm for finding the longest subsequence. Later, we can try printing the sequence itself by tracing the table. Our first step is finding the recursive formula. First, let us create the base conditions. If there is only one element in the input sequence then we don't have to solve the problem and we

just need to return that element. For any sequence we can start with the first element ($A[1]$). Since we know the first number in the LIS, let's find the second number ($A[2]$). If $A[2]$ is larger than $A[1]$ then include $A[2]$ also. Otherwise, we are done – the LIS is the one element sequence ($A[1]$).

Now, let us generalize the discussion and decide about i^{th} element. Let $L(i)$ represent the optimal subsequence which is starting at position $A[1]$ and ending at $A[i]$. The optimal way to obtain a strictly increasing subsequence ending at position i is to extend some subsequence starting at some earlier position j. For this the recursive formula can be written as:

$$L(i) = Max_{j<i \text{ and } A[j]<A[i]}\{L(j)\} + 1$$

The above recurrence says that we have to select some earlier position j which gives the maximum sequence. The 1 in the recursive formula indicates the addition of i^{th} element.



Now after finding the maximum sequence for all positions we have to select the one among all positions which gives the maximum sequence and it is defined as:

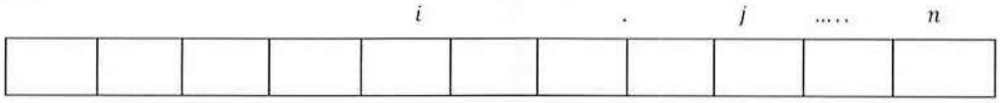
$$Max_i\{L(i)\}$$

```
def LongestIncreasingSequence(numList):
    LISTable = [1]
    for i in range(1, len(numList)):
        LISTable.append(1)
        for j in range(0, i):
            if numList[i] > numList[j] and LISTable[i]<=LISTable[j]:
                LISTable[i] = 1 + LISTable[j]
    print LISTable
    return max(LISTable)

print LongestIncreasingSequence([3,2,6,4,5,1])
```

Time Complexity: $O(n^2)$, since two *for* loops. Space Complexity: $O(n)$, for table.

Problem-22 **Longest Increasing Subsequence:** In Problem-21, we assumed that $L(i)$ represents the optimal subsequence which is starting at position $A[1]$ and ending at $A[i]$. Now, let us change the definition of $L(i)$ as: $L(i)$ represents the optimal subsequence which is starting at position $A[i]$ and ending at $A[n]$. With this approach can we solve the problem?

Solution: Yes.



Let $L(i)$ represent the optimal subsequence which is starting at position $A[i]$ and ending at $A[n]$. The optimal way to obtain a strictly increasing subsequence starting at position i is going to be to extend some subsequence starting at some later position j. For this the recursive formula can be written as:

$$L(i) = Max_{i<j \text{ and } A[i]<A[j]}\{L(j)\} + 1$$

We have to select some later position j which gives the maximum sequence. The 1 in the recursive formula is the addition of i^{th} element. After finding the maximum sequence for all positions select the one among all positions which gives the maximum sequence and it is defined as:

$$Max_i\{L(i)\}$$

Problem-23 Is there an alternative way of solving *Problem-22?*

Solution: Yes. The other method is to sort the given sequence and save it into another array and then take out the "Longest Common Subsequence" (LCS) of the two arrays. This method has a complexity of $O(n^2)$. For LCS problem refer *theory section* of this chapter.

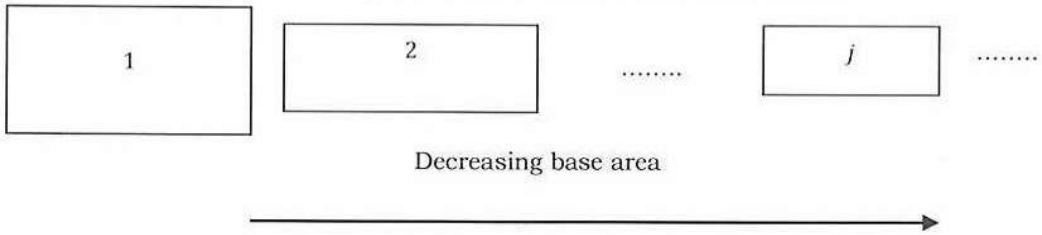
Problem-24 **Box Stacking:** Assume that we are given a set of n rectangular $3-D$ boxes. The dimensions of i^{th} box are height h_i, width w_i and depth d_i. Now we want to create a stack of boxes which is as tall as

possible, but we can only stack a box on top of another box if the dimensions of the 2 –D base of the lower box are each strictly larger than those of the 2 –D base of the higher box. We can rotate a box so that any side functions as its base. It is possible to use multiple instances of the same type of box.

Solution: Box stacking problem can be reduced to LIS [*Problem-22*].

Input: n boxes where i^{th} with height h_i, width w_i and depth d_i. For all n boxes we have to consider all the orientations with respect to rotation. That is, if we have, in the original set, a box with dimensions $1 \times 2 \times 3$, then we consider 3 boxes,

$$1 \times 2 \times 3 \Rightarrow \begin{cases} 1 \times (2 \times 3), with\ height\ 1, base\ 2\ and\ width\ 3 \\ 2 \times (1 \times 3), with\ height\ 2, base\ 1\ and\ width\ 3 \\ 3 \times (1 \times 2), with\ height\ 3, base\ 1\ and\ width\ 2 \end{cases}$$



This simplification allows us to forget about the rotations of the boxes and we just focus on the stacking of **n** boxes with each height as h_i and a base area of $(w_i \times d_i)$. Also assume that $w_i \leq d_i$. Now what we do is, make a stack of boxes that is as tall as possible and has maximum height. We allow a box i on top of box j only if box i is smaller than box j in both the dimensions. That means, if $w_i < w_j \,\&\&\, d_i < d_j$. Now let us solve this using DP. First select the boxes in the order of decreasing base area.

Now, let us say $H(j)$ represents the tallest stack of boxes with box j on top. This is very similar to the LIS problem because the stack of n boxes with ending box j is equal to finding a subsequence with the first j boxes due to the sorting by decreasing base area. The order of the boxes on the stack is going to be equal to the order of the sequence.

Now we can write $H(j)$ recursively. In order to form a stack which ends on box j, we need to extend a previous stack ending at i. That means, we need to put j box at the top of the stack [i box is the current top of the stack]. To put j box at the top of the stack we should satisfy the condition $w_i > w_j and\ d_i > d_j$ [this ensures that the low level box has more base than the boxes above it]. Based on this logic, we can write the recursive formula as:

$$H(j) = Max_{i<j\ and\ w_i>w_j and\ d_i>d_j}\{H(i)\} + h_i$$

Similar to the LIS problem, at the end we have to select the best j over all potential values. This is because we are not sure which box might end up on top.

$$Max_j\{H(j)\}$$

Time Complexity: $O(n^2)$.

Problem-25 Building Bridges in India: Consider a very long, straight river which moves from north to south. Assume there are n cities on both sides of the river: n cities on the left of the river and n cities on the right side of the river. Also, assume that these cities are numbered from 1 to n but the order is not known. Now we want to connect as many left-right pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, we can only connect city i on the left side to city i on the right side.

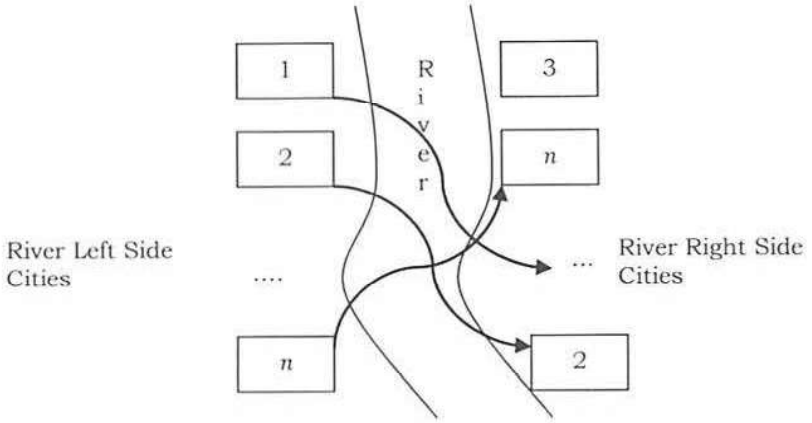
Solution:

Input: Two pairs of sets with each numbered from 1 to n.

Goal: Construct as many bridges as possible without any crosses between left side cities to right side cities of the river.

To understand better let us consider the diagram below. In the diagram it can be seen that there are n cities on the left side of river and n cities on the right side of river. Also, note that we are connecting the cities which have the same number [a requirement in the problem]. Our goal is to connect the maximum cities on the left side of river to cities on the right side of the river, without any cross edges. Just to make it simple, let us sort the cities on one side of the river.

If we observe carefully, since the cities on the left side are already sorted, the problem can be simplified to finding the maximum increasing sequence. That means we have to use the LIS solution for finding the maximum increasing sequence on the right side cities of the river.



Time Complexity: $O(n^2)$, (same as LIS).

Problem-26 **Subset Sum:** Given a sequence of n positive numbers $A_1 \ldots A_n$, give an algorithm which checks whether there exists a subset of A whose sum of all numbers is T?

Solution: This is a variation of the Knapsack problem. As an example, consider the following array:

$$A = [3, 2, 4, 19, 3, 7, 13, 10, 6, 11]$$

Suppose we want to check whether there is any subset whose sum is 17. The answer is yes, because the sum of $4 + 13 = 17$ and therefore $\{4, 13\}$ is such a subset.

Let us try solving this problem using DP. We will define $n \times T$ matrix, where n is the number of elements in our input array and T is the sum we want to check. Let, $M[i, j] = 1$ if it is possible to find a subset of the numbers 1 through i that produce sum j and $M[i, j] = 0$ otherwise.

$$M[i, j] = Max(M[i - 1, j], M[i - 1, j - A_i])$$

According to the above recursive formula similar to the Knapsack problem, we check if we can get the sum j by not including the element i in our subset, and we check if we can get the sum j by including i and checking if the sum $j - A_i$ exists without the i^{th} element. This is identical to Knapsack, except that we are storing 0/1's instead of values. In the below implementation we can use binary OR operation to get the maximum among $M[i - 1, j]$ and $M[i - 1, j - A_i]$.

```
def SubsetSum(A, T):
    n = len(A)
    M =[[0 for x in range(T+1)] for x in range(n+1)]
    M[0][0]=0
    for i in range(0, n+1):
            M[i][0] = 0
    for i in range(0, T+1):
            M[0][i] = 0
    for i in range(1,n+1):
            for j in range(1, T+1):
                    M[i][j] = M[i-1][j] or (M[i-1][j - A[i]])
    return M[n][T]
A = [3,2,4,19,3,7,13,10,6,11]
print SubsetSum(A, 17)
```

How many subproblems are there? In the above formula, i can range from 1 *to* n and j can range from 1 *to* T. There are a total of nT subproblems and each one takes $O(1)$. So the time complexity is $O(nT)$ and this is not polynomial as the running time depends on two variables [n and T], and we can see that they are anexponential function of the other.

Space Complexity: $O(nT)$.

Problem-27 Given a set of n integers and the sum of all numbers is at most K. Find the subset of these n elements whose sum is exactly half of the total sum of n numbers.

Solution: Assume that the numbers are $A_1 \ldots A_n$. Let us use DP to solve this problem. We will create a boolean array T with size equal to $K + 1$. Assume that $T[x]$ is 1 if there exists a subset of given n elements whose sum is x. That means, after the algorithm finishes, $T[K]$ will be 1, if and only if there is a subset of the numbers that has

sum K. Once we have that value then we just need to return $T[K/2]$. If it is 1, then there is a subset that adds up to half the total sum.

Initially we set all values of T to 0. Then we set $T[0]$ to 1. This is because we can always build 0 by taking an empty set. If we have no numbers in A, then we are done! Otherwise, we pick the first number, $A[0]$. We can either throw it away or take it into our subset. This means that the new $T[]$ should have $T[0]$ and $T[A[0]]$ set to 1. This creates the base case. We continue by taking the next element of A.

Suppose that we have already taken care of the first i – 1 elements of A. Now we take A[i] and look at our table T[]. After processing i – 1 elements, the array T has a 1 in every location that corresponds to a sum that we can make from the numbers we have already processed. Now we add the new number, A[i]. What should the table look like? First of all, we can simply ignore A[i]. That means, no one should disappear from T[] – we can still make all those sums. Now consider some location of T[j] that has a 1 in it. It corresponds to some subset of the previous numbers that add up to j. If we add A[i] to that subset, we will get a new subset with total sum j + A[i]. So we should set T[j + A[i]] to 1 as well. That's all. Based on the above discussion, we can write the algorithm as:

```
def SubsetSum2(A, T):
    n = len(A)
    T = [0] * (10240)
    K = 0
    for i in range(0, n):
            K += A[i]
    T[0] = 1
    for i in range(1, K):
            T[i] = 0
    # process the numbers one by one
    for i in range(0, n):
            for j in range(K - A[i],0, -1):
                    if( T[j] ):
                            T[j + A[i]] = 1
    return T[K / 2]
A = [3,2,4,19,3,7,13,10,6,11]
print SubsetSum2(A, 17)
```

In the above code, j loop moves from right to left. This reduces the double counting problem. That means, if we move from left to right, then we may do the repeated calculations.

Time Complexity: $O(nK)$, for the two *for* loops. Space Complexity: $O(K)$, for the boolean table T.

Problem-28 Can we improve the performance of Problem-27?

Solution: Yes. In the above code what we are doing is, the inner j loop is starting from K and moving left. That means, it is unnecessarily scanning the whole table every time.

What we actually want is to find all the 1 entries. At the beginning, only the 0^{th} entry is 1. If we keep the location of the rightmost 1 entry in a variable, we can always start at that spot and go left instead of starting at the right end of the table.

To take full advantage of this, we can sort $A[]$ first. That way, the rightmost 1 entry will move to the right as slowly as possible. Finally, we don't really care about what happens in the right half of the table (after $T[K/2]$) because if $T[x]$ is 1, then $T[Kx]$ must also be 1 eventually – it corresponds to the complement of the subset that gave us x. The code based on above discussion is given below.

```
def SubsetSum(A):
    n = len(A)
    K = 0
    for i in range(0, n):
            K += A[i]
    A.sort()
    T = [0] * ( K + 1 )
    T[0] = 1
    R = 0
    # process the numbers one by one
    for i in range(0, n):
            for j in range(R,-1, -1):
                    if( T[j] ):
                            T[j + A[i]] = 1
                    R = min(K/2, R+A[i])
```

```
    return T[K / 2]

A = [3,2,4,19,3,7,13,10,6,11]
print SubsetSum(A)
```

After the improvements, the time complexity is still $O(nK)$, but we have removed some useless steps.

Problem-29 Partition partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is the same [the same as the previous problem but a different way of asking]. For example, if A[] = {1, 5, 11, 5}, the array can be partitioned as {1, 5, 5} and {11}. Similarly, if A[] = {1, 5, 3}, the array cannot be partitioned into equal sum sets.

Solution: Let us try solving this problem another way. Following are the two main steps to solve this problem:

1. Calculate the sum of the array. If the sum is odd, there cannot be two subsets with an equal sum, so return false.
2. If the sum of the array elements is even, calculate $sum/2$ and find a subset of the array with a sum equal to $sum/2$.

The first step is simple. The second step is crucial, and it can be solved either using recursion or Dynamic Programming.

Recursive Solution: Following is the recursive property of the second step mentioned above. Let subsetSum(A, n, sum/2) be the function that returns true if there is a subset of A[0..n-1] with sum equal to $sum/2$. The isSubsetSum problem can be divided into two sub problems:

a) isSubsetSum() without considering last element (reducing n to $n-1$)
b) isSubsetSum considering the last element (reducing sum/2 by A[n-1] and n to $n-1$)

If any of the above sub problems return true, then return true.

$$subsetSum\ (A, n, sum/2) = isSubsetSum\ (A, n-1, sum/2)\ ||\ subsetSum\ (A, n-1, sum/2 - A[n-1])$$

```
# A utility function that returns 1 if there is a subset of A[] with sum equal to given sum
def subsetSum (A, n, sum):
    if (sum == 0):
            return 1
    if (n == 0 and sum != 0):
            return 0
    # If last element is greater than sum, then ignore it
    if (A[n-1] > sum):
            return subsetSum (A, n-1, sum)

    return subsetSum (A, n-1, sum) or subsetSum (A, n-1, sum-A[n-1])

# Returns 1 if A[] can be partitioned in two subsets of equal sum, otherwise 0
def findPartition(A):
    # calculate sum of all elements
    sum = 0
    n = len(A)
    for i in range(0,n):
            sum += A[i]

    # If sum is odd, there cannot be two subsets with equal sum
    if (sum%2 != 0):
            return 0

    # Find if there is subset with sum equal to half of total sum
    return subsetSum (A, n, sum/2)
```

Time Complexity: $O(2^n)$ In worst case, this solution tries two possibilities (whether to include or exclude) for every element.

Dynamic Programming Solution: The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array $part[][]$ of size $(sum/2)*(n+1)$. And we can construct the solution in a bottom-up manner such that every filled entry has a following property

$$part[i][j] = true\ if\ a\ subset\ of\ \{A[0], A[1], ..A[j-1]\}\ has\ sum\ equal\ to\ sum/2, otherwise\ false$$

```
# Returns 1 if A[] can be partitioned in two subsets of equal sum, otherwise 0
def findPartition(A):
    # calculate sum of all elements
    sum = 0
    n = len(A)
```

```
for i in range(0,n):
        sum += A[i]
# If sum is odd, there cannot be two subsets with equal sum
if (sum%2 != 0):
        return 0
Table = [[0 for x in range(n+1)] for x in range(sum//2 + 1)]
# initialize top row as true
for i in range(0,n):
        Table[0][i] = 1
# initialize leftmost column, except Table[0][0], as 0
for i in range(1,sum//2+1):
        Table[i][0] = 0
# Fill the partition table in bottom up manner
for i in range(1,sum//2+1):
        for j in range(0,n+1):
                Table[i][j] = Table[i][j-1]
                if (i >= A[j-1]):
                        Table[i][j] = Table[i][j] or Table[i - A[j-1]][j-1]
return Table[sum/2][n];
```

Time Complexity: O($sum \times n$). Space Complexity: O($sum \times n$). Please note that this solution will not be feasible for arrays with a big sum.

Problem-30 Counting Boolean Parenthesizations: Let us assume that we are given a boolean expression consisting of symbols *'true','false','and','or', and 'xor'*. Find the number of ways to parenthesize the expression such that it will evaluate to *true*. For example, there is only 1 way to parenthesize *'true and false xor true'* such that it evaluates to *true*.

Solution: Let the number of symbols be n and between symbols there are boolean operators like and, or, xor, etc. For example, if $n = 4$, *T or F and T xor F*. Our goal is to count the numbers of ways to parenthesize the expression with boolean operators so that it evaluates to *true*. In the above case, if we use *T or ((F and T) xor F)* then it evaluates to true.

$$T\ or(\ (F\ and\ T)xor\ F) = True$$

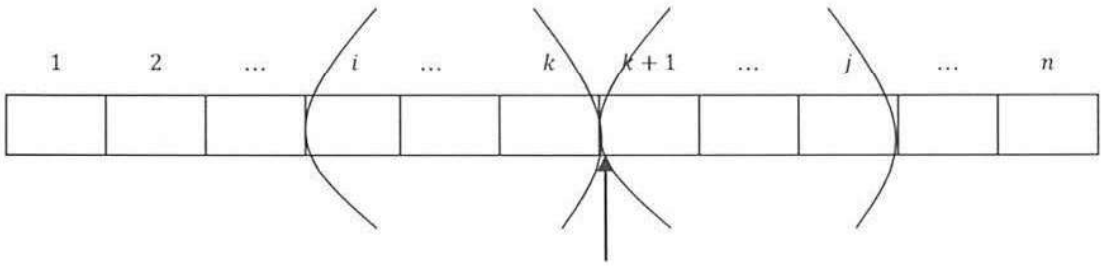
Now let us see how DP solves this problem. Let $T(i,j)$ represent the number of ways to parenthesize the sub expression with symbols $i \ldots j$ [symbols means only T and F and not the operators] with boolean operators so that it evaluates to *true*. Also, i and j take the values from 1 to n. For example, in the above case, $T(2,4) = 0$ because there is no way to parenthesize the expression *F and T xor F* to make it *true*.

Just for simplicity and similarity, let $F(i,j)$ represent the number of ways to parenthesize the sub expression with symbols $i \ldots j$ with boolean operators so that it evaluates to *false*. The base cases are $T(i,i)$ and $F(i,i)$.

Now we are going to compute $T(i,i+1)$ and $F(i,i+1)$ for all values of i. Similarly, $T(i,i+2)$ and $F(i,i+2)$ for all values of i and so on. Now let's generalize the solution.

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k)T(k+1,j), & for\ "and" \\ Total(i,k)Total(k+1,j) - F(i,k)F(k+1,j), & for\ "or" \\ T(i,k)F(k+1,j) + F(i,k)T(k+1,j), & for\ "xor" \end{cases}$$

Where, $Total(i,k) = T(i,k) + F(i,k)$.



What this above recursive formula says is, $T(i,j)$ indicates the number of ways to parenthesize the expression. Let us assume that we have some sub problems which are ending at k. Then the total number of ways to

parenthesize from i *to* j is the sum of counts of parenthesizing from i *to* k and from $k+1$ *to* j. To parenthesize between k and $k+1$ there are three ways: "*and*", "*or*" and "*xor*".

- If we use "*and*" between k and $k+1$, then the final expression becomes *true* only when both are *true*. If both are *true* then we can include them to get the final count.
- If we use "*or*", then if at least one of them is *true*, the result becomes *true*. Instead of including all three possibilities for "*or*", we are giving one alternative where we are subtracting the "false" cases from total possibilities.
- The same is the case with "*xor*". The conversation is as in the above two cases.

After finding all the values we have to select the value of k, which produces the maximum count, and for k there are i *to* $j-1$ possibilities.

How many subproblems are there? In the above formula, i can range from 1 *to* n, and j can range from 1 *to* n. So there are a total of n^2 subproblems, and also we are doing summation for all such values. So the time complexity is $O(n^3)$.

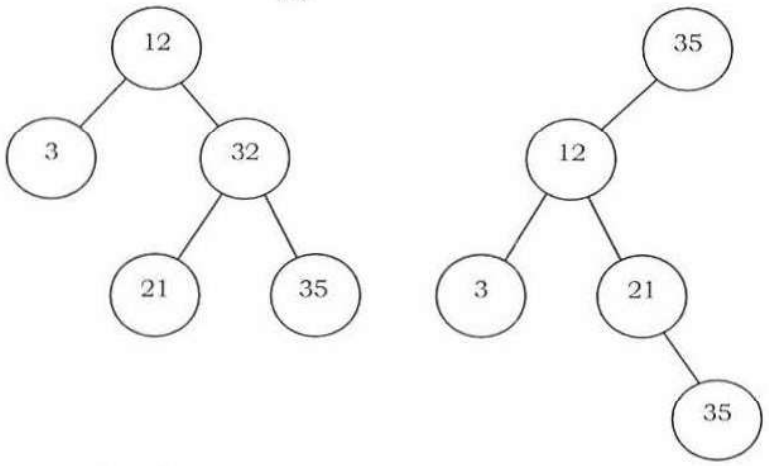
Problem-31 Optimal Binary Search Trees: Given a set of n (sorted) keys $A[1..n]$, build the best binary search tree for the elements of A. Also assume that each element is associated with *frequency* which indicates the number of times that a particular item is searched in the binary search trees. That means we need to construct a binary search tree so that the total search time will be reduced.

Solution: Before solving the problem let us understand the problem with an example. Let us assume that the given array is A = [3, 12, 21, 32, 35]. There are many ways to represent these elements, two of which are listed below.

Of the two, which representation is better? The search time for an element depends on the depth of the node. The average number of comparisons for the first tree is: $\frac{1+2+2+3+3}{5} = \frac{11}{5}$ and for the second tree, the average number of comparisons is: $\frac{1+2+3+3+4}{5} = \frac{13}{5}$. Of the two, the first tree gives better results.

If frequencies are not given and if we want to search all elements, then the above simple calculation is enough for deciding the best tree. If the frequencies are given, then the selection depends on the frequencies of the elements and also the depth of the elements. For simplicity let us assume that the given array is A and the corresponding frequencies are in array F. $F[i]$ indicates the frequency of i^{th} element $A[i]$. With this, the total search time S(root) of the tree with *root* can be defined as:

$$S(root) = \sum_{i=1}^{n} (depth(root, i) + 1) \times F[i])$$



In the above expression, $depth(root, i) + 1$ indicates the number of comparisons for searching the i^{th} element. Since we are trying to create a binary search tree, the left subtree elements are less than root element and the right subtree elements are greater than root element. If we separate the left subtree time and right subtree time, then the above expression can be written as:

$$S(root) = \sum_{i=1}^{r-1} (depth(root, i) + 1) \times F[i]) + \sum_{i=1}^{n} F[i] + \sum_{i=r+1}^{n} (depth(root, i) + 1) \times F[i])$$

Where r indicates the position of the root element in the array.

If we replace the left subtree and right subtree times with their corresponding recursive calls, then the expression becomes:

$$S(root) = S(root \rightarrow left) + S(root \rightarrow right) + + \sum_{i=1}^{n} F[i]$$

Binary Search Tree node declaration

Refer to *Trees* chapter.

Implementation:

```
class OptimalBinarySearchTree(BSTNode):  # For BSTNode, refer Trees chapter
    def __init__(self):
        super(OptimalBinarySearchTree, self).__init__()
        self.num_keys = 0
        self.keys = []
        self.probabilities = []
        self.dummyProbabilities = [1]
    def optimalBST(self):
        n = len(self.keys) + 1
        root_matrix = [[0 for i in xrange(n)] for j in xrange(n)]
        probabilitiesSumMatrix = [[0 for i in xrange(n)] for j in xrange(n)]
        expectedCostMatrix = [[99999 for i in xrange(n)] for j in xrange(n)]
        for i in xrange(1, n):
            probabilitiesSumMatrix[i][i-1] = self.dummyProbabilities[i - 1]
            expectedCostMatrix[i][i-1] = self.dummyProbabilities[i - 1]
        for l in xrange(1, n):
            for i in xrange(1, n - l):
                j = i + l - 1
                expectedCostMatrix[i][j] = 99999
                probabilitiesSumMatrix[i][j] = probabilitiesSumMatrix[i][j - 1] + \
                                    self.probabilities[j] + self.dummyProbabilities[j]
                for r in xrange(i, j + 1):
                    t = expectedCostMatrix[i][r - 1] + expectedCostMatrix[r+1][j] + probabilitiesSumMatrix[i][j]
                    if t < expectedCostMatrix[i][j]:
                        expectedCostMatrix[i][j] = t
                        root_matrix[i][j] = r
        return root_matrix
    def constructOptimalBST(self):
        root = self.optimalBST()
        n = self.num_keys
        r = root[1][n]
        value = self.keys[r]
        self.insert(value)
        self.constructOptimalSubtree(1, r-1, r, "left", root)
        self.constructOptimalSubtree(r+1, n, r, "right", root)
    def constructOptimalSubtree(self, i, j, r, direction, root):
        if i <= j:
            t = root[i][j]
            value = self.keys[t]
            self.insert(value)
            self.constructOptimalSubtree(i, t-1, t, "left", root)
            self.constructOptimalSubtree(t+1, j, t, "right", root)
```

Problem-32 **Edit Distance:** Given two strings A of length m and B of length n, transform A into B with a minimum number of operations of the following types: delete a character from A, insert a character into A, or change some character in A into a new character. The minimal number of such operations required to transform A into B is called the *edit distance* between A and B.

Solution:

Input: Two text strings A of length m and B of length n.

Goal: Convert string A into B with minimal conversions.

Before going to a solution, let us consider the possible operations for converting string A into B.

- If $m > n$, we need to remove some characters of A
- If $m == n$, we may need to convert some characters of A
- If $m < n$, we need to remove some characters from A

So the operations we need are the insertion of a character, the replacement of a character and the deletion of a character, and their corresponding cost codes are defined below.

Costs of operations:

Insertion of a character	c_i
Replacement of a character	c_r
Deletion of a character	c_d

Now let us concentrate on the recursive formulation of the problem. Let, $T(i,j)$ represents the minimum cost required to transform first i characters of A to first j characters of B. That means, $A[1 \ldots i]$ to $B[1 \ldots j]$.

$$T(i,j) = min \begin{cases} c_d + T(i-1,j) \\ T(i,j-1) + c_i \\ \begin{cases} T(i-1,j-1), & if\ A[i] == B[j] \\ T(i-1,j-1) + c_r & if\ A[i] \neq B[j] \end{cases} \end{cases}$$

Based on the above discussion we have the following cases.

- If we delete i^{th} character from A, then we have to convert remaining $i-1$ characters of A to j characters of B
- If we insert i^{th} character in A, then convert these i characters of A to $j-1$ characters of B
- If $A[i] == B[j]$, then we have to convert the remaining $i-1$ characters of A to $j-1$ characters of B
- If $A[i] \neq B[j]$, then we have to replace i^{th} character of A to j^{th} character of B and convert remaining $i-1$ characters of A to $j-1$ characters of B

After calculating all the possibilities we have to select the one which gives the lowest cost.

How many subproblems are there? In the above formula, i can range from 1 *to* m and j can range from 1 *to* n. This gives mn subproblems and each one takes O(1) and the time complexity is O(mn). Space Complexity: O(mn) where m is number of rows and n is number of columns in the given matrix.

```
def editDistance(A, B):
    m=len(A)+1
    n=len(B)+1
    table = {}
    for i in range(m): table[i,0]=i
    for j in range(n): table[0,j]=j
    for i in range(1, m):
        for j in range(1, n):
            cost = 0 if A[i-1] == B[j-1] else 1
            table[i,j] = min(table[i, j-1]+1, table[i-1, j]+1, table[i-1, j-1]+cost)
    return table[i,j]
print(editDistance("Helloworld", "HalloWorld"))
```

Problem-33 **All Pairs Shortest Path Problem: Floyd's Algorithm:** Given a weighted directed graph $G = (V,E)$, where $V = \{1,2,\ldots,n\}$. Find the shortest path between any pair of nodes in the graph. Assume the weights are represented in the matrix $C[V][V]$, where $C[i][j]$ indicates the weight (or cost) between the nodes i and j. Also, $C[i][j] = \infty$ or -1 if there is no path from node i to node j.

Solution: Let us try to find the DP solution (Floyd's algorithm) for this problem. The Floyd's algorithm for all pairs shortest path problem uses matrix $A[1..n][1..n]$ to compute the lengths of the shortest paths. Initially,

$$A[i,j] = C[i,j] \ if\ i \neq j$$
$$= 0 \quad if\ i = j$$

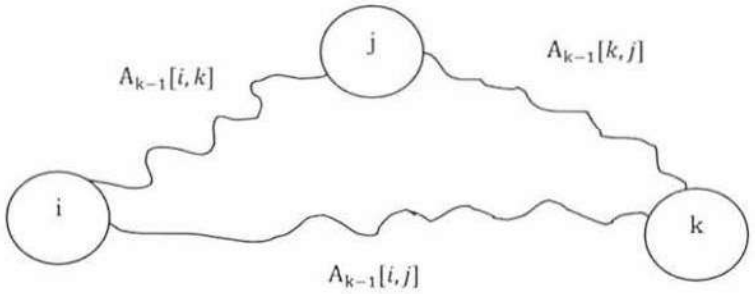
From the definition, $C[i,j] = \infty$ if there is no path from i to j. The algorithm makes n passes over A. Let $A_0, A_1, \ldots, A_n$ be the values of A on the n passes, with A_0 being the initial value.

Just after the $k-1^{th}$ iteration, $A_{k-1}[i,j]$ = smallest length of any path from vertex i to vertex j that does not pass through the vertices $\{k+1, k+2, \ldots . n\}$. That means, it passes through the vertices possibly through $\{1, 2, 3, \ldots, k-1\}$.

In each iteration, the value $A[i][j]$ is updated with minimum of $A_{k-1}[i,j]$ and $A_{k-1}[i,k] + A_{k-1}[k,j]$.

$$A[i,j] = \min \begin{cases} A_{k-1}[i,j] \\ A_{k-1}[i,k] + A_{k-1}[k,j] \end{cases}$$

The k^{th} pass explores whether the vertex k lies on an optimal path from i to j, for all i, j. The same is shown in the diagram below.




```
#script for Floyd Warshall Algorithm- All Pair Shortest Path
INF = 999999999
def printSolution(distGraph):
    string = "inf"
    nodes =distGraph.keys()
    for n in nodes:
        print "%10s"%(n),
    print " "
    for i in nodes:
        print"%s"%(i),
        for j in nodes:
            if distGraph[i][j] == INF:
                print "%10s"%(string),
            else:
                print "%10s"%(distGraph[i][j]),
        print" "
def floydWarshall(graph):
    nodes = graph.keys()
    distance = {}
    for n in nodes:
        distance[n] = {}
        for k in nodes:
            distance[n][k] = graph[n][k]
    for k in nodes:
        for i in nodes:
            for j in nodes:
                if distance[i][k] + distance[k][j] < distance[i][j]:
                    distance[i][j] = distance[i][k]+distance[k][j]
    printSolution(distance)
if __name__ == '__main__':
    graph = {'A':{'A':0,'B':6,'C':INF,'D':6,'E':7},
             'B':{'A':INF,'B':0,'C':5,'D':INF,'E':INF},
             'C':{'A':INF,'B':INF,'C':0,'D':9,'E':3},
             'D':{'A':INF,'B':INF,'C':9,'D':0,'E':7},
             'E':{'A':INF,'B':4,'C':INF,'D':INF,'E':0}
             }
```

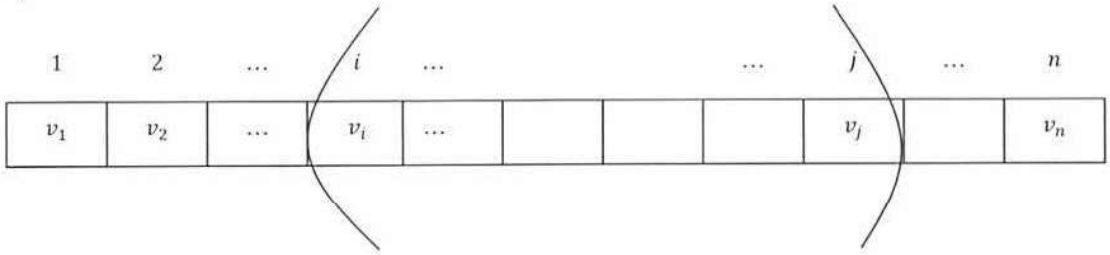
floydWarshall(graph)

Time Complexity: $O(n^3)$.

Problem-34 Optimal Strategy for a Game: Consider a row of n coins of values $v_1 \ldots v_n$, where n is even [since it's a two player game]. We play this game with the opponent. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Solution: Let us solve the problem using our DP technique. For each turn either we *or* our opponent selects the coin only from the ends of the row. Let us define the subproblems as:

$V(i,j)$: denotes the maximum possible value we can definitely win if it is our turn and the only coins remaining are $v_i \ldots v_j$.

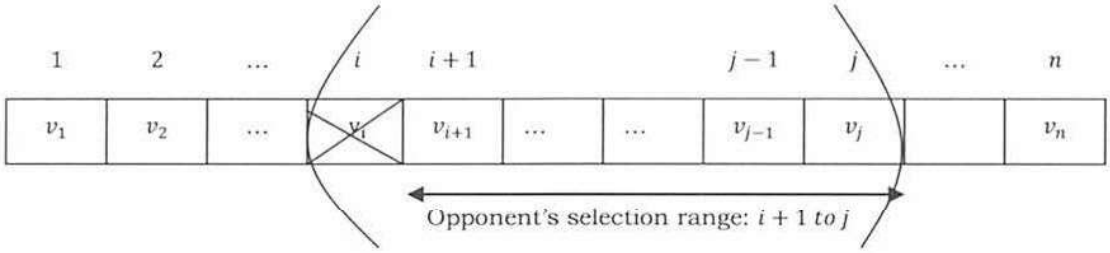


Base Cases: $V(i,i), V(i,i+1)$ for all values of i.
From these values, we can compute $V(i,i+2), V(i,i+3)$ and so on. Now let us define $V(i,j)$ for each sub problem as:

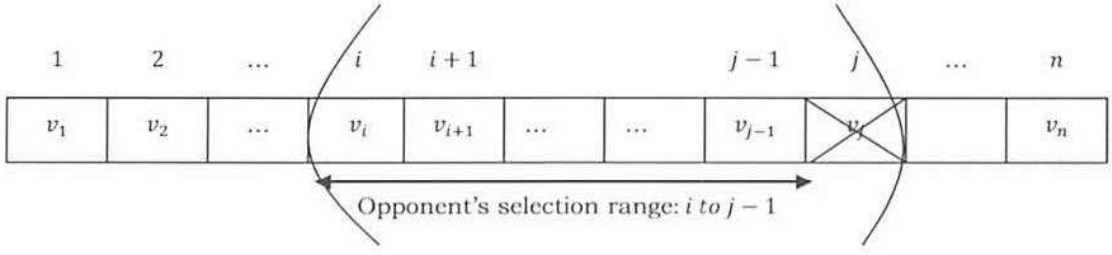
$$V(i,j) = Max\left\{Min\begin{Bmatrix}V(i+1,j-1)\\ V(i+2,j)\end{Bmatrix} + v_i, Min\begin{Bmatrix}V(i,j-2)\\ V(i+1,j-1)\end{Bmatrix} + v_j\right\}$$

In the recursive call we have to focus on ith coin to j^{th} coin $(v_i \ldots v_j)$. Since it is our turn to pick the coin, we have two possibilities: either we can pick v_i or v_j. The first term indicates the case if we select i^{th} coin (v_i) and the second term indicates the case if we select j^{th} coin (v_j). The outer *Max* indicates that we have to select the coin which gives maximum value. Now let us focus on the terms:

- Selecting i^{th} coin: If we select the i^{th} coin then the remaining range is from $i+1$ *to* j. Since we selected the i^{th} coin we get the value v_i for that. From the remaining range $i+1$ *to* j, the opponents can select either $i+1^{th}$ coin or j^{th} coin. But the opoonents selection should be minimized as much as possible [the *Min* term]. The same is described in the below figure.



- Selecting the j^{th} coin: Here also the argument is the same as above. If we select the j^{th} coin, then the remaining range is from i to j − 1. Since we selected the j^{th} coin we get the value v_j for that. From the remaining range i to j − 1, the opponent can select either the i^{th} coin or the $j-1^{th}$ coin. But the opponent's selection should be minimized as much as possible [the *Min* term].

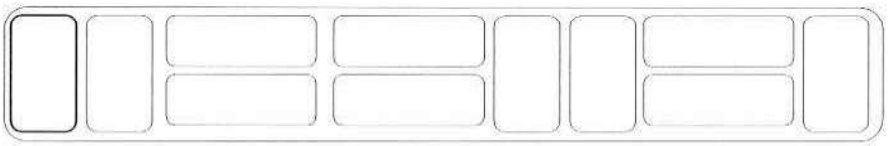


How many subproblems are there? In the above formula, i can range from 1 *to* n and j can range from 1 *to* n. There are a total of n^2 subproblems and each takes O(1) and the total time complexity is $O(n^2)$.

```
# row of n coins
coins = [1,2,3,4,5]
n = len(coins)
# each time it is our turn, take the max of the two available moves (but the minimum of
# the opponent's two     potential moves)
V = []
for i in range(n):
        V.append([0] * n)
for i in range(n):
        for j in range(n):
                if i == j:
                        V[i][j] = coins[i]
                elif j == i + 1:
                        V[i][j] = max(coins[i], coins[j])
                # only valid if i < j
                if (i + 2) <= j:
                        take_start = V[i + 2][j]
                else:
                        take_start = 0
                if (i + 1) <= (j - 1):
                        take_end = V[i + 1][j - 1]
                else:
                        take_start = 0
print V
```

Problem-35 **Tiling:** Assume that we use dominoes measuring 2×1 to tile an infinite strip of height 2. How many ways can one tile a $2 \times n$ strip of square cells with 1×2 dominoes?

Solution:



Solution: Notice that we can place tiles either vertically or horizontally. For placing vertical tiles, we need a gap of at least 2×2. For placing horizontal tiles, we need a gap of 2×1. In this manner, the problem is reduced to finding the number of ways to partition n using the numbers 1 and 2 with order considered relevant [1]. For example: $11 = 1 + 2 + 2 + 1 + 2 + 2 + 1$.

If we have to find such arrangements for 12, we can either place a 1 at the end or we can add 2 in the arrangements possible with 10. Similarly, let us say we have F_n possible arrangements for n. Then for $(n + 1)$, we can either place just 1 at the end *or* we can find possible arrangements for $(n - 1)$ and put a 2 at the end. Going by the above theory:

$$F_{n+1} = F_n + F_{n-1}$$

Let's verify the above theory for our original problem:

- In how many ways can we fill a 2×1 strip: $1 \rightarrow$ Only one vertical tile.
- In how many ways can we fill a 2×2 strip: $2 \rightarrow$ Either 2 horizontal or 2 vertical tiles.
- In how many ways can we fill a 2×3 strip: $3 \rightarrow$ Either put a vertical tile in the 2 solutions possible for a 2×2 strip, or put 2 horizontal tiles in the only solution possible for a 2×1 strip. $(2 + 1 = 3)$.
- Similarly, in how many ways can we fill a $2 \times n$ strip: Either put a vertical tile in the solutions possible for $2X(n - 1)$ strip or put 2 horizontal tiles in the solution possible for a $2 \times (n - 2)$ strip. $(F_{n-1} + F_{n-2})$.
- That's how we verified that our final solution is: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = 1$ and $F_2 = 2$.

Problem-36 **Longest Palindrome Subsequence:** A sequence is a palindrome if it reads the same whether we read it left to right or right to left. For example A,C,G,G,G,G,C,A. Given a sequence of length n, devise an algorithm to output the length of the longest palindrome subsequence. For example, the string $A,G,C,T,C,B,M,A,A,C,T,G,G,A,M$ has many palindromes as subsequences, for instance: A,G,T,C,M,C,T,G,A has length 9.

Solution: Let us use DP to solve this problem. If we look at the sub-string A[i,..,j] of the string A, then we can find a palindrome sequence of length at least 2 if A[i] == A[j]. If they are not the same, then we have to find the maximum length palindrome in subsequences A[i + 1,...,j] and A[i,...,j - 1].

Also, every character $A[i]$ is a palindrome of length 1. Therefore the base cases are given by $A[i,i] = 1$. Let us define the maximum length palindrome for the substring A[i,...,j] as L(i,j).

$$L(i,j) = \begin{cases} L(i+1, j-1) + 2, & \text{if } A[i] == A[j] \\ Max\{L(i+1,j), L(i,j-1)\}, & \text{otherwise} \end{cases}$$
$$L(i,i) = 1 \text{ for all } i = 1 \text{ to } n$$

```
def LongestPalindromeSubsequence(A):
    n = len(A)
    L =[[0 for x in range(n)] for x in range(n)]
    # palindromes with length 1
    for i  in range(0,n-1):
            L[i][i] = 1
    # palindromes with length up to j+1
    for k  in range(2,n+1):
            for i  in range(0,n-k+1):
                    j = i+k-1
                    if A[i] == A[j] and k ==2:
                            L[i][j] = 2
                    if A[i] == A[j]:
                            L[i][j] = 2 + L[i+1][j-1]
                    else:
                            L[i][j] = max( L[i+1][j] , L[i][j-1] )
    #print L
    return L[0][n-1]
print  LongestPalindromeSubsequence("Career Monk Publications")
```

Time Complexity: First 'for' loop takes $O(n)$ time while the second 'for' loop takes $O(n-k)$ which is also $O(n)$. Therefore, the total running time of the algorithm is given by $O(n^2)$.

Problem-37 Longest Palindrome Substring: Given a string A, we need to find the longest sub-string of A such that the reverse of it is exactly the same.

Solution: The basic difference between the longest palindrome substring and the longest palindrome subsequence is that, in the case of the longest palindrome substring, the output string should be the contiguous characters, which gives the maximum palindrome; and in the case of the longest palindrome subsequence, the output is the sequence of characters where the characters might not be contiguous but they should be in an increasing sequence with respect to their positions in the given string.

Brute-force solution exhaustively checks all $n(n+1)/2$ possible substrings of the given n-length string, tests each one if it's a palindrome, and keeps track of the longest one seen so far. This has worst-case complexity $O(n^3)$, but we can easily do better by realizing that a palindrome is centered on either a letter (for odd-length palindromes) or a space between letters (for even-length palindromes). Therefore we can examine all $n+1$ possible centers and find the longest palindrome for that center, keeping track of the overall longest palindrome. This has worst-case complexity $O(n^2)$.

Let us use DP to solve this problem. It is worth noting that there are no more than $O(n^2)$ substrings in a string of length n (while there are exactly 2^n subsequences). Therefore, we could scan each substring, check for a palindrome, and update the length of the longest palindrome substring discovered so far. Since the palindrome test takes time linear in the length of the substring, this idea takes $O(n^3)$ algorithm. We can use DP to improve this. For $1 \le i \le j \le n$, define

$$L(i,j) = \begin{cases} 1, & \text{if } A[i] \ldots A[j] \text{ is a palindrome substring,} \\ 0, & \text{otherwise} \end{cases}$$
$$L[i,i] = 1,$$
$$L[i,j] = L[i,i+1], \text{if } A[i] == A[i+1], \text{for } 1 \le i \le j \le n-1.$$

Also, for string of length at least 3,

$$L[i,j] = (L[i+1, j-1] \text{ and } A[i] = A[j]).$$

Note that in order to obtain a well-defined recurrence, we need to explicitly initialize two distinct diagonals of the boolean array $L[i,j]$, since the recurrence for entry $[i,j]$ uses the value $[i-1, j-1]$, which is two diagonals away from $[i,j]$ (that means, for a substring of length k, we need to know the status of a substring of length $k-2$).

```
def longestPalindromeSubstring(A):
    n = len(A)
    if n == 0: return ''
```

```
    L = {}
    for i in range(n): L[(i,i)] = True
    # k = j-i between 0 and n-1
    for k in range(n-1):
        for i in range(n):
            j = i+k
            if j >= n: continue
            if i+1 <= j-1:
                L[(i,j)] = L[(i+1,j-1)] and A[i] == A[j]
            else:
                L[(i,j)] = A[i] == A[j]
    start, end = max([k for k in L if L[k]],
                     key=lambda x:x[1]-x[0])
    return A[start:end+1]
print longestPalindromeSubstring('cabcbaabac')
print longestPalindromeSubstring('abbaaa')
print longestPalindromeSubstring('')
```

Time Complexity: First for loop takes $O(n)$ time while the second for loop takes $O(n - k)$ which is also $O(n)$. Therefore the total running time of the algorithm is given by $O(n^2)$.

Problem-38 Given two strings S and T, give an algorithm to find the number of times S appears in T. It's not compulsory that all characters of S should appear contiguous to T. For example, if $S = ab$ and $T = abadcb$ then the solution is 4, because ab is appearing 4 times in $abadcb$.

Solution:

Input: Given two strings $S[1..m]$ and $T[1...m]$.

Goal: Count the number of times that S appears in T.

Assume $L(i, j)$ represents the count of how many times i characters of S are appearing in j characters of T.

$$L(i,j) = Max \begin{cases} 0, & if\ j = 0 \\ 1, & if\ i = 0 \\ L(i-1, j-1) + L(i, j-1), & if\ S[i] == T[j] \\ L(i-1, j), & if\ S[i] \neq T[j] \end{cases}$$

If we concentrate on the components of the above recursive formula,

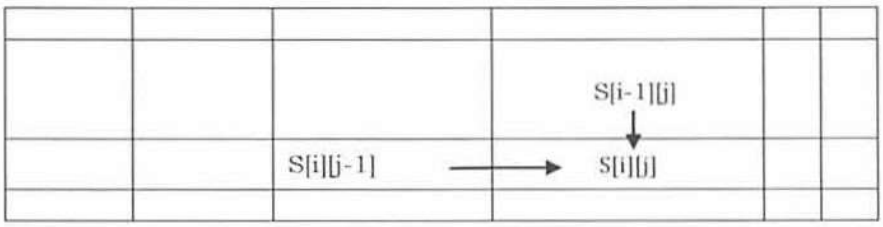
- If $j = 0$, then since T is empty the count becomes 0.
- If $i = 0$, then we can treat empty string S also appearing in T and we can give the count as 1.
- If S[i] == T[j], it means i^{th} character of S and j^{th} character of T are the same. In this case we have to check the subproblems with $i - 1$ characters of S and $j - 1$ characters of T and also we have to count the result of i characters of S with $j - 1$ characters of T. This is because even all i characters of S might be appearing in $j - 1$ characters of T.
- If S[i] ≠ T[j], then we have to get the result of subproblem with $i - 1$ characters of S and j characters of T.

After computing all the values, we have to select the one which gives the maximum count.

How many subproblems are there? In the above formula, i can range from 1 *to* m and j can range from 1 *to* n. There are a total of mn subproblems and each one takes O(1). Time Complexity is $O(mn)$.
Space Complexity: $O(mn)$ where m is number of rows and n is number of columns in the given matrix.

Problem-39 Given a matrix with n rows and m columns ($n \times m$). In each cell there are a number of apples. We start from the upper-left corner of the matrix. We can go down or right one cell. Finally, we need to arrive at the bottom-right corner. Find the maximum number of apples that we can collect. When we pass through a cell, we collect all the apples left there.

Solution: Let us assume that the given matrix is $A[n][m]$. The first thing that must be observed is that there are at most 2 ways we can come to a cell - from the left (if it's not situated on the first column) and from the top (if it's not situated on the most upper row).



To find the best solution for that cell, we have to have already found the best solutions for all of the cells from which we can arrive to the current cell. From above, a recurrent relation can be easily obtained as:

$$S(i,j) = \left\{ Apples[i][j] + Max \begin{Bmatrix} S(i,j-1), & if\ j > 0 \\ S(i-1,j), & if\ i > 0 \end{Bmatrix} \right\}$$

$S(i,j)$ must be calculated by going first from left to right in each row and process the rows from top to bottom, or by going first from top to bottom in each column and process the columns from left to right.

```
def FindApplesCount(Apples, n, m):
    S =[[0 for x in range(m)] for x in range(n)]
    S[0][0] = Apples[0][0]
    for i in range(1, n):
        S[i][0] = Apples[i][0] + S[i-1][0]
    for j in range(1, m):
        S[0][j] = Apples[0][j] + S[0][j-1]
    for i in range(1, n):
        for j in range(1, m):
            r1 = S[i][j-1]
            r2 = S[i-1][j]
            if (r1 > r2):
                S[i][j] = Apples[i][j]+r1
            else:
                S[i][j] = Apples[i][j]+r2
    return S[n-1][m-1]
Apples = [ [5, 24], [15, 25], [27, 40], [50, 60] ]
print FindApplesCount(Apples, 4, 2)
```

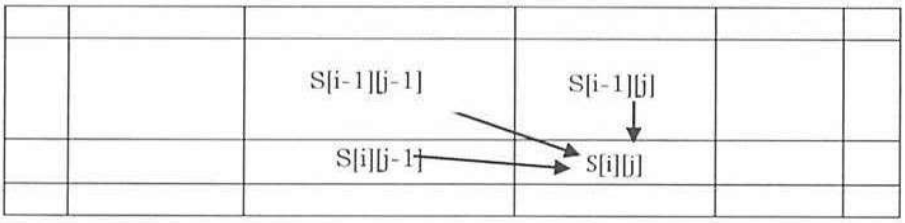
How many such subproblems are there? In the above formula, i can range from 1 *to* n and j can range from 1 *to* m. There are a total of nm subproblems and each one takes O(1). Time Complexity is O(nm). Space Complexity: O(nm), where m is number of rows and n is number of columns in the given matrix.

Problem-40 Similar to Problem-39, assume that we can go down, right one cell, or even in a diagonal direction. We need to arrive at the bottom-right corner. Give DP solution to find the maximum number of apples we can collect.

Solution: Yes. The discussion is very similar to Problem-39. Let us assume that the given matrix is A[n][m]. The first thing that must be observed is that there are at most 3 ways we can come to a cell - from the left, from the top (if it's not situated on the uppermost row) or from the top diagonal. To find the best solution for that cell, we have to have already found the best solutions for all of the cells from which we can arrive to the current cell. From above, a recurrent relation can be easily obtained:

$$S(i,j) = \left\{ A[i][j] + Max \begin{Bmatrix} S(i,j-1), & if\ j > 0 \\ S(i-1,j), & if\ i > 0 \\ S(i-1,j-1), & if\ i > 0\ and\ j > 0 \end{Bmatrix} \right\}$$

$S(i,j)$ must be calculated by going first from left to right in each row and process the rows from top to bottom, or by going first from top to bottom in each column and process the columns from left to right.



How many such subproblems are there? In the above formula, i can range from 1 to n and j can range from 1 *to* m. There are a total of nm subproblems and and each one takes O(1). Time Complexity is O(nm). Space Complexity: O(nm) where m is number of rows and n is number of columns in the given matrix.

Problem-41 **Maximum size square sub-matrix with all 1's:** Given a matrix with 0's and 1's, give an algorithm for finding the maximum size square sub-matrix with all 1s. For example, consider the binary matrix below.

0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

The maximum square sub-matrix with all set bits is

1	1	1
1	1	1
1	1	1

Solution: Let us try solving this problem using DP. Let the given binary matrix be $B[m][m]$. The idea of the algorithm is to construct a temporary matrix $L[\,][\,]$ in which each entry $L[i][j]$ represents size of the square sub-matrix with all 1's including $B[i][j]$ and $B[i][j]$ is the rightmost and bottom-most entry in the sub-matrix.

Algorithm:

1) Construct a sum matrix $L[m][n]$ for the given matrix $B[m][n]$.
   a. Copy first row and first columns as is from $B[\,][\,]$ to $L[\,][\,]$.
   b. For other entries, use the following expressions to construct $L[\,][\,]$
      if($B[i][j]$)
      $$L[i][j] = min(L[i][j-1], L[i-1][j], L[i-1][j-1]) + 1;$$
      else $L[i][j] = 0$;
2) Find the maximum entry in $L[m][n]$.
3) Using the value and coordinates of maximum entry in $L[i]$, print sub-matrix of $B[\,][\,]$.

```
def squareBlockWithAllOnesInMatrix(matrix, ZERO=0):
    nrows, ncols = len(matrix), (len(matrix[0]) if matrix else 0)
    if not (nrows and ncols): return 0 # empty matrix or rows
    Table = [[0]*ncols for _ in xrange(nrows)]
    for i in reversed(xrange(nrows)):     # for each row
        assert len(matrix[i]) == ncols # matrix must be rectangular
        for j in reversed(xrange(ncols)): # for each element in the row
            if matrix[i][j] != ZERO:
                Table[i][j] = (1 + min(
                    Table[i][j+1],  # east
                    Table[i+1][j],  # south
                    Table[i+1][j+1] # south-east
                    )) if i < (nrows - 1) and j < (ncols - 1) else 1 # edges
    return max(c for rows in Table for c in rows)
matrix=[[0, 1, 1, 0, 1], [1, 1, 0, 1, 0], [0, 1, 1, 1, 0], [1, 1, 1, 1, 0], [1, 1, 1, 1, 1], [0, 0, 0, 0, 0]]
print squareBlockWithAllOnesInMatrix(matrix)
```

How many subproblems are there? In the above formula, i can range from 1 to n and j can range from 1 to m. There are a total of nm subproblems and each one takes O(1). Time Complexity is $O(nm)$. Space Complexity is $O(nm)$, where n is number of rows and m is number of columns in the given matrix.

Problem-42 Maximum size sub-matrix with all 1's: Given a matrix with 0's and 1's, give an algorithm for finding the maximum size sub-matrix with all 1s. For example, consider the binary matrix below.

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

The maximum sub-matrix with all set bits is

1	1	1	1
1	1	1	1

Solution: If we draw a histogram of all 1's cells in the above rows for a particular row, then maximum all 1's sub-matrix ending in that row will be equal to maximum area rectangle in that histogram. Below is an example for 3rd row in the above discussed matrix [1]:

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

If we calculate this area for all the rows, maximum area will be our answer. We can extend our solution very easily to find start and end co-ordinates. For this, we need to generate an auxiliary matrix $S[][]$ where each element represents the number of 1s above and including it, up until the first 0. $S[][]$ for the above matrix will be as shown below:

1 1 0 0 1 0
0 2 1 1 2 1
1 3 2 2 3 0
0 0 3 3 0 0

Now we can simply call our maximum rectangle in histogram on every row in $S[][]$ and update the maximum area every time. Also we don't need any extra space for saving S. We can update original matrix (A) to S and after calculation, we can convert S back to A.

```
def maximumRectangleInMatrix(self, matrixInput):
    maxArea = 0
    rows = []
    columns = []
    for i in range(0,len(matrixInput)):
        rowTemp = []
        colTemp = []
        for j in range(0, len(matrixInput[0])):
            rowTemp.append(0)
            colTemp.append(0)
        rows.append(rowTemp)
        columns.append(colTemp)
    for i in range(len(matrixInput)-1,-1,-1):
        for j in range(len(matrixInput[0])-1,-1,-1):
            area = 0
            if matrixInput[i][j]=='1':
                if i==len(matrixInput)-1:
                    rows[i][j] = 1
                else:
                    rows[i][j] = rows[i+1][j] + 1
                if j == len(matrixInput[0])-1:
                    columns[i][j] = 1
                else:
                    columns[i][j] = columns[i][j+1]+1
                area = columns[i][j]
                minCol = columns[i][j]
            for k in range(1, rows[i][j]):
                if minCol > columns[i+k][j]:
                    minCol = columns[i+k][j]
                if (k+1)*minCol > area:
                    area = (k+1)*minCol
            if maxArea < area:
                maxArea = area
    return maxArea
```

Problem-43 **Maximum sum sub-matrix:** Given an $n \times n$ matrix M of positive and negative integers, give an algorithm to find the sub-matrix with the largest possible sum.

Solution: Let $Aux[r,c]$ represent the sum of rectangular subarray of M with one corner at entry $[1,1]$ and the other at $[r,c]$. Since there are n^2 such possibilities, we can compute them in $O(n^2)$ time. After computing all possible sums, the sum of any rectangular subarray of M can be computed in constant time. This gives an $O(n^4)$ algorithm: we simply guess the lower-left and the upper-right corner of the rectangular subarray and use the Aux table to compute its sum.

```
import sys
def preComputeMatrix(A,n):
    global Aux
    for i in range (0,n):
        for j in range (0,n):
            if(i==0 and j==0):
                Aux[i][j] = A[i][j]
            elif(i==0):
```

```
                        Aux[i][j] +=Aux[i][j-1] + A[i][j]
                elif(j==0):
                        Aux[i][j] +=Aux[i-1][j] + A[i][j]
                else:
                        Aux[i][j] +=Aux[i-1][j]+Aux[i][j-1]-Aux[i-1][j-1] + A[i][j]
def computeSum(A, i1, i2, j1, j2):
    if(i1==0 and j1==0):
            return Aux[i2][j2]
    elif(i1==0):
            return Aux[i2][j2] - Aux[i2][j1-1]
    elif(j1==0):
            return Aux[i2][j2] - Aux[i1-1][j2]
    else:
            return Aux[i2][j2] - Aux[i2][j1-1]- Aux[i1-1][j2] + Aux[i1-1][j1-1]
def getMaxMatrix(A,n):
    maxSum = -sys.maxint
    for row1 in range (0,n):
            for row2 in range (0,n):
                    for col1 in range (0,n):
                            for col2 in range (0,n):
                                    maxSum = max(maxSum,computeSum(A,row1,row2,col1,col2))
    return maxSum

A = [[-1, -2, -3, -4], [-5, -6, -7, -8 ],[-9, -10, -11, -12] ,[-13, -14, -15, -16]]
n = 4
Aux =[[0 for x in range(n)] for x in range(n)]
preComputeMatrix(A,n)
print getMaxMatrix(A,n)
```

Problem-44 Can we improve the complexity of Problem-43?

Solution: We can use Problem-7 solution with little variation, as we have seen that the maximum sum array of a $1-D$ array algorithm scans the array one entry at a time and keeps a running total of the entries. At any point, if this total becomes negative, then set it to 0. This algorithm is called *Kadane's* algorithm. We use this as an auxiliary function to solve a two-dimensional problem in the following way.

```
import sys
def preComputeMatrix(A,n):
    global Aux
    for i in range (0,n):
            for j in range (0,n):
                    if(i==0 and j==0):
                            Aux[i][j] = A[i][j]
                    elif(i==0):
                            Aux[i][j] +=Aux[i][j-1] + A[i][j]
                    elif(j==0):
                            Aux[i][j] +=Aux[i-1][j] + A[i][j]
                    else:
                            Aux[i][j] +=Aux[i-1][j]+Aux[i][j-1]-Aux[i-1][j-1] + A[i][j]
def computeSum(A, i1, i2, j1, j2):
    if(i1==0 and j1==0):
            return Aux[i2][j2]
    elif(i1==0):
            return Aux[i2][j2] - Aux[i2][j1-1]
    elif(j1==0):
            return Aux[i2][j2] - Aux[i1-1][j2]
    else:
            return Aux[i2][j2] - Aux[i2][j1-1]- Aux[i1-1][j2] + Aux[i1-1][j1-1]
def getSubmatSum( r1, c1, r2, c2) :
    if (r1 == 0 and c1 == 0):
            return Aux[r2][c2]
    if (r1 == 0):
            return Aux[r2][c2] - Aux[r2][c1 - 1]
    if (c1 == 0):
```

```
            return Aux[r2][c2] - Aux[r1 - 1][c2]
        return Aux[r2][c2] - Aux[r1 - 1][c2] - Aux[r2][c1 - 1] + Aux[r1 - 1][c1 - 1]

def getMaxMatrix(A,n):
    globalmax = 0
    for i in range (0,n):
            for j in range (i,n):
                    localmax = 0
                    for k in range (0,n):
                            localmax = max(localmax+getSubmatSum(i, k, j, k), 0)
                            globalmax = max(localmax, globalmax)
    return globalmax

A = [[-1, -2, 13, -4], [-5, -6, -7, -8 ],[-9, 10, -11, -12] ,[-13, -14, -15, -16]]
n = 4
Aux =[[0 for x in range(n)] for x in range(n)]
preComputeMatrix(A,n)
print getMaxMatrix(A,n)
```

Time Complexity: $O(n^3)$.

Problem-45 Given a number n, find the minimum number of squares required to sum a given number n.
Examples: $\min[1] = 1 = 1^2$, $\min[2] = 2 = 1^2 + 1^2$, $\min[4] = 1 = 2^2$, $\min[13] = 2 = 3^2 + 2^2$.

Solution: This problem can be reduced to a coin change problem. The denominations are 1 to $\sqrt{n}$. Now, we just need to make change for n with a minimum number of denominations.

Problem-46 **Finding Optimal Number of Jumps To Reach Last Element:** Given an array, start from the first element and reach the last by jumping. The jump length can be at most the value at the current position in the array. The optimum result is when you reach the goal in the minimum number of jumps. **Example**: Given array A = {2,3,1,1,4}. Possible ways to reach the end (index list) are:

- 0,2,3,4 (jump 2 to index 2, and then jump 1 to index 3, and then jump 1 to index 4)
- 0,1,4 (jump 1 to index 1, and then jump 3 to index 4)

Since second solution has only 2 jumps it is the optimum result.

Solution: This problem is a classic example of Dynamic Programming. Though we can solve this by brute-force, it would be complex. We can use the LIS problem approach for solving this. As soon as we traverse the array, we should find the minimum number of jumps for reaching that position (index) and update our result array. Once we reach the end, we have the optimum solution at last index in result array.

How can we find the optimum number of jumps for every position (index)? For first index, the optimum number of jumps will be zero. Please note that if value at first index is zero, we can't jump to any element and return infinite. For $n+1^{th}$ element, initialize result[$n+1$] as infinite. Then we should go through a loop from $0 \ldots n$, and at every index i, we should see if we are able to jump to $n+1$ from i or not. If possible, then see if total number of jumps (result[i] + 1) is less than result[$n+1$], then update result[$n+1$], else just continue to next index.

```
import sys
def minJumps(A):
    n = len(A)
    jumps= [0]*(n)
    if (n == 0 or A[0] == 0):
            return sys.maxint + 1
    jumps[0] = 0
    for i in range(1,n):
            jumps[i] = sys.maxint + 1
            for j in range(0,i):
                    if (i <= j + A[j] and jumps[j] != sys.maxint + 1):
                            jumps[i] = min(jumps[i], jumps[j] + 1)
                            break
    return jumps[n-1]
A = [1, 3, 6, 1, 0, 9]
print "Minimum number of jumps to reach end is ", minJumps(A)
A = [2,3,1,1,4]
print "Minimum number of jumps to reach end is ", minJumps(A)
```

Above code will return optimum number of jumps. To find the jump indexes as well, we can very easily modify the code as per requirement.

Time Complexity: Since we are running 2 loops here and iterating from 0 to i in every loop then total time takes will be $1 + 2 + 3 + 4 + \ldots + n - 1$. So time efficiency $O(n) = O(n * (n-1)/2) = O(n^2)$.
Space Complexity: $O(n)$ space for result array.

Problem-47 Explain what would happen if a dynamic programming algorithm is designed to solve a problem that does not have overlapping sub-problems.

Solution: It will be just a waste of memory, because the answers of sub-problems will never be used again. And the running time will be the same as using the Divide & Conquer algorithm.

Problem-48 Given a sequence of n positive numbers totaling to T, check whether there exists a subsequence totaling to X, where X is less than or equal to T.

Solution: Let's call the given Sequence S for convenience. Solving this problem, there are two approaches we could take. On the one hand, we could look through all the possible sub-sequences of S to see if any of them sum up to X. This approach, however, would take an exponential amount of work since there are 2^n possible sub-sequences in S. On the other hand, we could list all the sums between 0 and X and then try to find a sub-sequence for each one of them until we find one for X. This second approach turns out to be quite a lot faster: $O(n \times T)$. Here are the steps:

0. Create a boolean array called sum of size X+1: As you might guess, when we are done filling the array, all the sub-sums between 0 and X that can be calculated from S will be set to true and those that cannot be reached will be set to false. For example if S={2,4,7,9} then sum[5]=false while sum[13]=true since 4+9=13.
1. Initialize sum{} to false: Before any computation is performed, assume/pretend that each sub-sum is unreachable. We know that's not true, but for now let's be outrageous.
2. Set sum at index 0 to true:This truth is self-evident. By taking no elements from S, we end up with an empty sub-sequence. Therefore we can mark sum[0]=true, since the sum of nothing is zero.
3. To fill the rest of the table, we are going to use the following trick. Let S={2,4,7,9}. Then starting with 0, each time we find a positive sum, we will add an element from S to that sum to get a greater sum. For example, since sum[0]=true and 2 is in S, then sum[0+2] must also be true. Therefore, we set sum[0+2]=sum[2]=true. Then from sum[2]=true and element 4, we can say sum[2+4]=sum[6]=true, and so on.

Step 3 is known as the relaxation step. First we started with an absurd assumption that no sub-sequence of S can sum up to any number. Then as we find evidence to the contrary, we relax our assumption.

Alternative implementation: This alternative is easier to read, but it does not halt for small X. In the actual code, each for-loop checks for "not sum[X]" since that's really all we care about and should stop once we find it. Also this time complexity is $O(n \times T)$ and space complexity is O(T)

```
subSum = [False] * ( X + 1 )
sum[0] = True
for a in A:
  for i in range(sum(A), a-1,-1):  T = sum(A)
    if not sum[i] and sum[i - a]:
      sum[i] = True

def positiveSubsetSum( A, X ):
        # preliminary
        if X < 0 or X > sum( A ): # T = sum(A)
                return False

        # algorithm
        subSum = [False] * ( X + 1 )
        subSum[0] = True
        p = 0
        while not subSum[X] and p < len( A ):
                a = A[p]
                q = X
                while not subSum[X] and q >= a:
                        if not subSum[q] and subSum[q - a]:
                                subSum[q] = True
                        q -= 1
                p += 1
        return subSum[X]
```

Problem-49 You are climbing a stair case. It takes n steps to reach to the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution: The easiest idea is a Fibonacci number. fib(n) = fib(n-1) + fib(n-2). The n^{th}stairs is from either $n-1^{th}$ the stair or the $n-2^{th}$ stair. However recursive is time-consuming. We know that recursion can be written in loop, the trick here is not construct a length of n array, only three element array is enough.

Problem-50 Christmas is approaching. You're helping Santa Claus to distribute gifts to children. For ease of delivery, you are asked to divide n gifts into two groups such that the weight difference of these two groups is minimized. The weight of each gift is a positive integer. Please design an algorithm to find an optimal division minimizing the value difference. The algorithm should find the minimal weight difference as well as the groupings in O(nS) time, where S is the total weight of these n gifts. Briefly justify the correctness of your algorithm.

Solution: This problem can be converted into making one set as close to $\frac{S}{2}$ as possible. We consider an equivalent problem of making one set as close to W=$\left\lfloor\frac{S}{2}\right\rfloor$ as possible. Define FD(i,w) to be the minimal gap between the weight of the bag and W when using the first i gifts only. WLOG, we can assume the weight of the bag is always less than or equal to W. Then fill the DP table for 0≤i≤ n and 0≤ w ≤W in which F(0, w) = W for all w, and

$$FD(i,w) = min\{FD(i-1,w-w_i)-w_i, FD(i-1,w)\}\ if\ \{FD(i-1,w-w_i) \geq w_i$$
$$= FD(i-1,w)\ \text{otherwise}$$

This takes O(nS) time. $FD(n,W)$ is the minimum gap. Finally, to reconstruct the answer, we backtrack from (n,W). During backtracking, if $FD(i,j) = FD(i-1,j)$ then i is not selected in the bag and we move to F($i-1,j$). Otherwise, i is selected and we move to F($i-1,j-w_i$).