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?

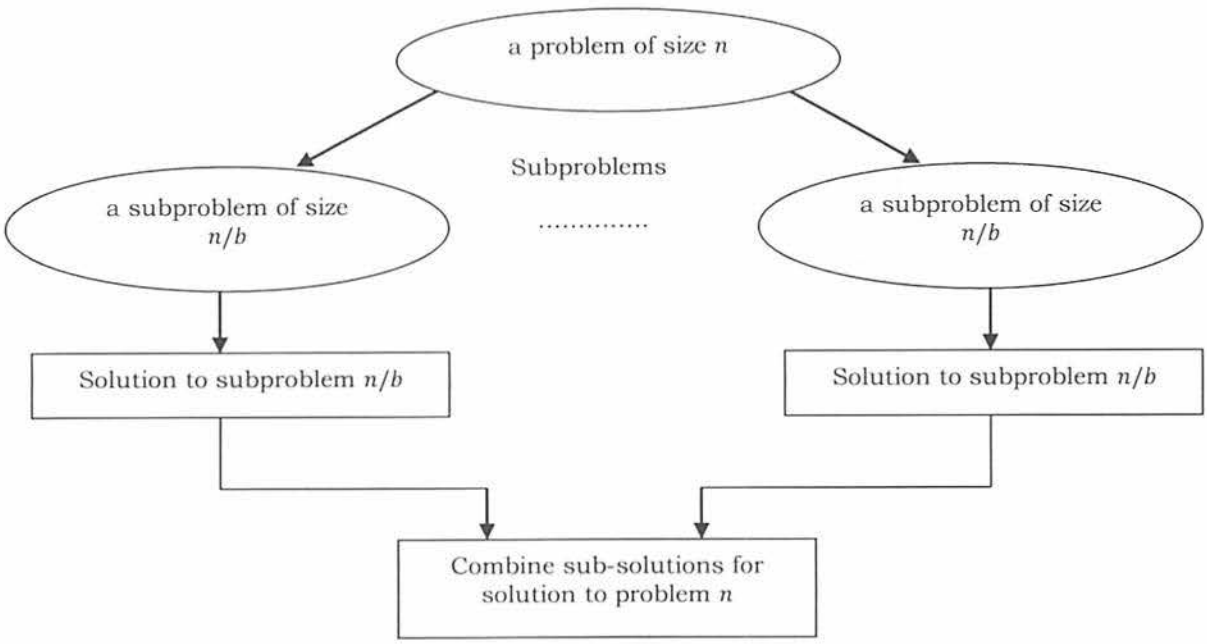
It's not possible to solve all the problems with the Divide & Conquer technique. As per the definition of D & C, the recursion solves the subproblems which are of the same type. For all problems it is not possible to find the subproblems which are the same size and *D & C* is not a choice for all problems.

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 $O\left(n^0 2^{\frac{n}{1}}\right) = O(2^n)$. (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 $A(1) \ldots A(n)$, give an algorithm for finding a contiguous subsequence $A(i) \ldots A(j)$ for which the sum of elements in the subsequence is maximum. **Example**: {-2, **11, -4, 13**, -5, 2} → 20 and {1, -3, **4, -2, -1, 6**} → 7.

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

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

We begin by looking at case 3. To avoid the nested loop that results from considering all $n/2$ starting points and $n/2$ ending points independently, replace two nested loops 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 $O(n^2)$.

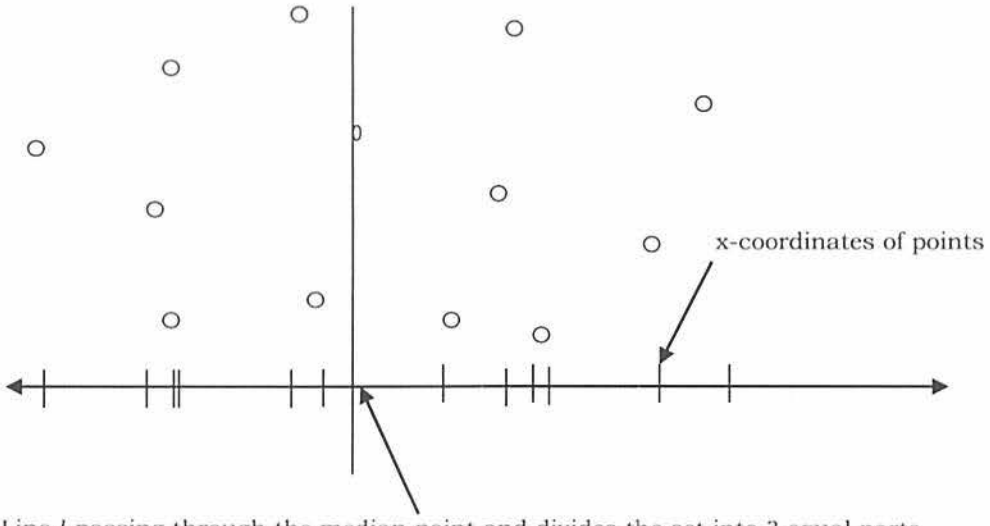
```
from math import sqrt, pow
def distance(a, b):
  return sqrt(pow(a[0] - b[0],2) + pow(a[1] - b[1],2))

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.

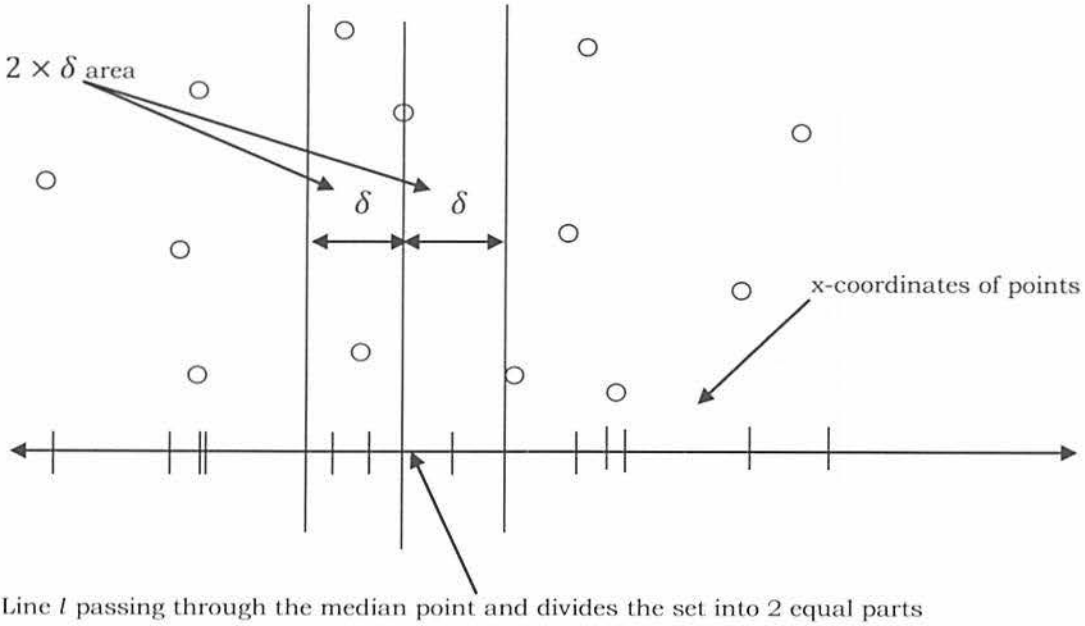
Algorithm:

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.



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

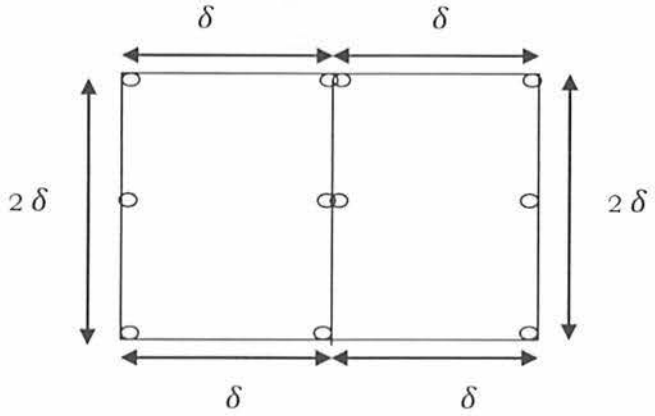
Combining the results in linear time



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

Let $\delta = min(L, R)$, where L is the solution to first 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.

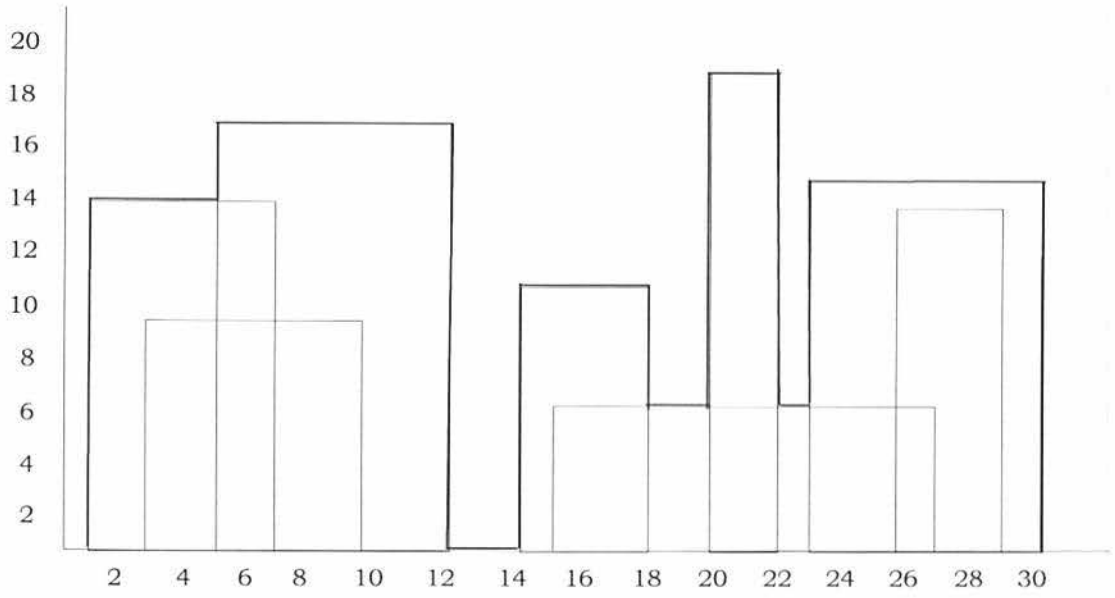
```
def powerBruteForce(k, n):
    """linear power algorithm"""
    x = k;
    for i in range(1, n):
        x *= k
    return x
def power(k, n):
    if n == 0: return 1
    x = power(k, math.floor(n/2))
    if n % 2 == 0: return pow(x, 2)
    else: return k * pow(x, 2)
```

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.

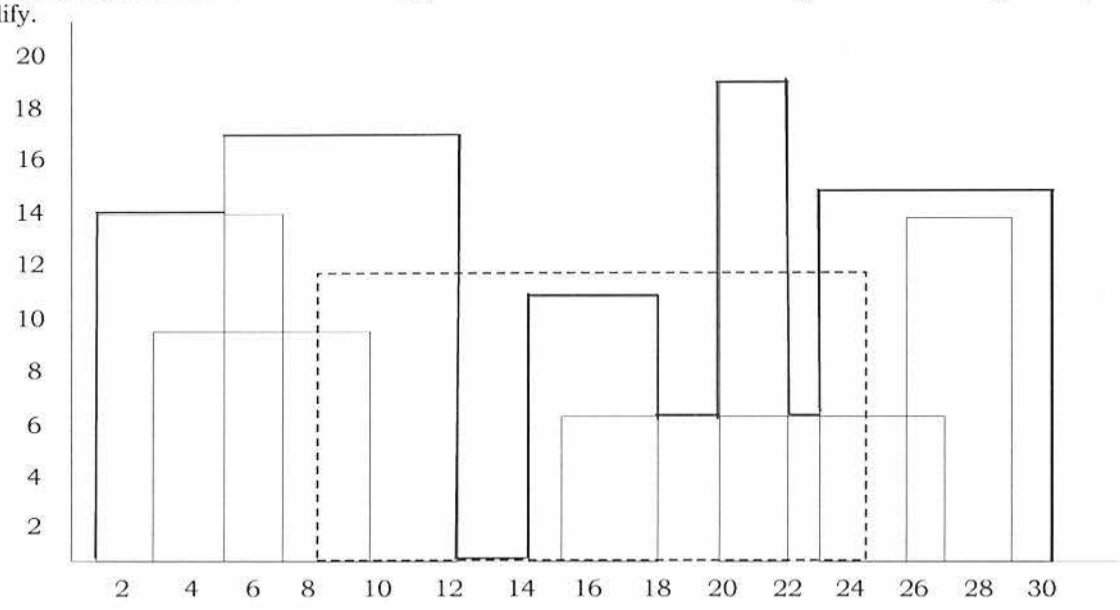
```
def SkyLineBruteForce():
    auxHeights = [0]*1000
    rightMostBuildingRi=0
    p = raw_input("Enter three values:  ") # raw_input() function
    inputValues = p.split()
    inputCount = len(inputValues)
    while inputCount==3:
        left = int(inputValues[0])
        h = int(inputValues[1])
        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*).