**Divide and Conquer**

Many algorithms are recursive in nature to solve a given problem recursively dealing with sub-problems.

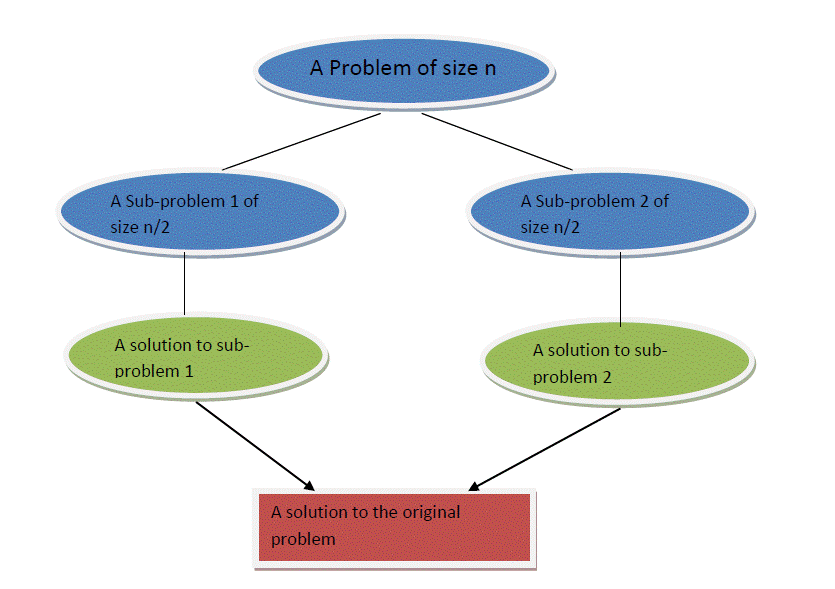
In the divide and conquer approach, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally, the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts −

**• Divide** the problem into a number of sub-problems that are smaller instances of the same problem.

**• Conquer** the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.

**• Combine** the solutions to the sub-problems into the solution for the original problem.



***CONTROL ABSTRACTION FOR DIVIDE AND CONQUER ALGORITHM***

Algorithm D and C (P)

**{**

**if small(P)**

**then return S(P)**

**else**

**{**

**divide P into smaller instances P1 ,P2 .....Pk**

**Apply D and C to each subproblem**

**Return combine (D and C(P1)+ D and C(P2)+.......+D and C(Pk))**

**} }**

Let a recurrence relation is expressed as :

T(n)= ϴ (1), if n<=C

aT(n/b) + D(n)+ C(n), otherwise

then n=input size

a=no. Of sub-problems

n/b= input size of the sub-problems

Following are some problems, which are solved using the divide and conquer approach:

* Finding the maximum and minimum of a sequence of numbers
* Strassen’s matrix multiplication
* Merge sort
* Quick Sort
* Binary search
* Convex Hull

**Pros and cons of the Divide and Conquer Approach**

* The divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.
* In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where the function state needs to be stored.

**AlgorithmsforFindMinandMax**

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

To find the maximum and minimum numbers in a given array of numbers[] of size n, the following algorithm can be used. First, we are representing the ***naive method*** and then we will present the ***divide and conquer approach.***

***Naïve Method***

The Naïve method is a basic method to solve any problem. In this method, the maximum and minimum numbers can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

**Algorithm:** Max-Min-Element (numbers[ ])

max := numbers[1]

min := numbers[1]

for i = 2 to n do

if numbers[i] > max then

max := numbers[i]

if numbers[i] < min then

min := numbers[i]

return (max, min)

**Analysis**

The number of comparisons in the *Naive method* is 2n - 2.

The number of comparisons can be reduced using the divide-and-conquer approach. Following is the technique.

***Divide and Conquer Approach***

In this approach, the array is divided into two halves. Then using a recursive approach maximum and minimum numbers in each half are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1, where y is greater than or equal to x.

Max−Min(x,y) will return the maximum and minimum values of an array of numbers[x...y].

**Algorithm:** Max - Min(x, y)

if x – y ≤ 1 then

return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))

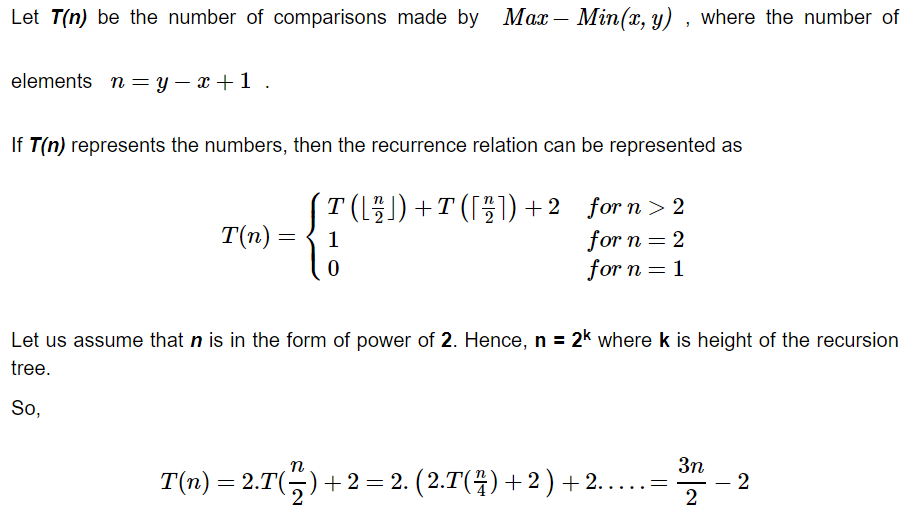
else

(max1, min1):=maxmin(x, ⌊((x + y)/2)⌋)

(max2, min2):=maxmin(⌊((x + y)/2) + 1)⌋,y)

return (max(max1, max2), min(min1, min2))

**Analysis:**

****

Compared to the Naïve method, in the divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n).**

**Sorting: QuickSort**

**Quick sorting** is a very popular sorting method. The name comes from the fact that ingeneral, quick sort can sort a list of data elements significantly faster than any of the common sorting algorithms. This algorithm is based on the fact that it is faster and easier to sort two small lists than one larger one. The basic strategy of quick sort is to divide and conquer. Quick sort is also known as ***partition exchange sort.***

The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore, reduces unnecessary swaps and moves an item a great distance in one move.

**Description of quicksort**

Quicksort is based on the ***three-step process*** of divide and conquer.

• To sort the subarrayA[p . . r ]:

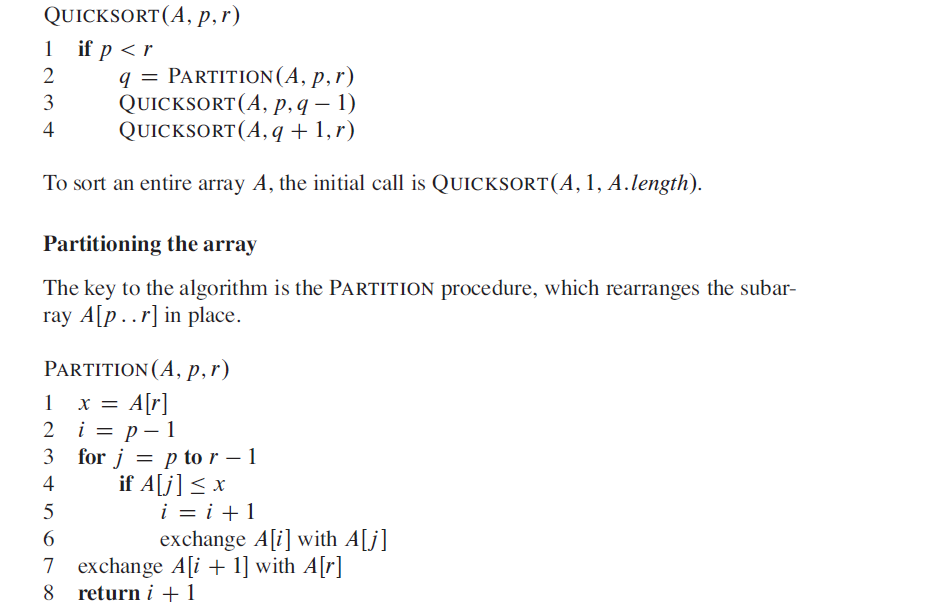
***Divide:*** Partition A[p . . r ], into two (possibly empty) subarraysA[p . . q − 1] and A[q + 1 . . r ], such that each element in the first subarrayA[p . . q − 1] is ≤ A[q] and

A[q] is ≤ each element in the second subarrayA[q + 1 . . r ].

***Conquer:*** Sort the two subarrays by recursive calls to QUICKSORT.

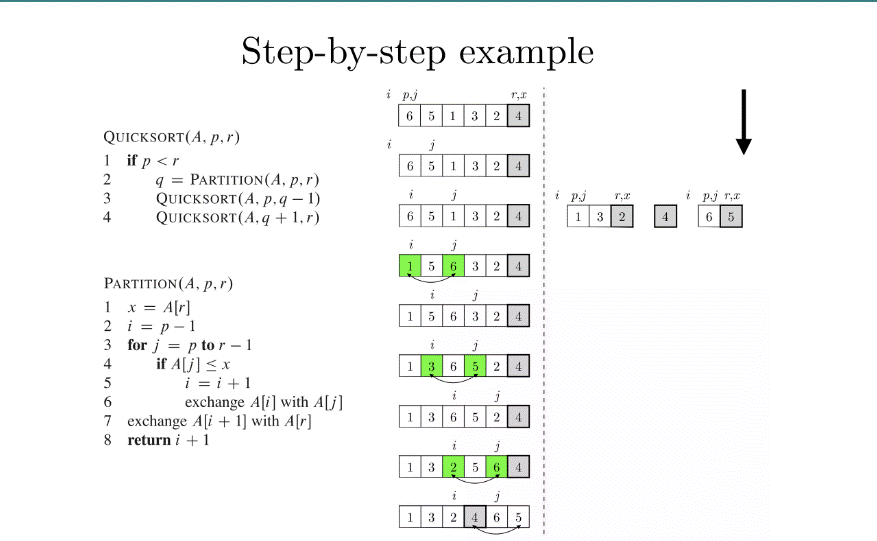
***Combine:*** No work is needed to combine the subarrays because they are sorted in place.

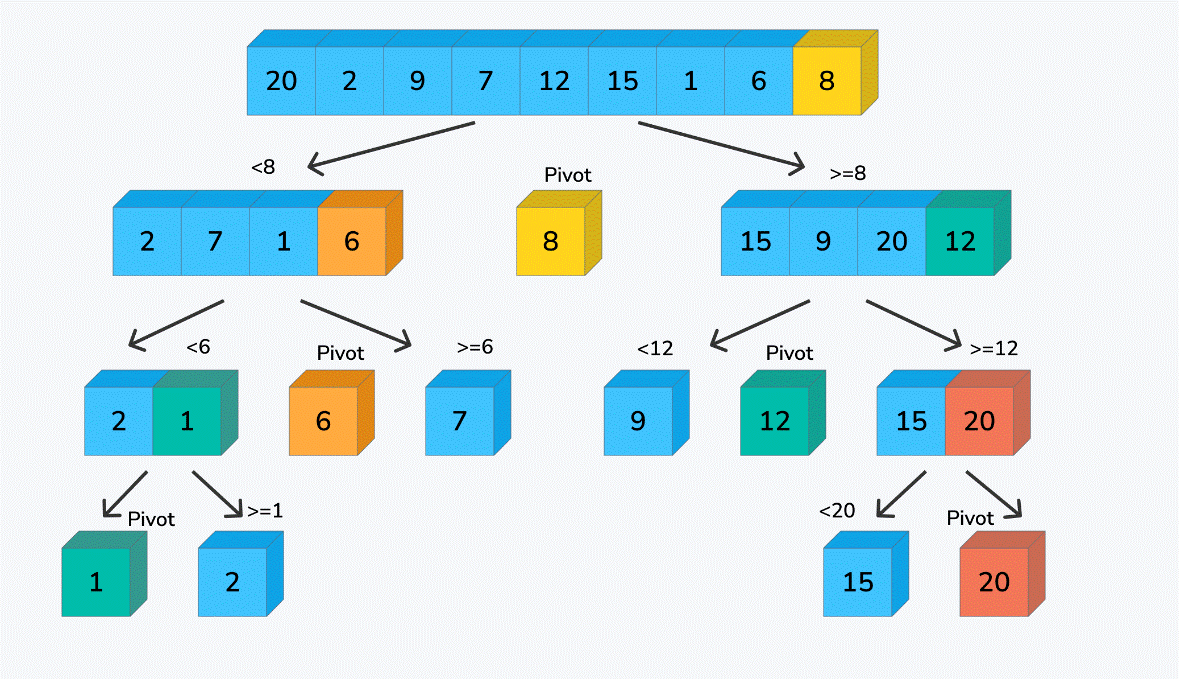
Perform the divide step by a procedure PARTITION, which returns the index q that marks the positionseparating the subarrays.

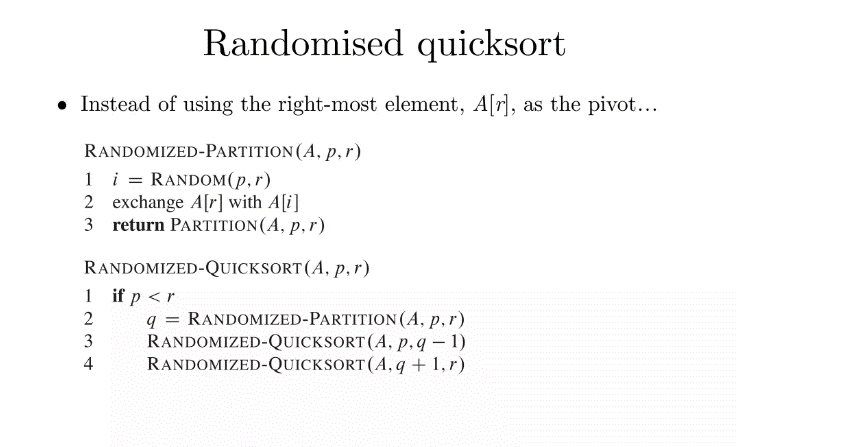


* PARTITION always selects the last element A[r ] in the subarrayA[p . . r ] as the pivot the

element around which to partition.







**Performance of Quicksort**

The running time of Quicksort depends on the partitioning of the subarrays:

• If the subarrays are balanced, then Quicksort can run as fast as mergesort.

• If they are unbalanced, then Quicksort can run as slowly as insertion sort.

***Worst case:***

• Occurs when the subarrays are completely unbalanced.

• Have 0 elements in one subarray and n − 1 elements in the other subarray.

• Get the recurrence

**T (n) = T (n − 1) + T (0) + Θ (n)**

**= T (n − 1) + Θ (n)**

**= O (n2).**

• Same running time as ***insertion sort.***

• In fact, the worst-case running time occurs when Quicksort takes a sorted array as input, but

insertion sort runs in O(n) time in this case.

***Best case:***

• Occurs when the subarrays are completely balanced every time.

• Each subarray has ≤ n/2 elements.

• Get the recurrence

**T (n) = 2T (n/2) + Θ (n) = O(n logn).**

**Balanced partitioning**

• QuickPort’s average running time is much closer to the best case than to the worst case.

• Imagine that PARTITION always produces a 9-to-1 split.

• Get the recurrence

T (n) ≤ T (9n/10) + T (n/10) + \_ (n) = O (n lgn).

• Intuition: look at the recursion tree.

• It’s like the one for T (n) = T (n/3) + T (2n/3) + O (n).

• Except that here the constants are different; we get log10 n full levels and log10/9 n levels thatare nonempty.

• As long as it’s a constant, the base of the log doesn’t matter in asymptotic notation.

• Any split of constant proportionality will yield a recursion tree of depth O (logn).

**Sorting: HeapSort**

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Heap sort is a comparison-based sorting technique based on the Binary Heap data structure.

Generally, **Heaps can be of two types:**

***Max-Heap:*** In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

***Min-Heap:***In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

**Advantages of heapsort:**

***Efficiency*** – The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.

***Memory Usage*** – Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work

***Simplicity*** – It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

**Operation/Methods on Heap**

***find*** - find an item in a heap.

***insert*** - add an item in a heap ensuring the heap property is maintained min-heap and max-heap property.

***Delete*** - remove an item from a heap.

***Extract*** - return the value of an item and then delete it from the heap.

***replace***- extract or pop the root and insert or push a new item in a heap ensuring the heap property has maintained min-heap and max-heap properties.

***size*** - returns the size of the heap.

***is-empty*** - returns 'true' if the heap is empty or 'false' if it has value.

***merge*** - joining or union of two heaps, all the values from both heaps are included but the original heaps are preserved.

***meld***- joining of two heaps where the values from both heaps are included but the original heaps are destroyed

**Heap Tree Representation:**

****

PARENT (i) => return [ i /2]

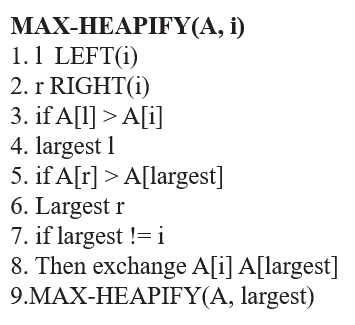
LEFT (i) => return 2i

RIGHT (i) => return 2i+ 1

**What is meant by Heapify?**

Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the first index of the non-leaf node whose index is given by n/2 – 1. **Heapify uses *recursion.***

*Maintaining the heap property:*





The running time of MAX-HEAPIFY by the recurrence can be described as

**T (n) < = T (2n/3) + O (1)**

The solution to this recurrence is T(n)=**O(log n)**

**Build Heap Tree:**

**Build-Max-Heap(A)**

1. for i[n/2] to 1

2. do

**MAX-HEAPIFY(A,i)**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **4** | **1** | **3** | **2** | **1** | **9** | **1** | **1** | **8** | **7** |

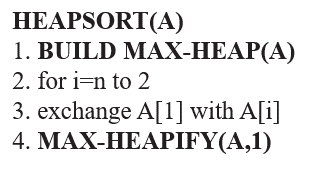
****





The total cost of BUILD-MAX-HEAP as being bounded is T(n)=**O(n)**

**HEAPSORT Algorithm:**









***Complexity for Heap Sort:***

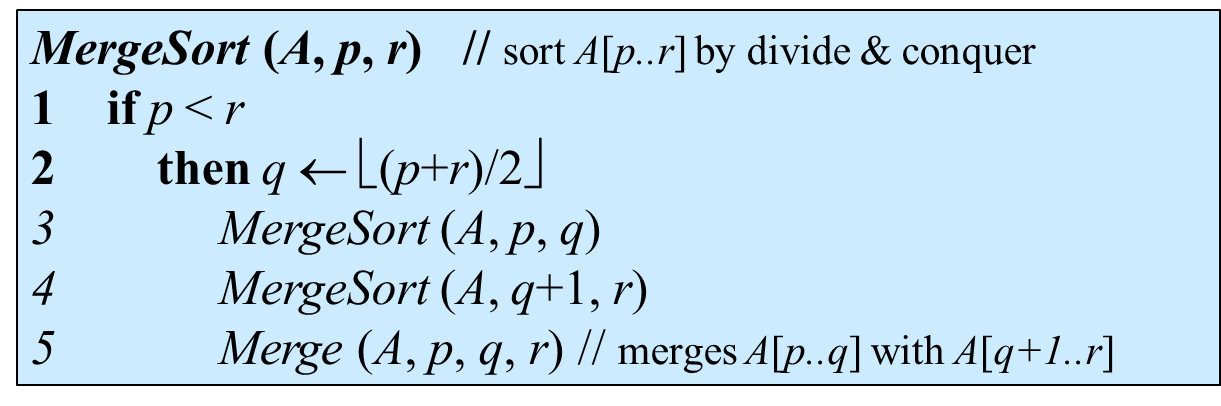
The **HEAPSORT procedure** takes time **O(n log n)** since the call to **BUILD-MAX-HEAP** takes time **O(n)** and each of the n - 1 calls to **MAX-HEAPIFY** takes time **O(log n).**

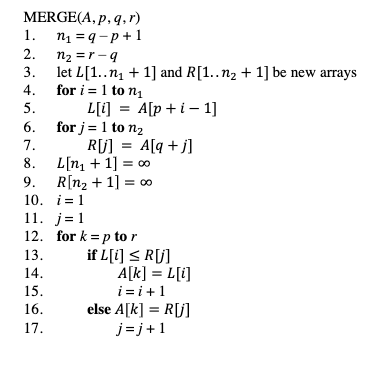
**Sorting: MergeSort**

**Principle:** The given list is divided into two roughly equal parts called the left and the right subfiles. These subfiles are sorted using the algorithm recursively and then the two subfiles are merged together to obtain the sorted file.

**There are 3 Phases in the Merge Sort Algorithm –**

1. Division Phase – Divide the array(list) into 2 halves by finding the midpoint of the array(list).
   1. Midpoint (m) = (left + right)/ 2
   2. *Here left is the starting index & right is the last index of the array(list)*
2. Recursion Phase –
   1. Call Merge Sort on the left sub-array (sub-list)
   2. Call Merge Sort on the right sub-array (sub-list)
3. Merge Phase –
   1. Call the merge function to merge the divided sub-arrays back to the original array.
   2. Perform sorting of these smaller subarrays before merging them back.





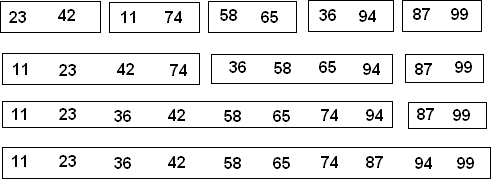
**LetL  low, Mmid, H  high**

### i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

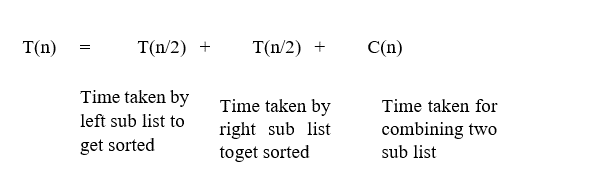
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **42** | **23** | **74** | **11** | **65** | **58** | **94** | **36** | **99** | **87** |

U M H

In each pass, the mid value is calculated and based on that the list is split into two. This s done recursively and at last N number of lists each having only one element is produced as shown

Now merging operation is called on the first two lists to produce a single sorted list, then the same thing is done on the next two lists, and so on. Finally, a single sorted list is obtained.

***Analysis:***

****

Let the recurrence relation for merge sort is

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

T(n)= 2T(n/2) +C(n) T (1) =0

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

Apply the Master theorem,

We will get, a=2, b=2, d=1 As per master theorem, a=bd

T(n)=Ɵ (ndlog2𝑛)

When d=1 T(n)=Ɵ (nlog2𝑛)

The time complexity for merge sort is Ɵ (nlog2𝑛)

**Searching: Binary Search**

The binary search method is very fast and efficient. This method requires that the list of elements bein sorted order. Binary search cannot be applied to an unsorted list.

**Principle:** The data item to be searched is compared with the approximate middle entry of the list. If it matches the middle entry, then the position will be displayed.

If the data item to be searched is lesser than the middle entry, then it is compared with the middle entry of the first half of the list, and the procedure is repeated on the first half until the required item is found.

If the data item is greater than the middle entry, then it is compared with the middle entry of the second half of the list, and the procedure is repeated on the second half until the required item is found. This process continues until the desired number is found or the search interval becomes empty.

* The binary search method is more efficient than the linear search method.
* The array must be sorted for binary search, which is not necessary for linear search.
* Binary search is a search strategy based on **divide and conquer.**
* The method divides the list into two halves at each step and checks whether the element to be searched is on the top or lower side of the array.
* If the element is located in the center, the algorithm returns.

Let us assign the minimum and maximum index of the array to variables low and high, respectively. The middle index is computed as (low + high)/2.In every iteration, the algorithm compares the middle element of the array with a key to be searched. The initial range of search is A[0] to A[n – 1]. When the key is compared with A [mid], there are three possibilities :

* The array is already sorted, so if key < A[mid] then the key cannot be present in the bottom half of the array. The search space of the problem will be reduced by setting the high index to (mid – 1). The new search range would be A[low] to A[mid – 1].
* If key > A[mid] then the key cannot be present in the upper half of the array. The search space of the problem will be reduced by moving the low index to (mid + 1). The new search range would be A[mid + 1] to A[high].
* If key = A[mid], the search is successful, and the algorithm halts.

This process is repeated till the index low is less than the high or the element is found.

**Algorithm BINARY\_SEARCH (A, Key)**

// Description: Perform a binary search on array A

// Input: Sorted array A of size n and Key to be searched

// Output: Success / Failure

low ← 1

high ← n

while low < high do

mid ← (low + high) / 2

if A[mid] == key then

return mid

else if A[mid] < key then

low ← mid + 1

else

high ← mid – 1

end

end

return 0

Binary search reduces search space by half in every iteration. In a linear search, the search space was reduced by one only.

If there are n elements in an array, binary search, and linear search have to search among (n / 2) and (n – 1) elements respectively in the second iteration.

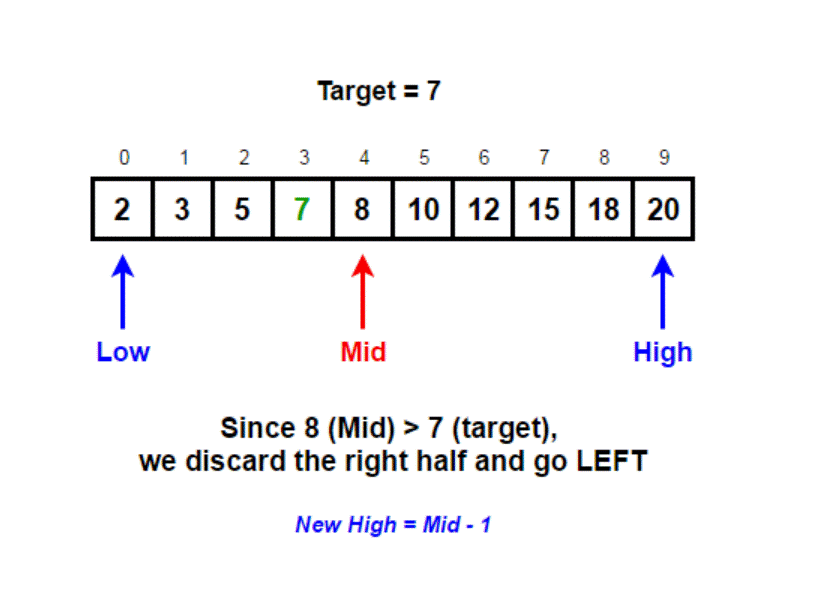
In the third iteration, the binary search has to scan only (n / 4) elements, whereas the linear search has to scan (n – 2) elements.

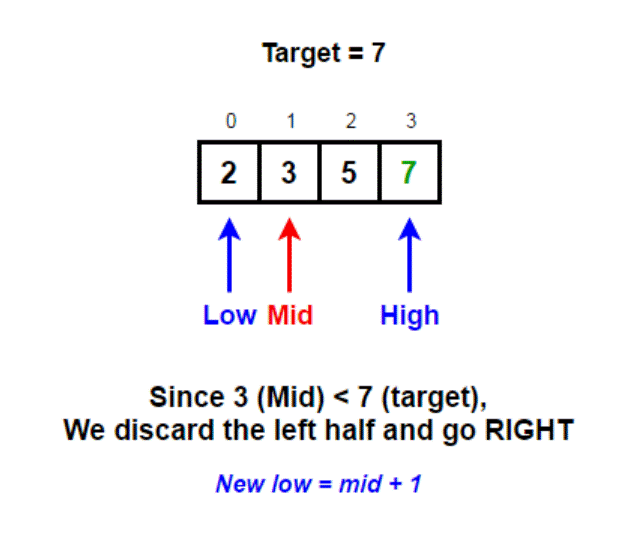
This shows that a binary search would hit the bottom very quickly.

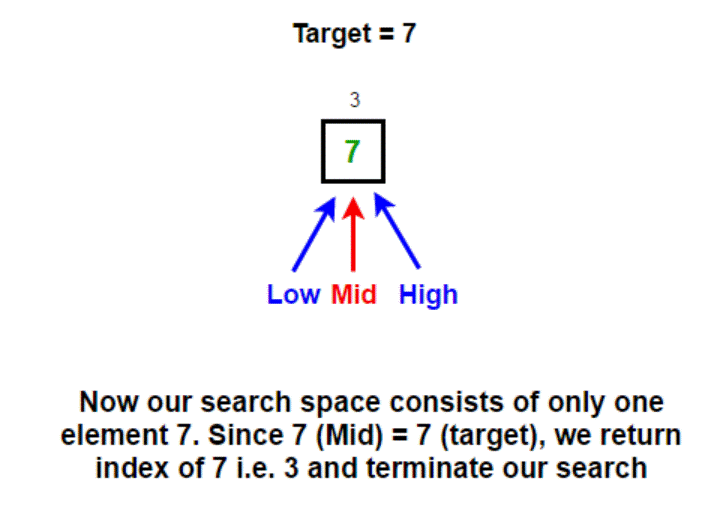
**Example:**

**nums = [2, 3, 5, 7, 8, 10, 12, 15, 18, 20]**

**target = 7**

****





***Complexity Analysis of Binary Search:***

***Best Case:***

In binary search, the key is initially compared to the array’s middle element. If the key is in the center of the array, the algorithm only does one comparison. As a result, the algorithm’s best-case running time is **T(n) = 1.**

***Worst Case:***

In every iteration, the binary search space is decreased by half, allowing for maximum log2n array divisions.

If the key is at the leaf of the tree or it is not present at all, then the algorithm does log2n comparisons, which is the maximum.

The number of comparisons increases in **logarithmic** proportion to the amount of the input. As a result, the algorithm’s worst-case running time would be **T(n) = O(log2 n).**

The problem size is reduced by a factor of two after each iteration, and the method does one comparison. **Recurrence of binary search can be written as T(n) = T(n/2) + 1.**

The solution to this recurrence leads to the same running time, i.e.**O(log2n).**

**T (n) = T(n/2) + 1 …(1)**

Substitute n by n/2 in Equation (1) to find T(n/2)

**T(n/2) = T(n/4) + 1 …(2)**

Substitute value of T(n/2) in Equation (1),

**T(n) = T(n/22) + 1 …(3)**

Substitute n by n/2 in Equation (2) to find T(n/4),

T(n/4) = T(n/8) + 1

Substitute value of T(n/4) in Equation (3),

T(n) = T(n/23) + 3

**.**

**.**

**.**

**.**

**.**

**After k iterations,**

T(n) = T(nk) + k

**A binary tree created by binary search can have maximum height log2n**

So, k = log2n ⇒ n = 2k

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

= T(1) + k

From the base case of recurrence,

T(n) = 1 + k = 1 + log2n

**T(n) = O(log2n)**

***Average Case:***

The average case for binary search occurs when the key element is neither in the middle nor at the leaf level of the search tree.

On average, it does half of the log2 n comparisons, which will turn out as

**T (n) = O(log2 n).**

**Strassen'smatrixmultiplication**

***Naive Method:*** Following is a simple way to multiply two matrices.

void multiply(int A[ ][N], int B[ ][N], int C[ ][N])

{

for (int i = 0; i< N; i++)

{

for (int j = 0; j < N; j++)

{

C[i][j] = 0;

for (int k = 0; k < N; k++)

{

C[i][j] += A[i][k]\*B[k][j];

}

}

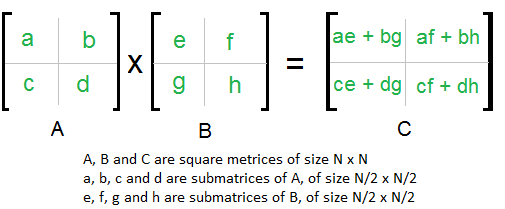
}

}

Time Complexity of above method is O(N3).

***Divide and Conquer:***

Following is a simple Divide and Conquer method to multiply two square matrices.  
1) Divide matrices A and B into 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.  
2) Calculate the following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

[](https://www.geeksforgeeks.org/wp-content/uploads/strassen_new.png)

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. The addition of two matrices takes O(N2) time. So, the time complexity can be written as

T(N) = 8T(N/2) + O(N2)

From [Master's Theorem](https://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/),the time complexity of the above method is O(N3)which is unfortunately the same as the above naive method.

***Simple Divide and Conquer also leads to O(N3), can there be a better way?***

In the above divide and conquer method, the main component for high-time complexity is 8 recursive calls. The idea of**Strassen’s method** is to reduce the number of recursive calls to 7.

Strassen’s method is similar to the above simple divide and conquers method in the sense that this method also divides matrices into sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen’s method, the four sub-matrices of the result are calculated using following formulae.

***Strassen’s method:***

Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions. **.(2log27 =22.807)**

This reduction can be done by ***Divide and Conquer Approach.***

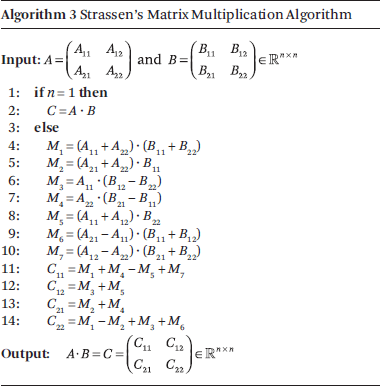
*Divide-and-conquer is a general algorithm design paradigm:*

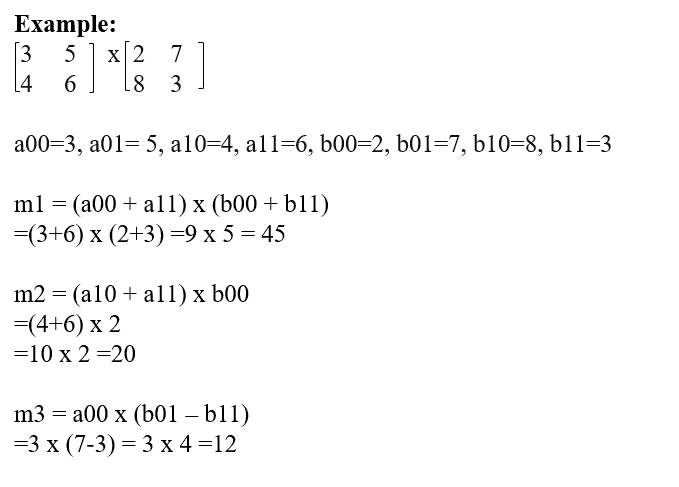
*Divide:* divide the input data S into two or more disjoint subsets S1, S2, …

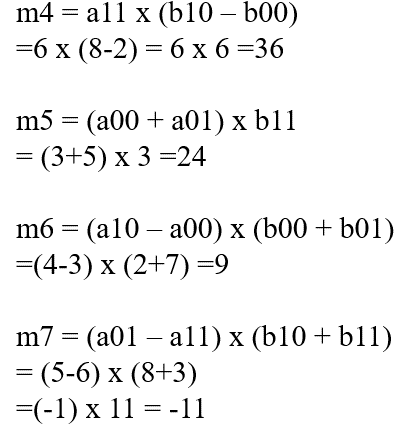
*Recur:* solve the subproblems recursively

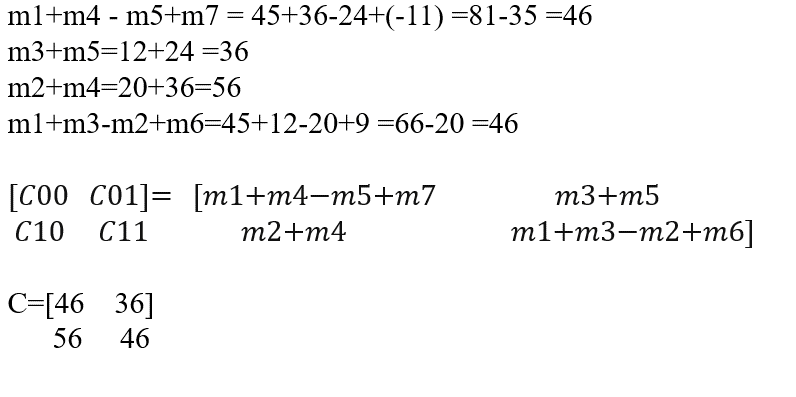
*Conquer:* combine the solutions for S1, S2, …, into a solution for S

The base case for the recursion is subproblems of constant size.Analysis can be done using recurrence equations.Divide matrices into sub-matrices and recursively multiply sub-matrices









***Recurrence equation for Strassen’s approach***

T(n) = 7.T(n/2)

Two matrices of size 1 x 1 need only one multiplication, so the base case would be, T (1) = 1.

Let us find the solution using the iterative approach.

By substituting n = (n / 2),

T(n) = 7.T(n/2)

T(n/2) = 7.T(n/4)

⇒ T(n) = 7.2T(n/22)

**T(n) = 7k.T(2k)**

Let’s assume n = 2k⇒ k = log2 n

T (n) = 7k .T(2k / 2k)

= 7k .T (1)

= 7k

= 7log2 n

= nlog2 7

= **n2.81< n3**

*NOTE:*

Generally, Strassen’s Method is generally not preferred for practical applications for the following reasons.  
1) The constants used in Strassen’s method are high and for a typical application Naive method works better.

2) For Sparse matrices, there are better methods especially designed for them.

3) The submatrices in recursion take extra space.

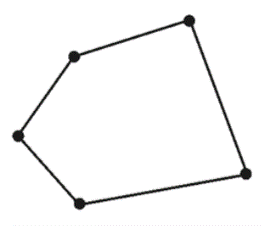
4) Because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen’s algorithm than in Naive Method.

**CONVEX HULL**

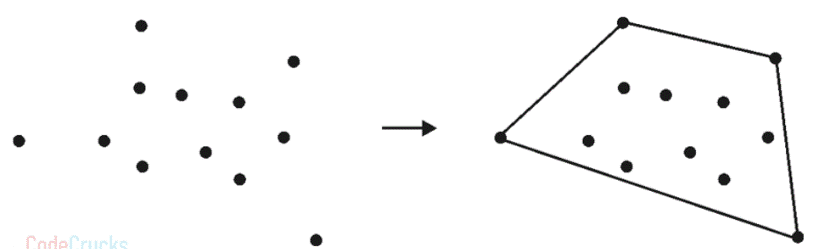
***In this problem, we want to compute the convex hull of a set of points.***

* Formally: It is the smallest convex set containing the points. A convex set is one in which if we connect any two points in the set, the line segment connecting these points must also be in the set.
* Informally: It is a rubber band wrapped around the "outside" points.

The convex hull is the smallest region covering a given set of points.Polygon is called a convex polygon if the angle between any of its two adjacent edges is always less than 1800. Otherwise, it is called a concave polygon. Complex polygons are self-intersecting polygons.



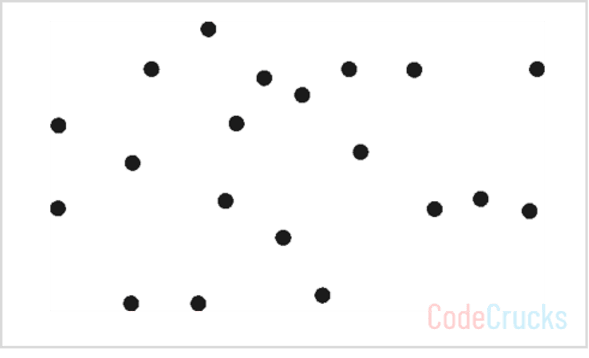
Set of points Q is the convex polygon P that encompasses all of the points given. The problem of finding the smallest polygon P such that all the points of set Q are either on the boundary of P or inside P is known as the **convex hull problem.** The vertex of a polygon is a point shared by **two neighboring edges.**



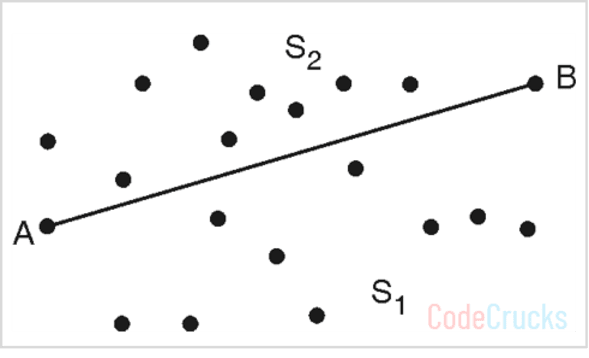
|  |  |
| --- | --- |
| **Algorithm ConvexHull(P)**  // P is a set of input points  Sort all the points in P and find two extreme points A and B  S1 ← Set of points right to the line AB  S2 ← Set of points right to the line BA  Solution ← AB followed by BA  **Call FindHull(S1, A, B)**  **Call FindHull(S2, B, A)** | **Algorithm FindHull(P, A, B)**  if isEmpty(P) then  return  else  C ← Orthogonally farthest point from AB  Solution ← Replace AB by AC followed by CB  Partition P – { C } in X0, X1 and X2  Discard X0 in side triangle  **Call FindHull(X1, A, C)**  **Call FindHull(X2, C, B)**  end |

**Example:**

***Find the convex hull for a given set of points using the divide and conquer approach.***



***Step 1:*According to the algorithm, find left most and rightmost points from the set P and label them as A and B. Label all the points on the right of AB as S1 and all the points on the right of BA as S2.**

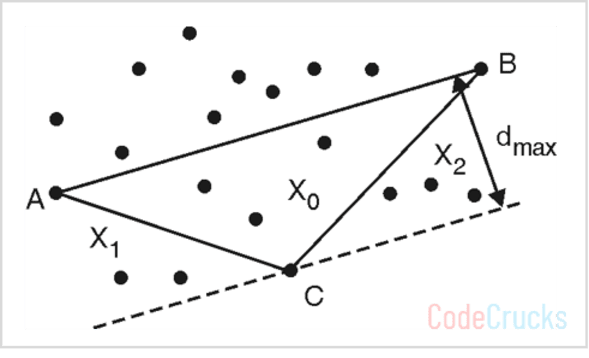


Solution = {AB, BA}

Make a recursive call to FindHull (S1, A, B) and FindHull(S2 ,B, A)

***Step 2:*FindHull(S1 ,A, B)**

**Find point C orthogonally farthest from line AB**



Solution = Solution – { AB } ∪ {AC, CB}

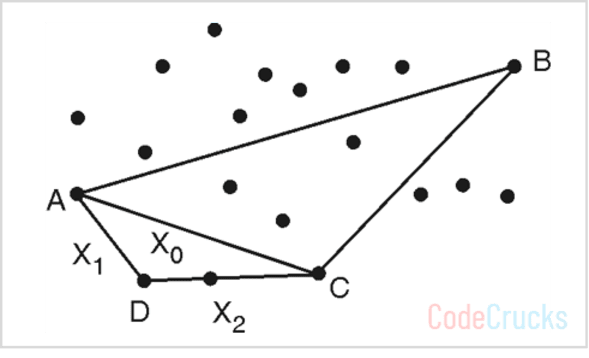
= {AC, CB, BA}

Label regions X0, X1 and X2 as shown in above figure

Make recursive calls: FindHull (X1, A, C) and FindHull (X2, C, B)

***Step 3:*FindHull(X1, A, C)**

**Find point D orthogonally farthest from line AC**



Solution = Solution – {AC} ∪ {AD, DC}

= {AD, DC, CB, BA}

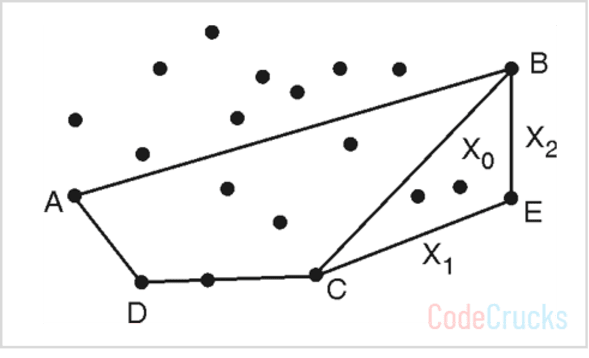
Label regions X0, X1 and X2 as shown in above figure

Make recursive calls: FindHull (X1, A, D) and FindHull (X2, D, C)

But X1 and X2 sets are empty, so the algorithm returns.

***Step 4:*FindHull(X2, C, B)**

**Find point E orthogonally farthest from line CB**



Solution = Solution – {CB} ∪ {CE, EB}

= {AD, DC, CE, EB, BA}

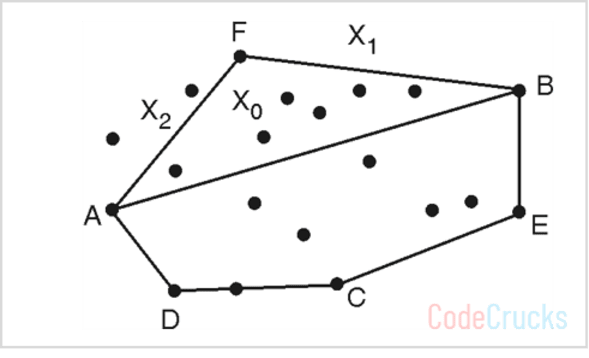
Label regions X0, X1, and X2 as shown in Fig.

Make recursive calls: FindHull (X1, C, E) and FindHull (X2, E, B).

But X1 and X2 sets are empty, so the algorithm returns Now we will explore the points in S2, on the right-hand side of the line BA

***Step 5:*FindHull(S2 ,B, A)**

**Find point F orthogonally farthest from line BA**



Solution = Solution – {BA} ∪ {BF, FA}

= {AD, DC, CE, EB, BF, FA}

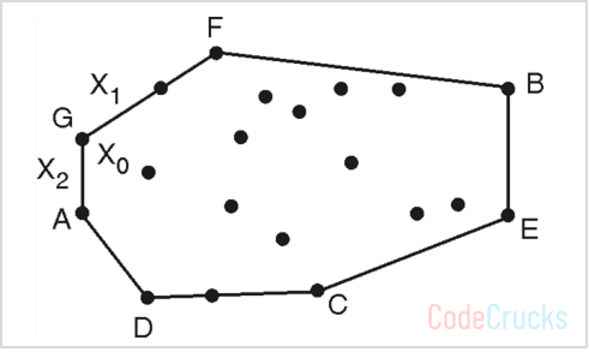
Label regions X0, X1 and X2 as shown in above figure.

Make recursive calls :FindHull (X1, B, F) and FindHull (X2, F, A)

But X1 set is empty, so call to FindHull (X1, B, F) returns

***Step 6:*FindHull (X2, F, A)**

**Find point G orthogonally farthest from line FA**



Solution = Solution – {FA} ∪ {FG, GA}

= {AD, DC, CE, EB, BF, FG, GA}

Make recursive calls: FindHull (X1, F, G) and FindHull (X2, G, A).

But X1 and X2 sets are empty, so the algorithm returns.

And no more recursive calls are left.

So,a polygon with edges (AD, DC, CE, EB, BF, FG, GA) is the convex hull of given points.

***Analysis:***

* Pre-processing step is to sort the points according to the increasing order of their X coordinate.
* **Sorting: O(nlog2n).**
* **Finding** the two farthest points from the sorted list:**O(1).**
* **Dividing points** into two halves S1 and S2: **O(1)** by joining A and B.

**T(n) = 2T(n/2) + O(n) + O(1)**

**= 2T(n/2) + n … (1)**

Solving original recurrence for n/2,

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

Substituting this in equation (1),

T(n) = 2[ 2T(n/4) + n/2 ] + n

= 22 T(n/22) + 2n

.

.

After k substitutions,

**T(n) = 2k T(n/2k) + k.n … (2)**

Division of array creates a binary tree, which has height log2n, so let us consider that k grows up to log2n,

k = log2n ⇒ n = 2k

Substitute these values in equation (2)

T(n) = n.T(n/n) + log2n . n

**T(n) = O(n.log2n)**

***Applications:***

* Collision avoidance
* Smallest Box
* Shape Analysis

**Decrease and Conquer Approach: Topological Sort**

As divide and conquer technique, which includes dividing the problem into smaller sub-problems of the same problem, then conquering the sub-problems by solving them recursively and if the sub-problem sizes are small enough, then solving them in a straightforward manner, and in last combining the solutions to the sub-problems into the solution for the initial big problem.

Similarly, the approach decrease and conquer works, it also includes steps such as, dividing the problem into smaller instances of the same problem, then conquering the problem by solving a smaller instance of the problem using recursion and extending the smaller instance to obtain the solution to a big problem.

But the difference is that it divides the problem into sub-problem but solves only one of the sub-problems which makes the condition holds true. Thus, decreasing the size of the original problem. This approach is known as the decrease and conquer approach and is also known as the incremental or inductive approach.

*This approach can be either implemented as top-down or bottom-up.*

***Top-down approach:*** It always leads to the recursive implementation of the problem.

***Bottom-up approach:*** It is usually implemented in an iterative way, starting with a solution to the smallest instance of the problem.

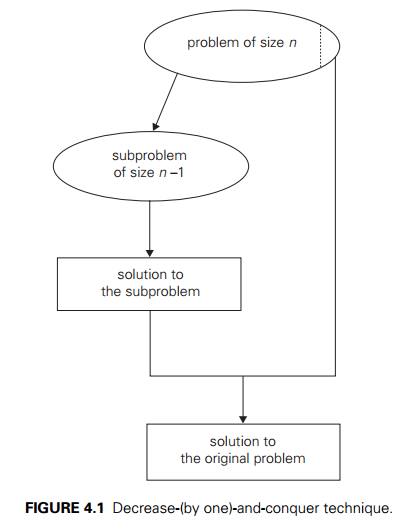
***Variations in Decrease and Conquer Technique***

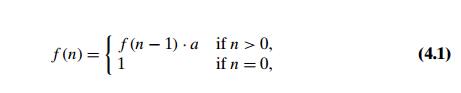
There are three major variations of decrease and conquer:

* ***Decrease by a constant***
* ***Decrease by a constant factor***
* ***Variable size decrease***

***Decrease by a Constant :***In this variation, the size of an instance is reduced by the same constant on each iteration or the recursive step of the algorithm. Typically, this constant is equal to one , although other constant size reductions can happen. This variation is used in many algorithms like;

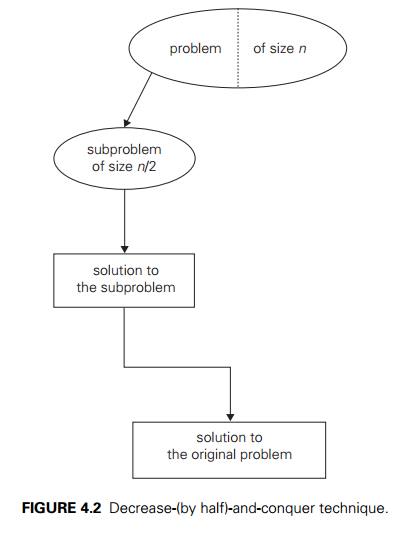
* Insertion sort
* Graph search algorithms: DFS, BFS
* Topological sorting
* Algorithms for generating permutations, or subsets

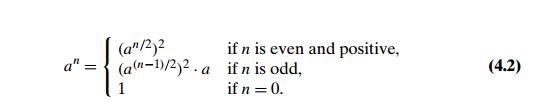




***Decrease by a Constant factor***: This technique reduces a problem instance by the same constant factor on each iteration or the recursive step of the algorithm. In most applications, this constant factor is equal to two. A reduction by a factor other than two is especially rare. Decrease by a constant factor algorithmis very efficient especially when the factor is greater than 2 as in the fake-coin problem. Examples where this approach is used;

* Binary search
* Fake-coin problems
* Russian peasant multiplication

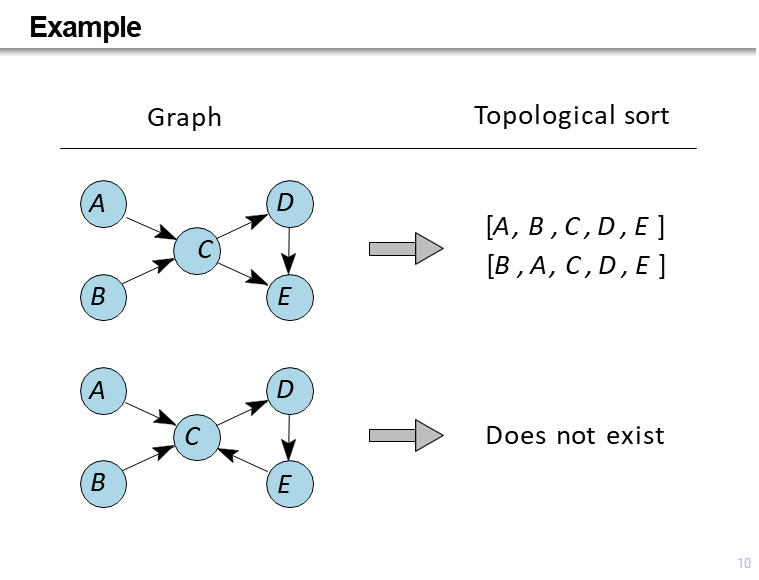


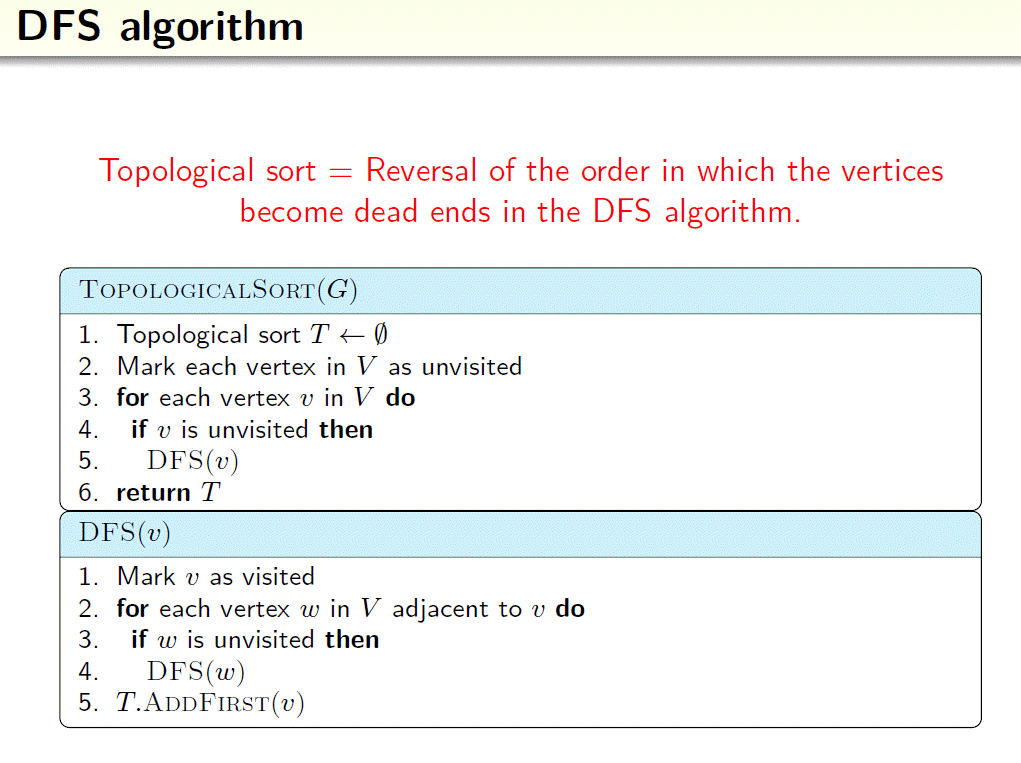


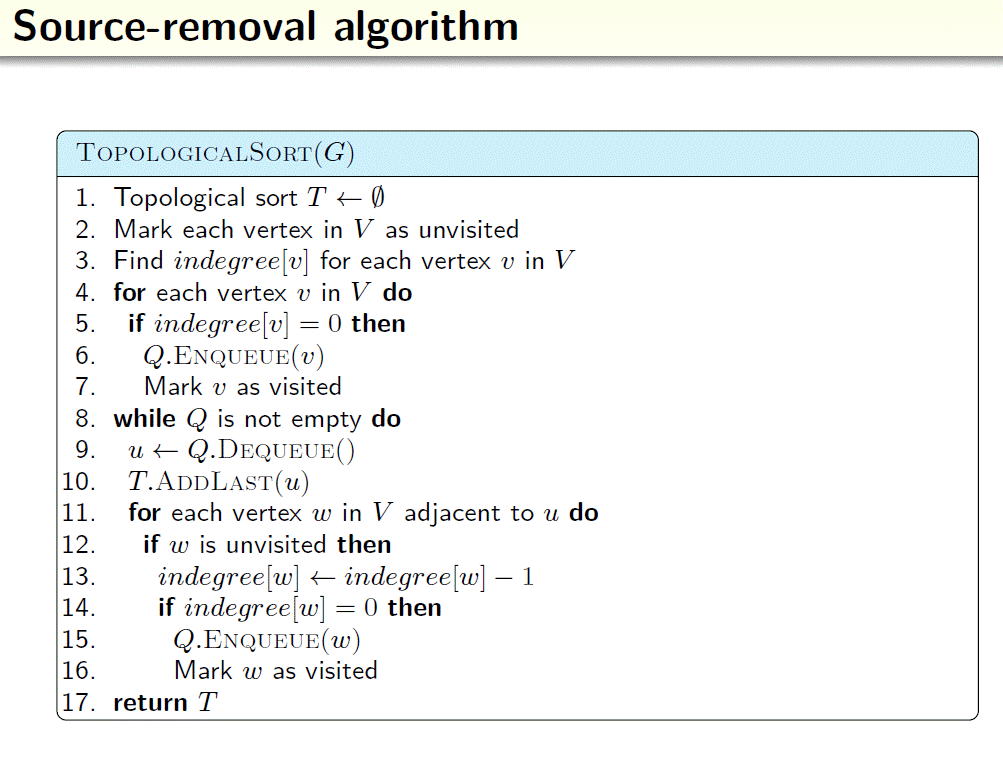
***Variable-Size-Decrease:*** In this variation, the size-reduction pattern varies from one iteration or step of an algorithm to another. As, in problem of finding gcd of two number though the value of the second argument is always smaller on the right-handside than on the left-hand side, it decreases neither by a constant nor by a constant factor. Below are some examples which uses this approach:

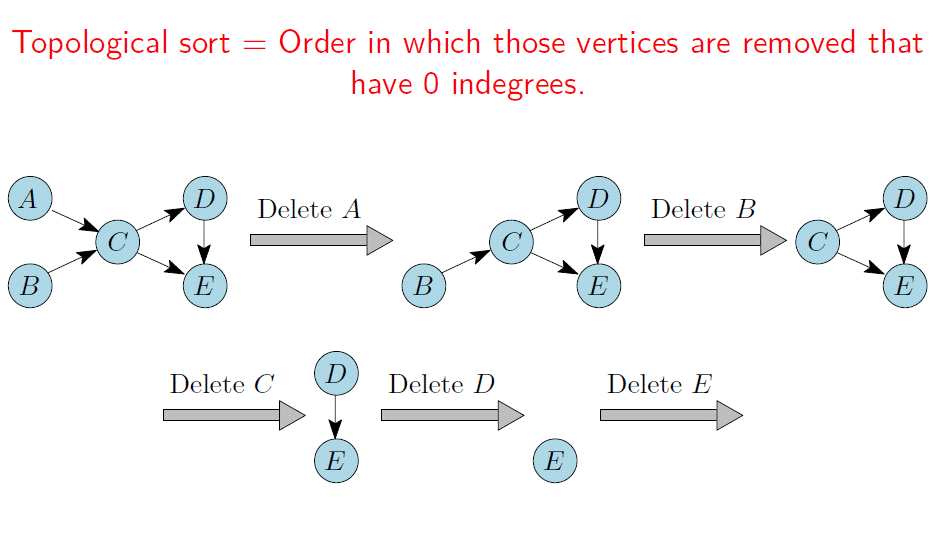
* Computing median and selection problem
* Interpolation Search
* Euclid’s algorithm

**Problem:Topological sorting of vertices of a directed acyclic graph is an ordering of the vertices *v*1*, v*2*, . . . ,vn*in such a way that there is an edge directed towards vertex *vj*from vertex *vi*, then *vi* comes before *vj*.**









***Applications of Topological Sorting:***

* Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs.
* In computer science, applications of this type arise in:
* Instruction scheduling
* Ordering of formula cell evaluation when recomputing formula values in spreadsheets
* Logic synthesis
* Determining the order of compilation tasks to perform in making files
* Data serialization
* Resolving symbol dependencies in linkers.

**RELEVANT READING MATERIAL AND REFERENCES:**

**Source Notes:**

1. <https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_np_hard_complete_classes.htm>
2. <https://www.javatpoint.com/daa-complexity-classes>

**Lecture Video:**

1. https://youtu.be/e2cF8a5aAhE

|  |  |  |
| --- | --- | --- |
| **SR NO** | | **BOOK NAME** |
| T1 | | **Introduction to Algorithms** by Thomas H. Cormen, Charles E. Leiserson,Ronald L.3RD edition, PHI Learning Private Limited (2012). |
| T2 | | **Introduction to the Design and Analysis of Algorithms** by Anany Levitin 3RD edition, Pearson Education (2012) |
| R1 | | **Design & Analysis of Computer Algorithms** by Alfred V Aho, John E Hopcroft, Jeffrey D Ullman 1ST edition, Pearson Education Limited (2013) |
| R2 | | **Fundamentals of Computer Algorithms** by Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajas 2ND edition, Universities Press (2008) |
| **SR NO** | **Tutorial/ Video Lectures Link** | |
| 1 | https://nptel.ac.in/courses/106101060/ | |
| 2 | [TPAT-CourseContent-DAA - Google Drive](https://drive.google.com/drive/u/0/folders/1oBFYeRgZTy7-qtUi796_uxgKQfPmVUvt) | |
| 3 | **Algorithmica**–A journal about the design of algorithms in many applied and fundamental areas http://www.springerlink.com | |
| 4 | https://dokumen.tips/documents/algorithm-analysisanddesign558445f7ef0b1.html?page=1 | |
| 5 | [CS-245 PPT.pdf - Google Drive](https://drive.google.com/file/d/174BXxM9TA5qVJZl1O4IOLN2OK2HG2qTY/view) | |
| 6 | <http://vssut.ac.in/lecture_notes/lecture1428551222.pdf> | |