Unit :-

I

Algorithm Designing

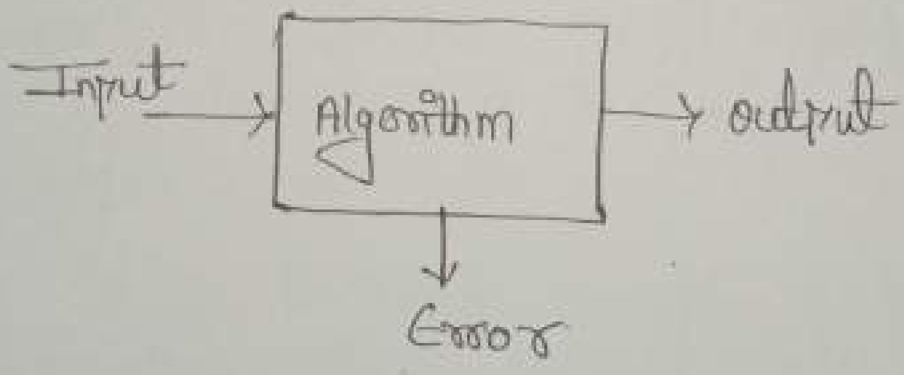
Algorithms

Topic: Algorithm, Designing of algorithm.

Objective: Student will learn definition of algorithm, how to design of algorithm.

Outcomes: Student will able to compute time complexity of simple algorithm.

Algorithm:- An algorithm is any set of detailed instructions which result in a Predictable end-state from a known beginning. Algorithms are only as good as the instructions given, however, and the result will be incorrect if the algorithm is not properly defined. Algorithms are used for calculation, data Processing and automated reasoning.



Algorithm.

Input :- There are zero or more quantities, which are externally supplied.

Output :- At least one quantity is produced.

Definiteness :- Each instruction must be clear.

Finiteness :- If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps.

Effectiveness :- Every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

- Algorithm is a backend concept of Program.
- A program does not necessarily satisfy the fourth condition.
- One important example of such a program for a computer is its operating system, which never terminates but continues in a wait loop until more jobs are entered.

Example :-

```
PUZZLE(x)
while x!=1
  if x is even
then x = x/2
 else x = 3x+1
```

Input :- x = 2

Output :- 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

"Algorithmic is the backend concept of the program or it just like the recipe of the program".

Designing Algorithms :-

One the Basis of Implementation :-

- Serial, Parallel or distributed :-

Serial Algorithm :- A sequential algorithm or serial algorithm is an algorithm that is executed sequentially - once through, from start to finish, without other processing executing.

Parallel Algorithm :- A parallel Algorithm is an algorithm which can be executed a piece at a time on many different processing devices and then combined together again at the end to get the correct result.

Distributed Algorithm :- A distributed algorithm is an algorithm designed to run on computer hardware constructed from interconnected processors. Distributed algorithms are used in many varied application areas of distributed computing, such as telecommunications, scientific computing, distributed information processing and real time process control.

• Recursion or Iteration:-

Recursion Algorithm:- A recursion Algorithm is an algorithm which call itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operation to the returned value of for the smaller (or simpler) input.

Iteration Algorithm:- An iteration algorithm executes steps in iterations. It aims to find successive approximation is sequence to reach a solution. They are most commonly used in linear programs where large numbers of variables are involved.

Advantages of an Algorithm.

- Effective Communication :- Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.

- Easy Debugging :- A well designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.

- Easy and Efficient Coding :- An algorithm is nothing but a blueprint of a program that helps develop a program.

- Independent of Programming Language :- Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantage of an Algorithm.

- Developing algorithms for Complex problems could be time-consuming and difficult to understand.

- It is a challenging task to understand complex logic through algorithm.

Pseudocode:-

Pseudocode refers to an informal high-level description of the operating principle of a computer program or other algorithm. It uses structural conventions of a standard programming language intended for a human reading rather than the machine reading.

P.T.O

Advantages of Pseudocode:-

- Since it is similar to a programming language, it can be quickly transformed into the actual programming language than a flowchart.
- The layman can easily understand it.
- Easily modifiable as compared to the flowcharts.
- Its implementation is beneficial for structured, designed elements.

Disadvantage of Pseudocode:-

- Since it does not incorporate any standardized style or format, it can vary from one company to another.
- Error possibility is higher while transforming into a code.
- It may require a tool for extracting out the Pseudocode and facilitate drawing flowcharts.
- It does not depict the design.

Difference between Algorithm & the Pseudocode

An algorithm is simply a problem-solving process, which is used not only in Computer science to write a program but also in our day to day life. It is nothing but a series of instructions to solve a problem or get to the problem's solution.

However, Pseudocode is a way of writing an algorithm. Programmers can use informal, simple language to write pseudocode without following any strict syntax.

Refrences :-

R_1 :- Reference by Ellis Horowitz.

R_2 :- Reference by Sartaj Sahni.

Summary : Algorithm is a well defined Computational Procedure we can analyze the algorithm assuming that.

Analyzing

Algorithm

Topic: Analyzing Algorithm.
Objective: student will learn definition of analyzing algorithm.
Outcomes: student will able to compute time complexity of simple algorithm.

Analyzing Algorithm :- The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. Moreover, the analysis of an algorithm can help us understand it better, and can suggest informed improvements. Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

Analysis of Algorithms :-

A complete analysis of the running time of an algorithm involves the following steps:

~~Differ~~

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modelled input.
- Calculate the total running time by multiplying the time by the frequency for each operation, the adding all the products.

Complexity of An Algorithm :-

- Time Complexity :- The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run. It is commonly estimated by counting the number of elementary operations

Performed by the algorithm, where an elementary operation takes a fixed amount of time to perform.

• Space Complexity :- This is essentially the number of memory cells which an algorithm needs to run. A good algorithm keeps this number as small as possible.

• Amortized analysis :-

Sometimes we find the statement in the manual that an operation takes amortized time $O(f(n))$.

This means that the total time for n such operations is bounded asymptotically from above by a function $g(n)$ and that $f(n) = O(g(n)/n)$. So the amortized time is the average time of an operation is the worst case.

Step Count :-

- s/e is the number of steps per execution of the statement.
- Frequency is how often each statement is executed.
- The time Complexity is estimated as Total steps.

Statement	s/e	frequency	Total.
1.) Algorithm sum(a,n)	0	-	0
2.) {	0	-	0
3.) s=0.0;	1	1	1.
4.) for I=1 to n do	1	n+1	n+1
5.) s=s+a[I];	1.	n	n.
6.) return s;	1	1	1.
7.) }	0	-	0
Total			2n+3.

Worst-case Complexity :- The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. It represents the curve passing through the highest point of each column.

Best-case Complexity :- The best-case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n. It represents the curve passing through the lowest point of each column.

Average-case Complexity :- The average-case complexity of the algorithm is the function defined by the average number of steps taken on any instance of size n.

1.15

Algorithm Design Techniques :-

The following is a list of several popular ~~desing~~ design approaches :-

1.) Divide and Conquer Approach :- It is a top-down approach. The algorithms which follow the divide & conquer techniques involves three steps :-

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2.) Greedy Technique :- Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized i.e some constraints or conditions.

- Greedy Algorithm always makes the choice looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always ~~given~~ guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

3.) Dynamic Programming :- Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large.

Dynamic Programming is frequently related to Optimization Problems.

4) Branch and Bound :- In Branch & Bound Algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

5) Randomized Algorithms :- A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

6.) Randomized Algorithm :- A randomized algorithm uses a random number at least once during the computation make a decision.

7.) Backtracking Algorithm :- Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution.

References :-

1.) Reference by E. Horowitz.

2.) R2 :- Thomas h. Cormen

Summary : Algorithm is a well defined computational procedure and also design algorithm.

Asymptotic

Notations

Topic: Asymptotic Notations.

Objective: student will learn different types of asymptotic Notations.

Outcomes: student will able to compare the complexity of two algorithm.

Asymptotic Notations :- Resources for an algorithm are usually expressed as a function regarding input. often this function is messy and complicated to work.

We reduce the function down to the important Part.

$$\text{Let } f(n) = an^2 + bn + c$$

In this function, the n^2 term dominates the function that is when n gets sufficiently large.

Asymptotic Notation :-

The word Asymptotic means approaching a value or curve ~~asib~~ arbitrarily closely.

Asymptotic Analysis :-

It is a technique of representing limiting behavior. The methodology has the applications across science.

It can be used to analyze the performance of an algorithm for some large data set.

-> In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets.

The simplest example is a function $f(n) = n^2 + 3n$, the term $3n$ becomes insignificant compared to n^2 when n is very large.

The function "f(n) is said to be asymptotically equivalent to n^2 as $n \to \infty$", and here is written symbolically as $f(n) \sim n^2$.

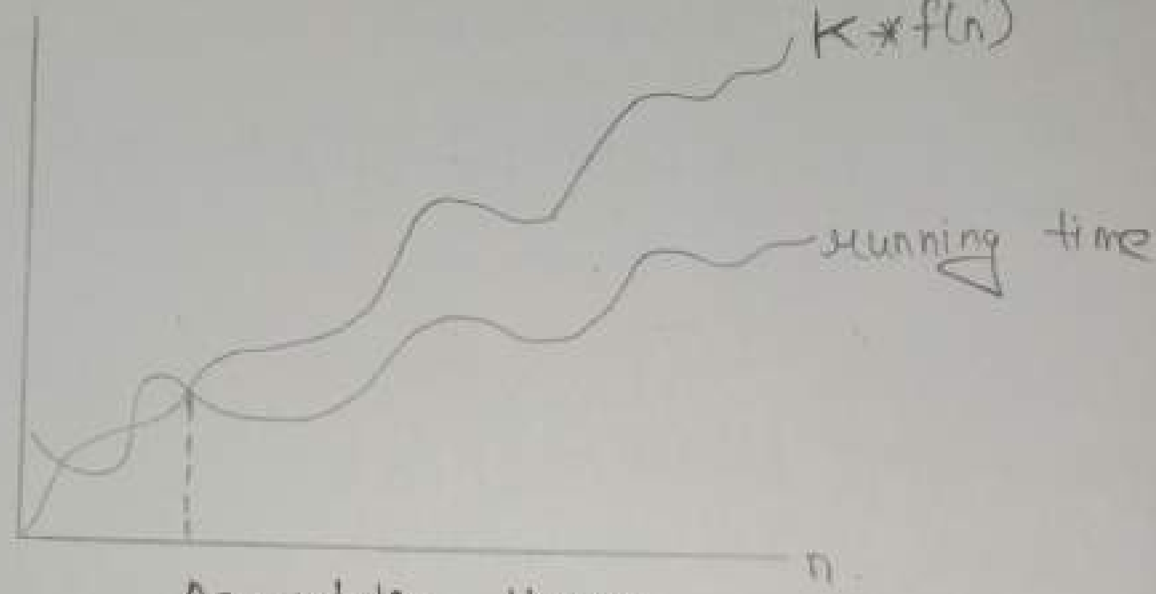
Asymptotic Notations :-

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes.
Three notations are used to calculate the running time complexity of an algorithm:

1.) Big-oh notation :- Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function $f(n) = O(g(n))$ [read as "f of n is big-oh of g of n"] if and only if exist positive contant c and such that.

$f(n) \le k \cdot g(n) f(n) \le k \cdot g(n)$ for $n > n0 \, n > n0$ in all cas.

Hence, function g(n) is an upper ~~bond~~ bound for function f(n), as g(n) grows faster than f(n).



Asymptotic Upper Bound

for Example :-

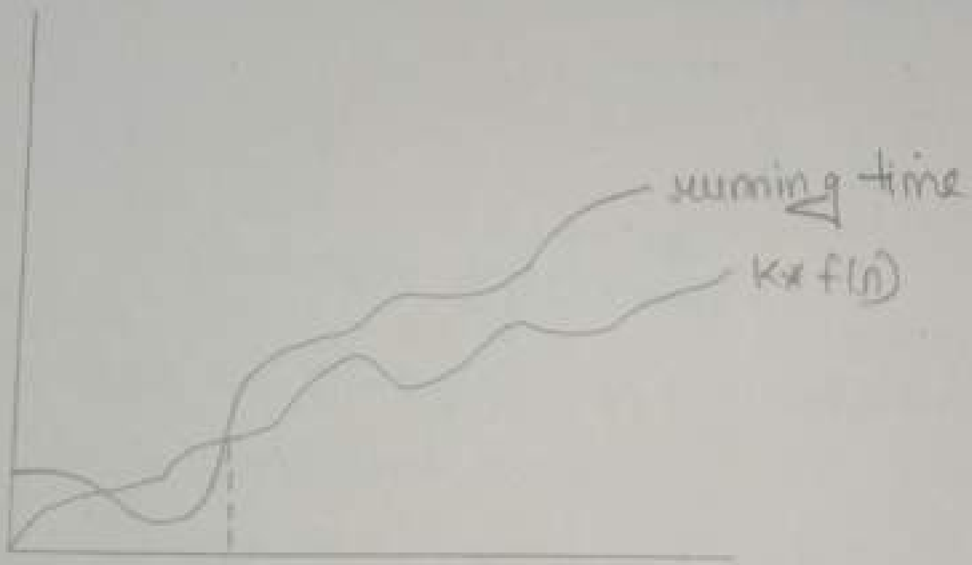
1.) $3n+2 = O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$

2.) $3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$.

Hence, the Complexity of f(n) can be represented as $O(g(n))$.

2) Omega() Notation :- The function $f(n) = \Omega(g(n))$ [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that.

$$f(n) \geq K * g(n) \text{ for all } n, n \geq n_0$$



Asymptotic lower bound.

for Example:-

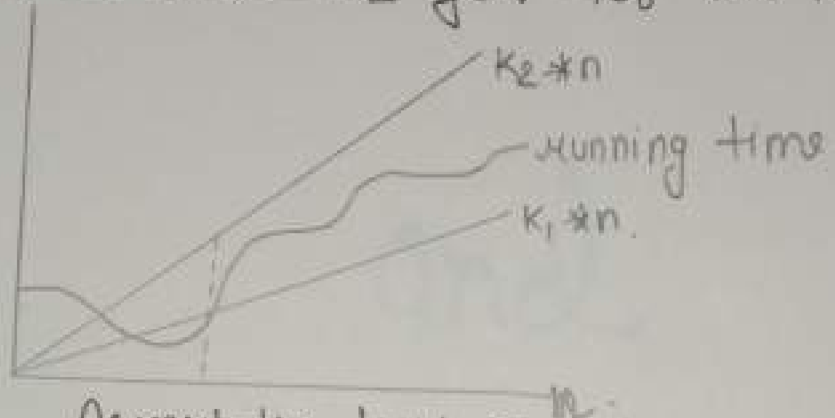
$$f(n) = 8n^2 + 2n - 3 \geq 8n^2 - 3$$
$$= 7n^2 + (n^2 - 3) \geq 7n^2 \ (g(n))$$

Thus, $K_1 = 7$

Hence, the complexity of $f(n)$ can be represented as $\Omega(g(n))$.

3.) Big - THETA (θ) :- The function $f(n) = \theta(g(n))$ [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that.

$$k_1 * g(n) \leq f(n) \leq k_2\, g(n) \text{ for all } n, n \geq n_0$$



Asymptotic tight Bound.

for Example:-

$3n+2 = \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n.

$k_1 = 3$, $k_2 = 4$ and $n_0 = 2$.

Reference :- R1 :- Reference by E Horowitz.

R2 :- Reference by Thomas h. Cormen.

Summary : There are majorly three types of asymptotic notations O, Ω, θ, On the basis of time complexity is measured on the basis of there asymptotic notations.

Divide

and

Conquer

Topic: Divide and conquer Introduction.
Objective: Student will learn divide & conquer strategy & how to compute time complexity of such algo.
Outcomes: Student will able to compute the time complexity of recursion function.

Divide and Conquer Introduction :-

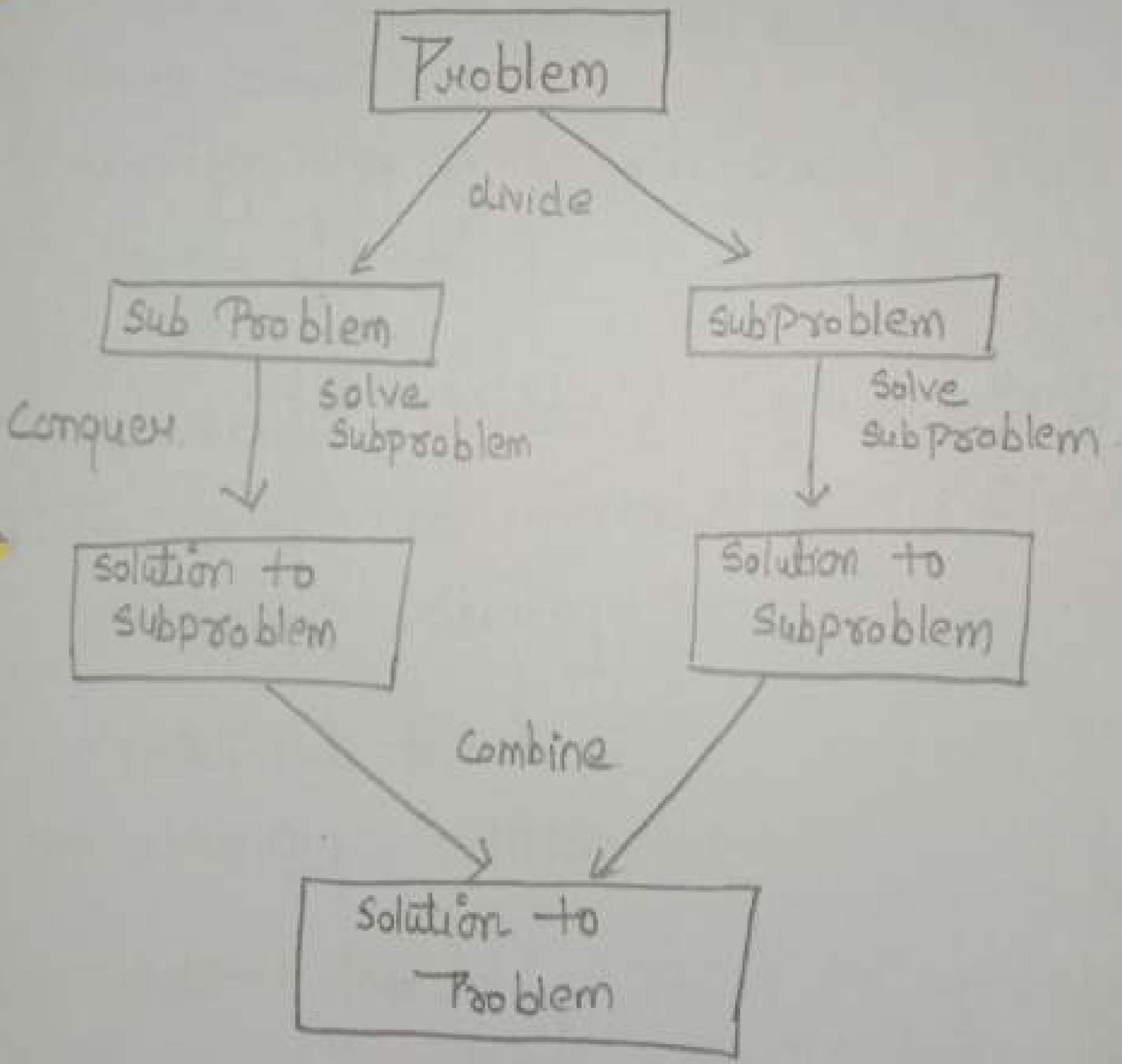
Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer strategy.

Divide and conquer algorithm consists of a dispute using the following three steps.

1.) Divide :- The original problem into a set of subproblems.

2) Conquer :- Solve every subproblem individually, recursively.

3) Combine :- Put together the solutions of the the subproblems to get the solution to the whole problem.



Generally, we can follow the divide-and-conquer approach in a three-step Process.

Examples :- The specific computer algorithms are based on the divide & conquer ~~algorit~~ approach :-

1.) Maximum and minimum Problem.
2.) Binary search
3.) sorting (merge sort, quick sort)
4.) Tower of Hanoi.

Fundamental of Divide & Conquer strategy :-

- There are two fundamental of Divide & Conquer strategy :-

1.) Relational formula.
2.) Stopping Condition.

1.) Relational formula :- It is the formula that we generate from the given technique. After generation of formula we apply D&C strategy. ie we break the problem recursively and solve the broken subproblems.

2.) Stopping Condition :- When we break the Problem using Divide & Conquer strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion step of D & C is called as stopping condition.

Application of Divide and Conquer Approach :-

following algorithm are based on the Concept of the Divide and Conquer Technique :-

1.) Binary Search :- The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value.

2.) Quick sort :- It is the most efficient sorting algorithm, which is also know as Partition exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the array recursively.

3.) Merge sort :- It is sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub array and then recursively sorts each of them. After the sorting is done, it merges them back.

4.) Closest Pair of Point :- It is a problem of computational geomarity. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.

5.) Strassen's Algorithm:- It is an algorithm for matrix multiplication, which is named after Volker strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.

Advantages of Divide and Conquer:-

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithm.
- It is more proficient than that of its counterpart Brute force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantage of Divide and Conquer

1) since most of its algorithms are designed by incorporating recursion, so it necessitates high memory manggement.

2) An explicit stack may overuse the space.

3) It may even crash the system if the recursion is Performed rigorously greater than the stack Present in the CPU.

Refrences :-

R1 :- Reference by Ellis Horowitz

R2 :- Reference by Sartej Sahni

Summary : Divide and Conquer and their some advantage and dis advange of divide and conquer.

Heap and Heap Sort

Topic: Heap and Heap Sort.

Objective: Student will learn heap and heap sort and analysis & their property.

Outcomes: student will able to compute the heap sort.

Heap Sort :-

Binary Heap:- Binary Heap is an array object can be viewed as complete Binary Tree. Each node of the Binary Tree corresponds to an element in an array.

1) Length [A], number of elements in array.

2) Heap-size [A], number of elements in a heap stored within array A.

The root of tree A[1] and gives index 'i' of a node that indices of its parents, left child, and the right child can be computed.

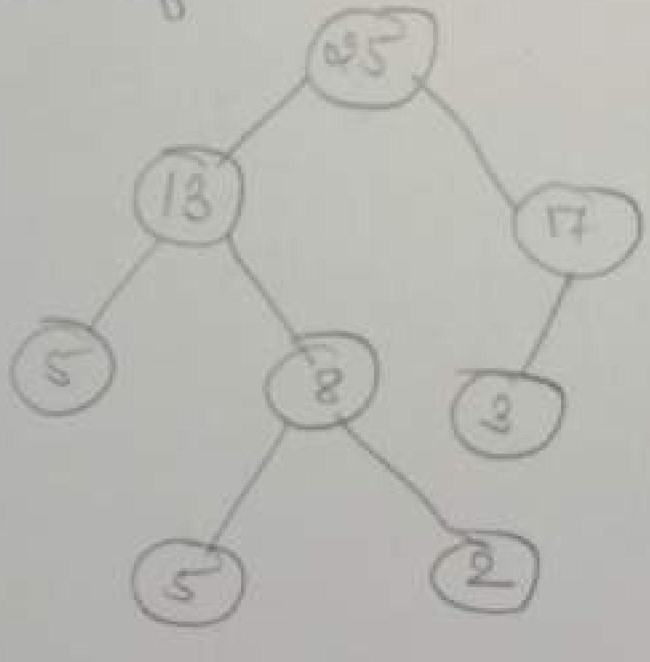
PARENT(i)
Return floor(i/2)
LEFT(i)
Return 2i
RIGHT(i)
Return 2i+1

Heap Property :-
A binary heap can be classified as Max Heap or Min heap.

1) Max heap :- In a Binary Heap, for every node I other than the root, the value of the node is greater than or equal to the value of its highest child.

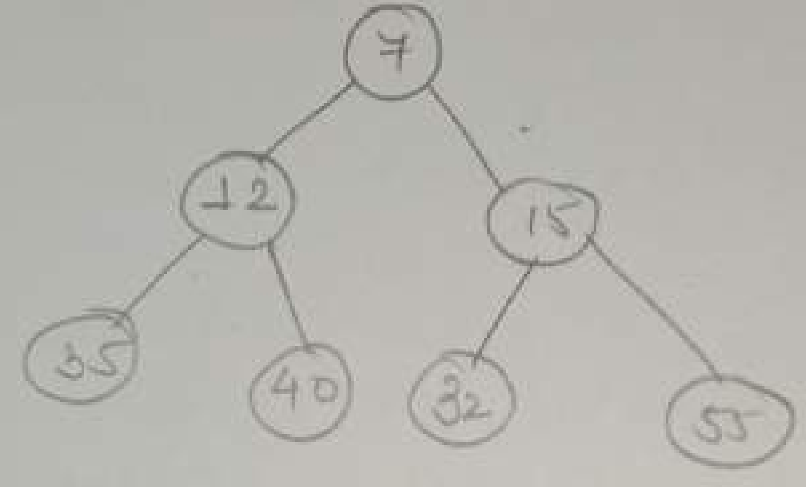
A[PARENT [i]] >= A[i]

Thus, the highest element in a heap is stored at the root. following is an example of MAX-HEAP.



(b) MIN-Heap :- In MIN-HEAP, the value of the node is lesser than or equal to the value of its lowest child.

A[PARENT (i)] ≤ A[i]



Heapify Method :-

1) Maintaining the Heap Property :- Heapify is a procedure for manipulating heap data structure. It is given an array A and index I into the array. The subtree rooted at the children of A[i] are heap but node A[i] itself may probably violate the heap property i.e A[i] < A[2i] or A[2i+1].

The procedure 'Heapify' manipulates the tree rooted as A[i] so it becomes a heap.

MAX-HEAPIFY (A, i)

1.> l ← left [i]
2.> r ← right [i]
3.> if l ≤ heap-size [A] and A[l] > A[i]
4.> then largest ← l
5.> else largest ← i
6.> If r ≤ heap-size [A] and A [r] > A[largest]
7.> Then largest ← r
8.> If largest ≠ i
9.> Then exchange A[i] A[largest]
10.> MAX-HEAPIFY (A, largest)

Analysis :-

The maximum levels an element could move up are O(log n) levels. At each level, we do simple comparison which O(1). The total time for heapify is thus O(log n).

Building a Heap :-

BUILDHEAP (array A, int n)
1.> for i ← n/2 down to 1
2.> do
3.> HEAPIFY (A, i, n).

HEAP - SORT ALGORITHM:

HEAP-SORT (A)

1. BUILD-MAX-HEAP(A)
2. For I ← length [A] down to 2
3. Do exchange A[1] ←→ A[i]
4. Heap-Size [A] ← heap-Size [A]-1
5. MAX-HEAPIFY (A, 1)

Analysis :-

Build max-heap takes O(n) running time.
The Heap Sort algorithm makes a call to 'Build Max-Heap' which we take O(n) time & each of the (n-1) calls to MAX-heap to fix up a new heap.
We know 'Max-Heapify' takes time O(log n).

The total running time of Heap-Sort is O(n logn).

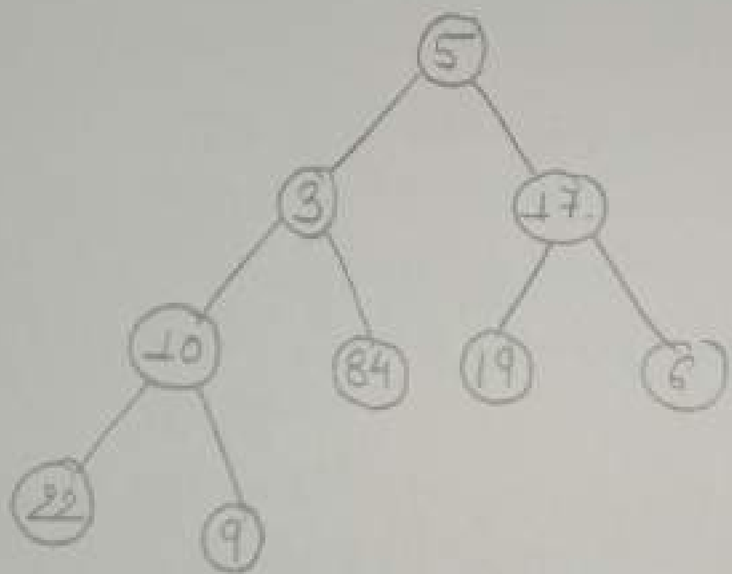
Example:- Illustrate the operation of BUILD-MAX-HEAP on the array.

A = (5, 3, 17, 10, 84, 19, 6, 22, 9)

Solution:-

Heap-size (A) = 9, so first we call MAX-HEAPIFY (A, 4)

And 1 = 4.5 = 4 to 1.



After MAX-HEAPIFY (A, 4) and i=4

L←8, r←9

l ≤ heap-size [A] and A[l] > A[i]

8 ≤ 9 and 22 > 10

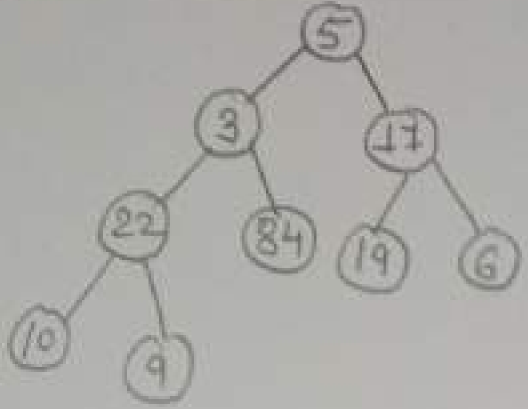
Then largest ← 8

If r ≤ heap-size [A] and A[r] > A[largest]

9 ≤ 9 and 9 > 22

If largest (8) ≠ 4

Then exchange A[4] ←→ A[8]
MAX-HEAPIFY (A, 8)



After MAX-HEAPIFY (A, 3) and i=3
l ← 6 , r ← 7
l ≤ heap-size[A] and A[l] > A[i]
6 ≤ 9 and 19 > 17
largest ← 6
If r ≤ heap-size[A] and A[r] > A[largest]
7 ≤ 9 and 6 > 19
if largest (6) ≠ 3
Then Exchange A[3] ←→ A[6]
MAX-HEAPIFY (A, 6)

After MAX-HEAPIFY (A, 2) and $i=2$

$l \leftarrow 4, r \leftarrow 5$

$l \leq$ heap-size[A] and $A[l] > A[i]$

$4 \leq 9$ and $22 > 3$

Largest $\leftarrow 4$

If $r \leq$ heap-size[A] and $A[r] > A[\text{largest}]$

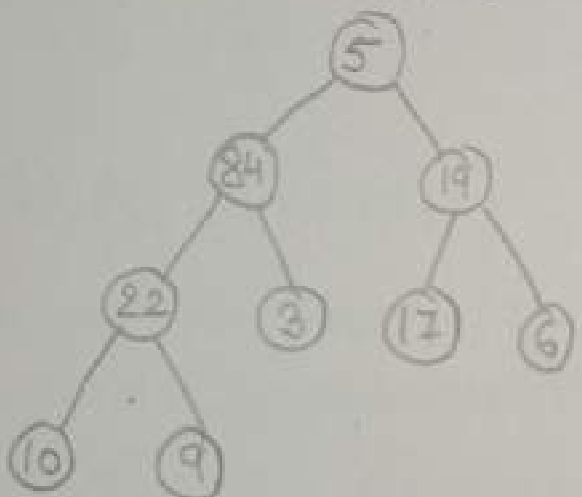
$5 \leq 9$ and $84 > 22$

Largest $\leftarrow 5$

If largest (4) $\neq 2$

Then Exchange $A[2] \leftrightarrow A[5]$

MAX-HEAPIFY (A, 5)



After MAX-HEAPIFY (A, 1) and $i=1$

$l \leftarrow 2, r \leftarrow 3$

$l \leq$ heap-size[A] and $A[l] > A[i]$

$2 \leq 9$ and $84 > 5$

Largest $\leftarrow 2$

If $r \leq$ heap-size[A] and $A[r] > A[\text{largest}]$

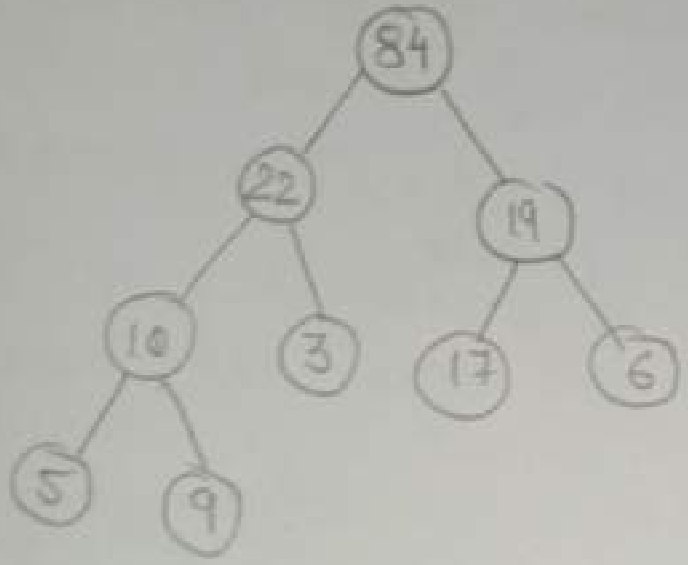
1.40

$3 \leq 9$ and $19 < 84$

If largest (2) $\neq$ 1

Then Exchange $A[1] \leftrightarrow A[2]$

MAX-HEAPIFY(A, 2)



Priority Queue:-

As with heaps, priority queues appear in two forms: max-priority queue and min-priority queue.

A ~~pr~~ priority queue is a data structure for maintaining a set S of elements, each with a combined value called a key. A max-priority queue guides the following operations:-

INSERT(S, x) :- inserts the element x into the set S, which is proportionate to the operation $S = S \cup \{x\}$.

MAXIMUM (S) :- returns the element of S with the highest key.

EXTRACT-MAX (S) :- removes and returns the element of S with the highest key.

INCREASE-KEY(S, x, K) :- increases the value of element x's key to the new value K, which is considered to be at least as large as x's current key value.

Reference :-

R1 :- Reference by thomas H. Cormen.

R2 :- Reference by Charles E. Leiserson.

Summary: Heap and Heap sort (Understood the Heap and Heap sort . and write to max & min heapify).

Merge

Sort

Topic: Merge Sort.
Objective: student will learn merge sort & how can merge concept is used for sorting.
Outcomes: Student will able to implement merge sort

Merge sort :-

Merge sort is yet another sorting algorithm that falls under the category of Divide and conquer technique. It is one of the best sorting techniques that sucessfully build a recursive algorithm.

Divide and Conquer Strategy.

In this technique, we segment a Problem into two halves and solve them individually. after finding the solution of each half, we merge them back to represent the solution of the main Problem.

Suppose we have an array A, such that our main concern will be to sort the subsection, which starts at index p and ends at index r, represented by A[p..r]

Divide.

If assumed q to be the central point somewhere in between p and r, then we will fragment the subarray A[p...r] into two arrays A[p...q] and [q+1, r].

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays A[p...q] and A[q+1, r]. In case if we did not reach the base situation, then we again follow the same procedure, i.e, we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays A[p..q] and A[q+1, r], after which we merge them back to form a new sorted array [p..r].

Merge Sort Algorithm.

The Merge sort function keeps on splitting an array into two halves until a condition is met where we try to perform mergesort on a subarray of size 1, i.e p==r.

And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

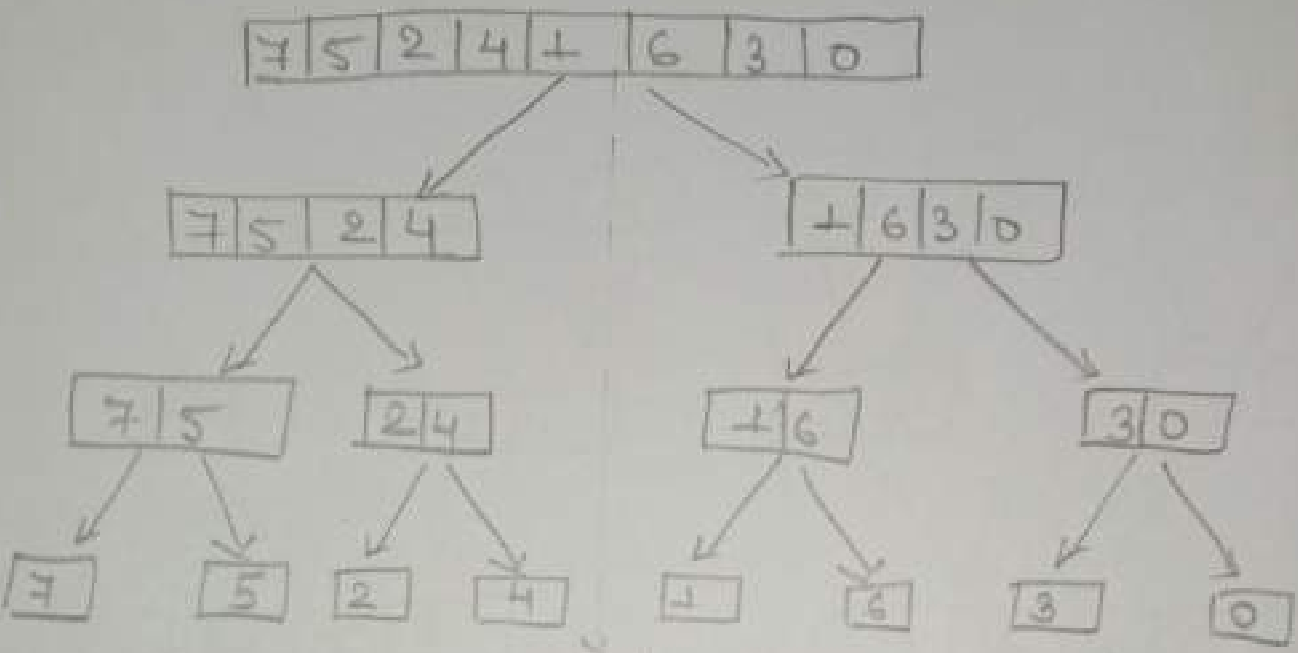
ALGORITHM-MERGE SORT

1. If p<r
2. Then q→(p+r)/2
3. MERGE-SORT (A,p,q)
4. MERGE-SORT(A,q+1,r)
5. MERGE (A,p,q,r)

Here we called mergesort (A,0, length (A)-1) to sort the complete array.

As you can see in the image given below the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.

The following figure illustrates the dividing procedure.

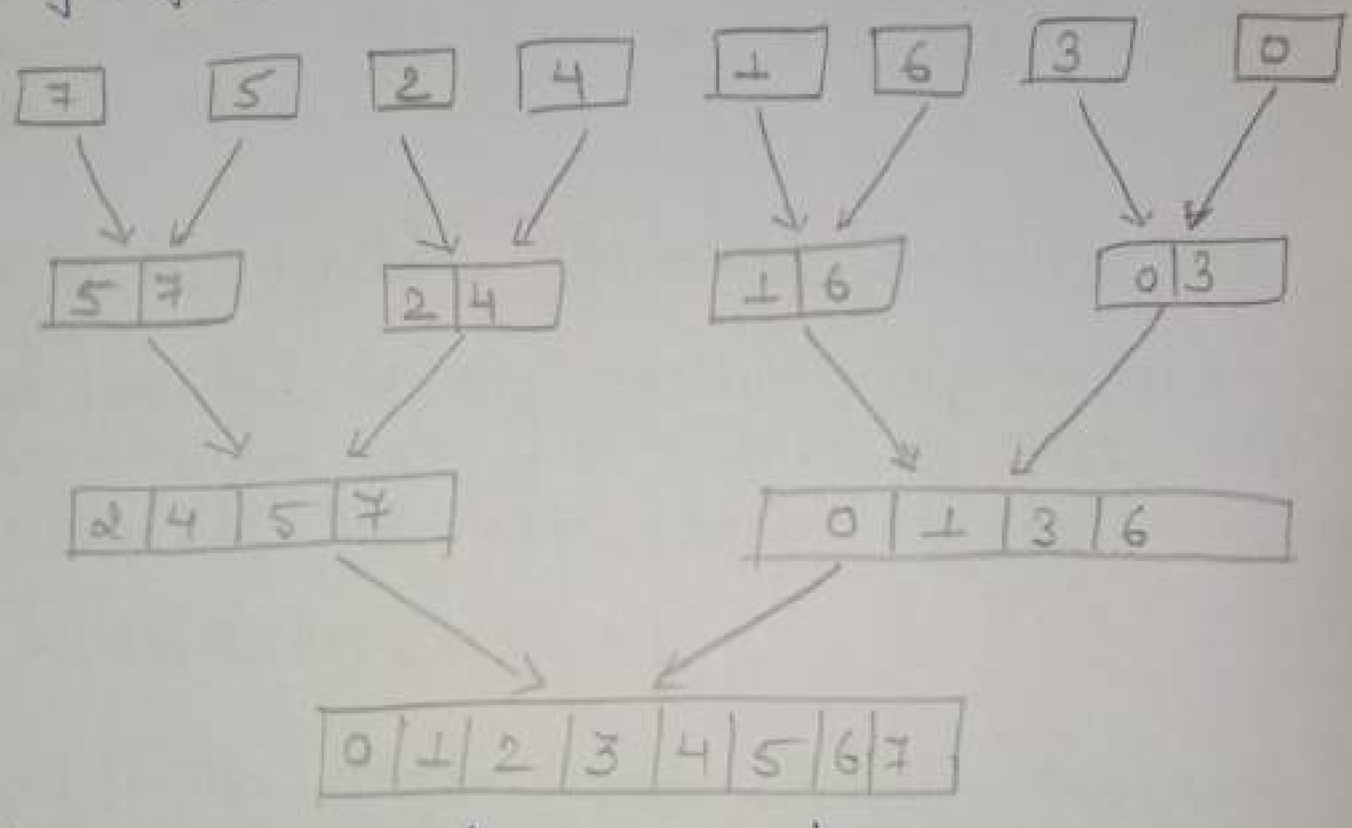


split

FUNCTIONS: MERGE (A, p, q, r)

1. n1 = q-p+1
2. n2 = r-p
3. Create arrays [1 … n1+1] and R [1 …. n2+1]
4. for i ← 1 to n1
5. do [i] ← A [p+i-1]
6. for j ← 1 to n2
7. do R[j] ← A[q+j]
8. L[n1+1] ← ∞
9. R[n2+1] ← ∞
10. i ← 1

11. J ← 1
12. For k ← p to r
13. Do if L[i] ≤ R[j]
14. then A[k] ← L[i]
15. i ← i+1
16. else A[k] ← R[j]
17. j ← j+1



The merge step of merge sort

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. merge sort is no different algorithm, just the fact here the merge step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists (arrays) to build one large sorted list (array).
The merge sort algorithm upholds three pointers, ie one for both of the two arrays and other one to preserve the final sorted array's current index

Merge () function Explained step-By-step.

Consider the following example of an unsorted array, which we are going to sort with the help of the merge sort algorithm.

A = (36, 25, 40, 2, 7, 80, 15)

Step 1:- The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

1-48

Step 2:- After dividing an array into two subarrays we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step 3:- Again, we will divide these arrays until we achieve an atomic value, i.e value that cannot be further divided.

Step 4:- Next, we will merge them back in the same way as they were broken down.

Step 5:- For each list, we will first compare the element and the combine them to form a new sorted list.

Step 6:- In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.

Analysis of Merge Sort:-

let T(n) be the total time taken by the merge sort algorithm.

- sorting two halves will take at the most $2T\frac{n}{2}$ times.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be.

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

But we ignore '-1' because the element will take some time to be copied in merge lists.

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

Putting $n = \frac{n}{2}$ in place of n in ... equation 1.

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$ .... equation 2

Put 2 equation in 1 equation

$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$

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

$T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2n$ ...... equation 3

Putting $n = \frac{n}{2^2}$ in equation 1

$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$ ... equation 4.

Putting 4 equation in 3 equation

$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

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

$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n$ ...... equation 5

from eq 1, eq 3, eq 5 ... we get

$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$ .... equation 6.

From Stopping Condition:-

$$\frac{n}{2^i} = 1 \text{ And } T\left(\frac{n}{2^i}\right) = 0$$

$$n = 2^i$$

Apply log both sides:-

$$\log n = \log 2^i$$

$$\log n = i \log 2$$

$$\frac{\log n}{\log 2} = i$$

$$\log_2 n = i$$

from 6 equation

$$T(n) = 2^i\, T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n . n$$

$$T(n) = n \log n.$$

If the time for the merging operation is proportional to n, then the computing time for merge sort described by the recurrence relation.

$$T(n) = \begin{cases} a & n=1,\ a \text{ is constant} \\ 2T\left(\frac{n}{2}\right)+cn & n>1 \\ & c \text{ is a constant.} \end{cases}$$

where n is a power of 2 i.e $n=2^k$ we can solve recurrence relation

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

$$= 2^2 T\left(\frac{n}{2^2}\right) + cn + cn$$

$$- - - - - -$$

$$- - - - - -$$

$$= 2^k T\left(\frac{n}{2^k}\right) + k\,cn \qquad \left\{ \begin{array}{l} \text{If } n = 2^k \\ k = \log_2 n \end{array} \right.$$

$$= a2^k + cn\log_2 n$$

$$= an + cn\log_2 n$$

$$T(n) = O(n \log_2 n)$$

Best case complexity :- The merge sort algorithm has a best case time complexity of $O(n * \log n)$ for the already sorted array.

Average case complexity :- The average-case time complexity for the merge sort algorithm is $O(n * \log n)$.

Worst case complexity :- The worst-case time complexity is also $O(n * \log n)$. which occurs when we sort the descending order of an array into the ascending order.

Space complexity :- The space complexity of merge sort is $O(n)$.

Merge sort Applications :- • Inversion count Problem • External sorting • E-Commerce applications

References :-

R1. :- Reference by Sartaj sahni
R2 :- Reference by Thomas H. Cormen.

Summary : Merge sort (student understand how to implement merge sort and how can merge concept is used for sorting).

Binary

Search

Topic: Binary search.
Objective: Student will learn binary search algorithm & when it is applicable.
Outcomes: Student will able to perform binary search

Binary search :-

1.) In Binary search technique, we search an element in a sorted array by recursively dividing the interval in half.

2.) Firstly, we take the whole array as an interval.

3.) If the Pivot element is less than the item in the middle of the interval, we discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.

4.) If the Pivot element is greater than the item in the middle of the interval, we discard the first ~~all~~ half of the list and work recursively on the second half by calculating the new beginning and middle element

5) Repeatedly check until the value is found or interval is empty.

Analysis :-

① Input :- An array A of size n, already sorted in the ascending or descending order.

② Output :- Analyze to search an element item in the sorted array of size n.

③ Logic :- Let T(n) = number of comparisons of an item with n elements in a sorted array.

- set BEG = 1 and END = n
- find mid = int$\left(\frac{beg + end}{2}\right)$
- Compare the search item with the mid item.

Case 1 :- item = A[mid], then LOC = mid, but it the best case and T(n) = 1

Case 2 :- Item $\neq$ A[mid], then we will split the array into two equal part of size $\frac{n}{2}$.

And again find the midpoint of the half sorted array and compare with search element.

Repeat the same process until a search element is found.

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

{Time to compare the search element with mid element, then with half of the selected half part of array}.

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right)+1$, Putting $\frac{n}{2}$ in place of n.

Then we get :-

$T(n) = \left(T\left(\frac{n}{2^2}\right)+1\right)+1$ ∴ By putting $T\frac{n}{2}$ in equation

$T(n) = T\left(\frac{n}{2^2}\right)+2$ —— (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right)+1$ .... Putting $\frac{n}{2}$ in place of n in eq 1.

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

$T(n) = T\left(\frac{n}{2^3}\right) + 3$ . . . . (equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$ . . . putting $\frac{n}{3}$ in place of n in eq

Put $T\left(\frac{2}{2^2}\right)$ in eq (3)

$T(n) = T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times.

$T(n) = T\left(\frac{n}{2^i}\right) + i$ . . . .

Stopping Condition : $T(1) = 1$

At least there will be only one term left that's why that term will compare out and only one comparison be done that's why $T(1) = 1$

Item Comparison.

Is the last term of the equation and it will be equal to 1

$\frac{n}{2^i} = 1$ [$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1]

$n = 2^i$

Applying log both sides

$\log n = \log 2^i$

$\log n = i \log 2$

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

$T(n) = T\left(\frac{n}{2^i}\right) + i$ $\boxed{\frac{n}{2^i} = 1 \text{ as in eq 5}}$

$= T(1) + i$

$= 1 + i$ .... $T(1) = 1$ by stopping condition

$= 1 + \log_2 n$

$= \log_2 n$ ..... (1 is a constant that's why ignore it)

Therefore, binary search is of order $O(\log_2 n)$.

The efficiency of Binary search :
As it cuts half part in each iteration its time complexity is O(log n).

Reference :-

R1 :- Reference by ~~Ell~~ Ellis Horowitz.

R2 :- Reference by Sartaj Sahni.

Summary : Binary search is more efficient than linear search. Condition is that data should be ordered list & access to mid position is directly possible.

Quick

Sort

Topic: Quick sort

Objective: student will learn the algorithm of quick sort.

Outcomes: student will able to sort element through quick sort.

Quick Sort :-

It is an algorithm of Divide & Conquer type.

Divide :- Rearrange the elements and split arrays into two subarrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub array is larger than the middle element.

Conquer :- Recursively, sort two sub arrays.

Combine :- Combine the already sorted array.

Algorithm :-

QUICKSORT (array A, int m, int n)

1.) if (n>m)

2) then

3) i ← a random index from [m,n]

4.) Swap A[i] with A[m]
5.) o ← PARTITION (A,m,n)
6.) QUICKSORT (A,m,o-1)
7.) QUICKSORT (A,o+1,n)

Partition Algorithm :-

Partition algorithm rearranges the sub arrays in a place.

PARTITION (array A, int m, int n)
1.) x ← A[m]
2.) o ← m
3.) for p ← m+1 to n
4.) do if (A[P] < x)
5.) then o ← o+1
6.) Swap A[o] with A[P]
7.) Swap A[m] with A[o]
8.) return o

Sort following elements using Quick sort:-

26, 5, 37, 1, 61, 11, 59, 15, 48, 19.

26, 5, 37, 1, 61, 11, 59, 15, 48, 19

↑ Pivot ↑i ↑j

swap

26, 5, 19, 1, 61, 11, 59, 15, 48, 37

↑ Pivot ↑i ↑j

swap

26, 5, 19, 1, 15, 11, 59, 61, 48, 37

↑ Pivot ↑j ↑i

swap

11, 5, 19, 1, 15 $\boxed{26}$ 59, 61, 48, 37

↑ Pivot ↑i ↑j

SL=1 SL=2

swap

$\boxed{11}$, 5, 1, 19, 15 $\boxed{26}$ 59, 61, 48, 37

↑ Pivot j i

swap

1, 5, $\boxed{11}$, 19, 15 $\boxed{26}$ 59, 61, 48, 37

SL3 SL4 SL2

1.63

$\boxed{1}$, 5, $\boxed{11}$ 19, 15 $\boxed{26}$ 59, 61, 48, 37.
pivot i
j

$\boxed{1}$ 5 $\boxed{11}$ 19 15 $\boxed{26}$ 59, 61, 48, 37
i j
pivot swap.

$\boxed{1}$ 5 $\boxed{11}$ 15 $\boxed{19}$ $\boxed{26}$ 59, 61, 48, 37
i j
pivot swap

$\boxed{1}$ 5 $\boxed{11}$ 15 $\boxed{19}$ $\boxed{26}$ 59 37 48 61
j i
pivot swap

$\boxed{1}$ 5 $\boxed{11}$ 15 $\boxed{19}$ $\boxed{26}$ 48 37 $\boxed{59}$ 61
SLS
i j
pivot swap

$\boxed{1}$ 5 $\boxed{11}$ 15 $\boxed{19}$ $\boxed{26}$ 37 $\boxed{48}$ $\boxed{59}$ 61.

P.T.O

Time Complexity of Quick sort.

$$T(n) = \begin{cases} T(i-1) + T(n-i) & \\ n & n > 1 \\ 1 & n = 1 \end{cases}$$

Best Case :- $i = n/2$ Pivot is median

$$T(n) = \begin{cases} 2T(n/2) + n & n > 1 \\ 1 & n = 1 \end{cases}$$

$$T(n) = O(n \log_2 n)$$

Worst Case :- $i = 1$

$$T(n) = \begin{cases} T(n-1) + n & n > 1 \\ 1 & n = 1 \end{cases}$$

$$T(n) = n + (n-1) + (n-2) + \dots + 1$$

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

$$= O(n^2)$$

1·65

Worst Case Analysis :- It is the case when items are already in sorted form and we try to sort them again. This will takes lots of time and space.

Equation :-

$$T(n) = T(1) + T(n-1) + n.$$

T(1) is time taken by pivot element

T(n-1) is time taken by remaining element except for pivot element

N is the number of comparisons required to identify the exact position of itself.

If we compare first element pivot with other, then there will be 5 comparisons.

It means there will be n comparisons if there are n items

~~It means there will be n comp.~~

1, 2, 3, 4, 5

T(1) Remaining [T(n-1)]

Relational formula for worst case :-

$T(n) = T(1) + T(n-1) + n \quad \dots \text{①}$

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

Put T(n-1) in equation 1.

[By putting (n-1) in place of n in equation 1]

$T(n) = T(1) + T(1) + (T(n-2) + (n-1) + n \quad \dots \text{(ii)}$

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

$T(n-2) = T(1) + T(n-3) + (n-2)$

Put T(n-2) in equation (ii) [By putting (n-2) in place of n in eq 1]

$T(n) = 2T(1) + T(1) + T(n-3) + (n-2) + (n-1) + n$

$T(n) = 3T(1) + T(n-3) + (n-2) + (n-1) + n$

$T(n-3) = T(1) + T(n-4) + n-3$

[By putting (n-3) in place of n in eq 1]

$T(n) = 3T(1) + T(1) + T(n-4) + (n-3) + (n-2) + (n-1) + n$

$= 4T(1) + T(n-4) + (n-3) + (n-2) + (n-1) + n \quad \dots \text{(iii)}$

$$T(n) = (n-1)\,T(1) + T(n-(n-1)) + (n-(n-2)) + (n-(n-3)) + (n-(n-4)) + n.$$

$$T(n) = (n-1)\,T(1) + T(1) + 2 + 3 + 4 + \cdots n$$

$$T(n) = (n-1)\,T(1) + T(1) + 2 + 3 + 4 + \cdots + n + 1 - 1.$$

[Adding 1 and subtracting 1 for making AP series]

$$T(n) = (n-1)\,T(1) + T(1) + 1 + 2 + 3 + 4 + \cdots + n - 1$$

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

stopping condition :- $T(1) = 0$

- Because at last there is only one element left and no comparison is required

$$T(n) = (n-1)(0) + 0 + \frac{n(n+1)}{2} - 1$$

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

[Avoid all the terms expect higher terms n^2]

$$T(n) = O(n^2)$$

worst case complexity of Quick sort is

$$T(n) = O(n^2)$$

Average Case :-

Generally, we assume the first element of the list as the pivot element.

In an average case, the number of chances to get a pivot element is equal to the number of items.

let total time taken = $T(n)$

for eg: In a given list

P1, P2, P3, P4 .... Pn.

If P1 is the pivot list then we have 2 lists ie $T(0)$ and $T(n-1)$

If P2 is the pivot list then we have 2 lists. ie $T(1)$ and $T(n-2)$

P1, P2, P3, P4 ... Pn.

If P3 is the pivot list then we have 2 lists. ie $T(2)$ and $T(n-3)$

P1, P2, P3, P4 ... Pn.

So in general if we taken the kth element to be the pivot element.

Then,

$$T(n) = \sum_{k=1}^{n} T(k-1) + T(n-k)$$

Pivot element will do n Comparison and we are doing average case so.

$$T(n) = \underbrace{n+1}_{\text{N Comparison}} + \underbrace{\frac{1}{n}\left(\sum_{k=1}^{n} T(k-1) + T(n-k)\right)}_{\text{Average of n elements}}$$

So Relational formula for Randomized Quick Sort. is :-

$$T(n) = n+1 + \frac{1}{n}\left(\sum_{k=1}^{n} T(k-1) + T(n-k)\right)$$

$$= n+1 + \frac{1}{n}\left(T(0) + T(1) + T(2) + \cdots T(n-1) + T(n-2) + T(n-3) + \cdots T(0)\right)$$

$$= n+1 + \frac{1}{n} \times 2\left(T(0) + T(1) + T(2) + \cdots T(n-2) + T(n-1)\right)$$

$$n\,T(n) = n(n+1) + 2\left(T(0) + T(1) + T(2) + \cdots T(n-1)\right) \cdots eq\,1$$

Put $n = n-1$ in eq 1

$$(n-1)\,T(n-1) = (n-1)\,n + 2\left(T(0) + T(1) + T(2) + \cdots T(n-2)\right) \cdots eq\,2$$

from eq1 & eq2.

$$n\,T(n) - (n-1)\,T(n-1) = n(n+1) - n(n-1) + 2(T(0) + T(1) + T(2) + ?T(n-2) + T(n-1)) - 2(T(0) + T(1) + T(2) + \dots T(n-2))$$

$$n\,T(n) - (n-1)\,T(n-1) = n\,[n+1-n+1] + 2T(n-1)$$

$$n\,T(n) = [2 + (n-1)]\,T(n-1) + 2n$$

$$nT(n) = n+1\,T(n-1) + 2n$$

$$\frac{n}{n+1}T(n) = \frac{2n}{n+1} + T(n-1) \qquad [\text{Divide by } n+1]$$

$$\frac{1}{n+1}T(n) = \frac{2}{n+1} + \frac{T(n-1)}{n} \qquad [\text{Divide by } n] \dots \text{eq3}$$

Put $n = n-1$ in eq 3

$$\frac{1}{n}T(n-1) = \frac{2}{n} + \frac{T(n-2)}{n-1} \quad \dots \text{eq 4}$$

Put 4 eq in 3q.

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1} \quad \dots \text{eq 5}$$

Put $n = n-2$ in eq 3

$$\frac{T(n-2)}{n-1} = \frac{2}{n-1} + \frac{2}{n} + \frac{T(n-3)}{n-2} \quad \dots \text{eq.6}$$

1.71

Put 6 eq in 5 eq.

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{T(n-3)}{n-2} \quad \ldots \text{ eq } 7$$

Put n=n-3 in eq 3

$$\frac{T(n-3)}{n-2} = \frac{2}{n-2} + \frac{T(n-4)}{n-3} \quad \ldots \text{ eq } 8$$

Put 8 eq. in 7 eq.

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \frac{T(n-4)}{n-3} \quad \ldots \text{ eq } 9$$

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

3 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} = T(n-3)$

4 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} = T(n-4)$

from 3 eq, 5 eq, 7 eq, 9 eq we get.

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3} + \frac{T(n-(n-1))}{n-(n-2)} \quad \cdots \text{ eq-10}$$

from 3eq $\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{T(n-1)}{n}$

put $n=1$

$$\frac{T(1)}{2} = \frac{2}{2} + \frac{T(0)}{1}$$

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

from 10 eq.

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3} + \frac{T(1)}{2}$$

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3} + 1$$

Multiply and divide the last term by 2.

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3} + 2 \times \frac{1}{2}$$

$$= 2\left[\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} + \frac{1}{n+1}\right]$$

$$= 2\sum^{n}_{2\le k\le n+1} \frac{1}{k} = 2\int_{2}^{n+1} \frac{1}{k}.$$

Multiply & divide k by n.

$$= 2\int_{2}^{n+1} \frac{\frac{1}{kn}}{n}$$

put $\frac{k}{n} = x$ and $\frac{1}{n} = dx$

$$\frac{T(n)}{n+1} = 2\int_{2}^{n+1} \frac{1}{x}\, dx.$$

[Note : $\int \frac{1}{x} dx = \log x$

$$= \frac{T(n)}{n+1}\ 2\log x \Big|_{2}^{n+1}$$

$$= 2\left[\log(n+1) - \log 2\right]$$

$$= T(n) = 2(n+1)\left[\log(n+1) - \log 2\right]$$

Igoring Constant we get

$T(n) = n\log n$

$T(n) = O(n\log n)$

3.> Quick sort [Best Case] :- In any Sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

Method Name	Equation	Stopping Condition.	Complexities.
1.> Worst Case.	$T(n)=T(n-1)+T(0)+n$	$T(1)=0$	$T(n)=n^2$
2.> Average case	$T(n)=n+1+\frac{1}{n}\left(\sum_{k=1}^{n} T(k-1)+T(n-k)\right)$		$T(n)=n\log n$.

References :-

R1 :- Reference by Ellis Horowitz.
R2 :- Reference by Thomas H. Cormen.

Summary: Quick sort is fastest algorithm but in worst case its time Complexity is very poor.

Strassen's

Matrix

Multiplication

Topic: Strassen's Matrix Multiplication.
Objective: student will learn matrix multiplication algorithm.
Outcomes: student will learn how the time complexity of algorithm will improve by this algorithm.

Strassen's Matrix Multiplication :-

- let A and B be two n x n matrix the Product matrix C = AB is also an n x n matrix

$$c(i,j) = \sum_{1 \le k \le n} A(i,k)\, B(k,j)$$

for all i & j between 1 and n with conventional method time complexity is $\theta(n^3)$

- The divide & conquer strategy suggests another way to compute Product of two n x n matrices.

Let $n = 2^k$ If is not in power of 2 than enough rows & columns of zeroes can be added to both A & B so that the resulting dimensions are a Power of two. Imagine that A & B are each are partitioned in to four square submatrices each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$. The Product AB can be computed by using above formula.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$
$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$
$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$
$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

$n/2 \times n/2$ matrices can be added in time cn^2 for some constant c, the overall computing time T(n) of the resulting divide & conquer algorithm is given by recurrence

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

then $T(n) = O(n^3)$

So, no improvement obtained.

Volker strassen has found a way which requires 7 multiplication & 18 addition or substraction.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

$$T(n) = \begin{cases} b & n <= 2 \\ 7\,T(n/2) + an^2 & n > 2 \end{cases}$$

$$T(n) = an^2\left[1 + \left(\frac{7}{4}\right) + \left(\frac{7}{4}\right)^2 + \cdots + \left(\frac{7}{4}\right)^{k-1}\right] + 7^k T(1)$$

$$\leq cn^2\left(\frac{7}{4}\right)^{\log_2 n} + 7^{\log_2 n}$$

$$= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7}$$

$$= O(n^{\log_2 7})$$

$$\approx O(n^{2.81})$$

Summary: strassen's matrix multiplication reduce the time complexity by fraction.