UNIT II

DIVIDE AND CONQUER METHOD AND GREEDY METHOD

2 marks

1. What is brute force algorithm?

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved.

2. List the strength and weakness of brute force algorithm.

Strengths

a. wide applicability,
b. simplicity
c. yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

Weaknesses

a. rarely yields efficient algorithms
b. some brute-force algorithms are unacceptably slow not as constructive as some other design techniques

3. What is exhaustive search?

A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

4. Give the general plan of exhaustive search.

Method:

- generate a list of all potential solutions to the problem in a systematic manner
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

5. Give the general plan for divide-and-conquer algorithms.

The general plan is as follows

- A problems instance is divided into several smaller instances of the same problem, ideally about the same size
- The smaller instances are solved, typically recursively
- If necessary the solutions obtained are combined to get the solution of the original problem

Given a function to compute on 'n' inputs the divide-and-comquer strategy suggests splitting the inputs in to'k' distinct susbsets, 1<k <n, yielding 'k' subproblems. The subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and conquer strategy can possibly be reapplied.

6. List the advantages of Divide and Conquer Algorithm

Solving difficult problems, Algorithm efficiency, Parallelism, Memory access, Round off control.

7. Define of feasibility

A feasible set (of candidates) is promising if it can be extended to produce not merely a solution, but an optimal solution to the problem.

8. Define Hamiltonian circuit.

A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

9. State the Master theorem and its use.

If $f(n) \in \theta(n^d)$ where $d \geq 0$ in recurrence equation $T(n) = aT(n/b)+f(n)$, then

$$T(n) \in \begin{cases} \theta(n^d) & \text{if } a<b^d \\ \theta(n^d \log n) & \text{if } a=b^d \\ \theta(n^{\log_b a}) & \text{if } a>b^d \end{cases}$$

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the use of Master theorem.

10. What is the general divide-and-conquer recurrence relation?

An instance of size 'n' can be divided into several instances of size n/b, with 'a' of them needing to be solved. Assuming that size 'n' is a power of 'b', to simplify the analysis, the following recurrence for the running time is obtained:

$$T(n) = aT(n/b)+f(n)$$

Where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

11. Define mergesort.

Mergesort sorts a given array A[0..n-1] by dividing it into two halves a[0..(n/2)-1] and A[n/2..n-1] sorting each of them recursively and then merging the two smaller sorted arrays into a single sorted one.

12. List the Steps in Merge Sort

1. Divide Step: If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.

2. Recursion Step: Recursively sort array A1 and A2.

3. Conquer Step: Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence

13. List out Disadvantages of Divide and Conquer Algorithm

- Conceptual difficulty
- Recursion overhead
- Repeated subproblems

14. Define Quick Sort

Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good \"general purpose\" sort and it consumes relatively fewer resources during execution.

15. List out the Advantages in Quick Sort

- It is in-place since it uses only a small auxiliary stack.
- It requires only n log(n) time to sort n items.
- It has an extremely short inner loop
- This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

16. List out the Disadvantages in Quick Sort

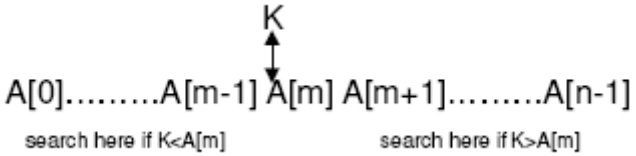
- It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (i.e., n2) time in the worst-case.
- It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

17. What is the difference between quicksort and mergesort?

Both quicksort and mergesort use the divide-and-conquer technique in which the given array is partitioned into subarrays and solved. The difference lies in the technique that the arrays are partitioned. For mergesort the arrays are partitioned according to their position and in quicksort they are partitioned according to the element values.

18. **What is binary search?**

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the arrays middle element A[m]. If they match the algorithm stops; otherwise the same operation is repeated recursively for the first half of the array if K < A[m] and the second half if K > A[m].



19. List out the 4 steps in Strassen's Method?

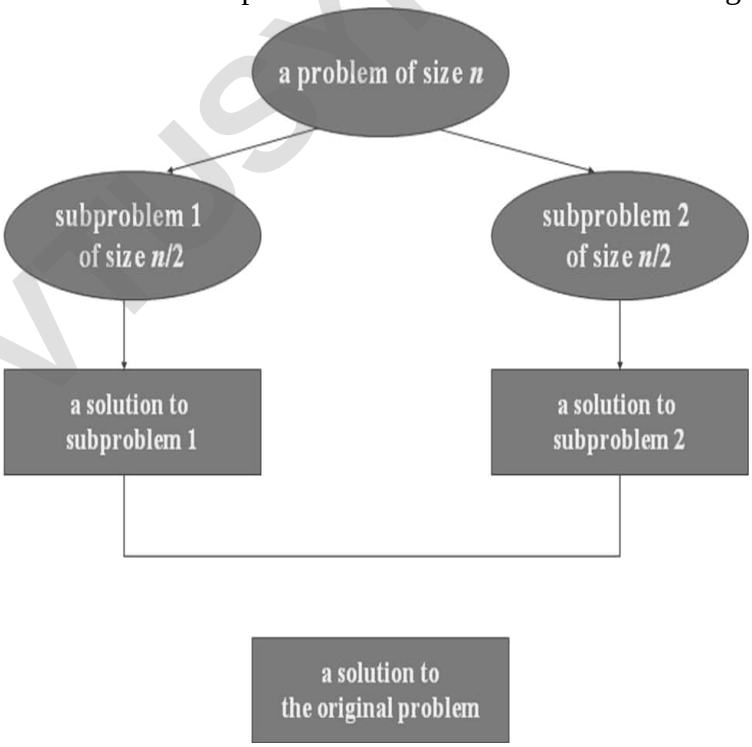
1. Divide the input matrices A and B into n/2 * n/2 submatrices, as in equation (1).
2. Using Θ(n2) scalar additions and subtractions, compute 14 n/2 * n/2 matrices A1, B1, A2, B2, …, A7, B7.
3. Recursively compute the seven matrix products Pi =AiBi for i =1, 2, 7.
4. Compute the desired submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the Pi matrices, using only Θ(n2) scalar additions and subtractions.

16 marks

1. **Explain Divide And Conquer Method**

✓ The most well known algorithm design strategy is Divide and Conquer Method. It

- Divide the problem into two or more smaller subproblems.
- Conquer the subproblems by solving them recursively.
- Combine the solutions to the subproblems into the solutions for the original problem.



✓ Divide and Conquer Examples

- Sorting: mergesort and quicksort
- Tree traversals

- Binary search
- Matrix multiplication-Strassen's algorithm

2. Explain Merge Sort with suitable example.

✓ Merge sort definition**.**

Mergesort sorts a given array A[0..n-1] by dividing it into two halves a[0..(n/2)-1] and A[n/2..n-1] sorting each of them recursively and then merging the two smaller sorted arrays into a single sorted one.

✓ Steps in Merge Sort

1. Divide Step

If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.

2. Recursion Step

Recursively sort array A1 and A2.

3. Conquer Step

Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence

✓ Algorithm for merge sort.

ALGORITHM *Mergesort*(A[0..n-1])
//Sorts an array A[0..n-1] by recursive mergesort
//Input: An array A[0..n-1] of orderable elements
//Output: Array A[0..n-1] sorted in nondecreasing order
if n > 1
copy A[0..(n/2)-1] to B[0..(n/2)-1]
copy A[(n/2)..n-1] to C[0..(n/2)-1]
Mergesort(B[0..(n/2)-1])
Mergesort(C[0..(n/2)-1])
Merge(B,C,A)

✓ Algorithm to merge two sorted arrays into one.

ALGORITHM Merge (B [0..p-1], C[0..q-1], A[0..p+q-1])
//Merges two sorted arrays into one sorted array
//Input: arrays B[0..p-1] and C[0..q-1] both sorted
//Output: sorted array A [0..p+q-1] of the elements of B & C
I 0; j 0; k 0
while I < p and j < q do
if B[I] <= C[j]
A[k] B [I]; I I+1
else
A[k] C[j]; j j+1
k k+1
if i = p
copy C[j..q-1] to A [k..p+q-1]
else
copy B[i..p-1] to A [k..p+q-1]

3. Discuss Quick Sort

✓ Quick Sort definition

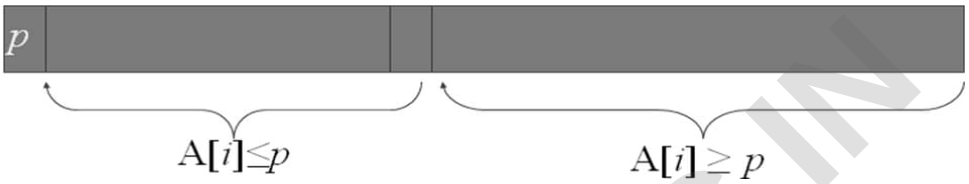
Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good \"general purpose\" sort and it consumes relatively fewer resources during execution.

✓ Quick Sort and divide and conquer

- Divide: Partition array A[l..r] into 2 subarrays, A[l..s-1] and A[s+1..r] such that each element of the first array is ≤A[s] and each element of the second array is ≥ A[s]. (Computing the index of s is part of partition.)
- Implication: A[s] will be in its final position in the sorted array.
- Conquer: Sort the two subarrays A[l..s-1] and A[s+1..r] by recursive calls to quicksort
- Combine: No work is needed, because A[s] is already in its correct place after the partition is done, and the two subarrays have been sorted.

✓ Steps in Quicksort

- Select a pivot w.r.t. whose value we are going to divide the sublist. (e.g., p = A[l])
- Rearrange the list so that it starts with the pivot followed by a ≤ sublist (a sublist whose elements are all smaller than or equal to the pivot) and a ≥ sublist (a sublist whose elements are all greater than or equal to the pivot) Exchange the pivot with the last element in the first sublist(i.e., ≤ sublist) – the pivot is now in its final position
- Sort the two sublists recursively using quicksort.



✓ The Quicksort Algorithm

ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: A subarray A[l..r] of A[0..n-1],defined by its left and right indices l and r
//Output: The subarray A[l..r] sorted in nondecreasing order
if l < r
    s ← Partition (A[l..r]) // s is a split position
    Quicksort(A[l..s-1])
    Quicksort(A[s+1..r]

ALGORITHM Partition (A[l ..r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n-1], defined by its left and right indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as this function's value
P ←A[l]
i ←l; j ← r + 1;
Repeat
    repeat i ← i + 1 until A[i]>=p //left-right scan
    repeat j ←j – 1 until A[j] <= p//right-left scan
    if (i < j) //need to continue with the scan
        swap(A[i], a[j])
until i >= j //no need to scan
swap(A[l], A[j])
    return j

✓ Advantages in Quick Sort

• It is in-place since it uses only a small auxiliary stack.
• It requires only n log(n) time to sort n items.
• It has an extremely short inner loop

• This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

✓ Disadvantages in Quick Sort

• It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
• It requires quadratic (i.e., n2) time in the worst-case.
• It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

✓ Efficiency of Quicksort

Based on whether the partitioning is balanced.

Best case: split in the middle — $\Theta(n \log n)$

C(n) = 2C(n/2) + Θ(n) //2 subproblems of size n/2 each

Worst case: sorted array! — $\Theta(n^2)$

C(n) = C(n-1) + n+1 //2 subproblems of size 0 and n-1 respectively

Average case: random arrays — $\Theta(n \log n)$

4. Explain Binary Search.

✓ **Binary Search –Iterative Algorithm**

ALGORITHM BinarySearch(A[0..n-1], K)

```
//Implements nonrecursive binary search
//Input: An array A[0..n-1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//       or –1 if there is no such element
l ← 0, r ← n – 1
while l ≤ r do                    //l and r crosses over→ can't find K.
    m ←⌊(l + r) / 2⌋
    if K = A[m] return m          //the key is found
    else
        if K < A[m] r← m – 1//the key is on the left half of the array
        else    l ← m + 1     // the key is on the right half of the array
return -1
```

✓ **Binary Search – a Recursive Algorithm**

ALGORITHM BinarySearchRecur(A[0..n-1], l, r, K)

```
if l > r              //base case 1: l and r cross over→ can't find K
    return –1
else
   m ←⌊(l + r) / 2⌋
    if K = A[m]           //base case 2: K is found
        return m
    else                  //general case: divide the problem.
        if K < A[m]   //the key is on the left half of the array
            return BinarySearchRecur(A[0..n-1], l, m-1, K)
        else          //the key is on the left half of the array
            return BinarySearchRecur(A[0..n-1], m+1, r, K)
```

✓ **Binary Search – Efficiency**

- recurrence relation
  C(n) = C(⌊n / 2⌋) + 2
- Efficiency

$C(n) \in \Theta(\log n)$

5. **Explain Strassen's Algorithm**
   - ✓ **Multiplication of Large Integers**
     - Multiplication of two n-digit integers.
     - Multiplication of a m-digit integer and a n-digit integer (where n > m) can be modeled as the multiplication of 2 n-digit integers (by padding n – m 0s before the first digit of the m-digit integer)

     - **Brute-force algorithm**

$$\begin{pmatrix} \mathbf{c_{00}} & \mathbf{c_{01}} \\ \mathbf{c_{10}} & \mathbf{c_{11}} \end{pmatrix} = \begin{pmatrix} \mathbf{a_{00}} & \mathbf{a_{01}} \\ \mathbf{a_{10}} & \mathbf{a_{11}} \end{pmatrix} * \begin{pmatrix} \mathbf{b_{00}} & \mathbf{b_{01}} \\ \mathbf{b_{10}} & \mathbf{b_{11}} \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{a_{00} * b_{00} + a_{01} * b_{10}} & \mathbf{a_{00} * b_{01} + a_{01} * b_{11}} \\ \mathbf{a_{10} * b_{00} + a_{11} * b_{10}} & \mathbf{a_{10} * b_{01} + a_{11} * b_{11}} \end{pmatrix}$$

     - 8 multiplications
     - 4 additions
     - Efficiency class: Θ (n3)
   - ✓ Strassen's Algorithm (two 2x2 matrices)

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

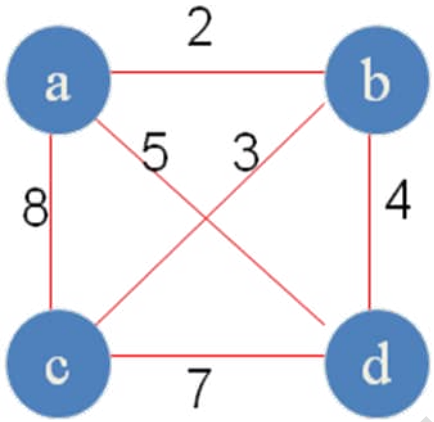
$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11}) * b_{00}$
$m_3 = a_{00} * (b_{01} - b_{11})$
$m_4 = a_{11} * (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) * b_{11}$
$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

- Efficiency of Strassen's Algorithm
  - If n is not a power of 2, matrices can be padded with rows and columns with zeros
  - Number of multiplications
  - Number of additions
  - Other algorithms have improved this result, but are even more complex

6. **Explain in detail about Travelling Salesman Problem using exhaustive search.**
   Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city

Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph

Example :



Tour	Cost
a→b→c→d→a	**2+3+7+5 = 17**
a→b→d→c→a	**2+4+7+8 = 21**
a→c→b→d→a	**8+3+4+5 = 20**
a→c→d→b→a	**8+7+4+2 = 21**
a→d→b→c→a	**5+4+3+8 = 20**
a→d→c→b→a	**5+7+3+2 = 17**

Efficiency: Θ((n-1)!)

7. Explain in detail about knapsack problem.

Given *n* items:

weights: w_1 w_2 ... w_n

values: v_1 v_2 ... v_n

a knapsack of capacity *W*

Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=16

item	**weight**	**value**
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60

{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Efficiency: **Θ(2^n)**

8. Explain in detail about closest pair problem.

Find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane).

Brute-force algorithm

Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n $(n \geq 2)$ points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$

//Output: Indices *index*1 and *index*2 of the closest pair of points

$dmin \leftarrow \infty$

for $i \leftarrow 1$ **to** $n - 1$ **do**

  for $j \leftarrow i + 1$ **to** n **do**

    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //*sqrt* is the square root function

    if $d < dmin$

      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$

return *index*1, *index*2

Efficiency: Θ(n^2) multiplications (or sqrt)