|  |  |
| --- | --- |
| **BCSPC3020** | **Design and Analysis of Algorithms – KEY SET 1** |

**PART – A: (Multiple Choice Questions) (1 x 10 = 10 Marks)**

**Q(1)** a.4 b. 2 c.3 d.3 e.4 f. 3 g.3 h.2 i.1 j.1

**PART – B (Short Answer Questions) (2 x 10 = 20 Marks)**

**Q(2)**

**a.** *‘*’a set of steps to accomplish or complete a task that is described precisely enough that a

computer can run it’’.

**Described precisely**: very difficult for a machine to know how much water, milk to be added

etc. in the above tea making algorithm.

Correctness:- Correct: Algorithms must produce correct result. Produce an incorrect answer:Even if it fails to give correct results all the time still there is a control on how often it gives wrong result. (Used in RSA algorithm): It doesn’t give correct answer all the time.1 out

 Approximation algorithm: Exact solution is not found, but near optimal solution can

be found out. (Applied to optimization problem.)

· Less resource usage:

Algorithms should use less resources (time and space).

**b.** The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

1. How long the algorithm takes :-will be represented as a function of the size of the input.

f(n)→how long it takes if ‘n’ is the size of input.

2. How fast the function that characterizes the running time grows with the input

size. “Rate of growth of running time”. The algorithm with less rate of growth of running time is considered better.

**c,** *Union*: Join two subsets into a single subset. In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find*(x) returns the representative of the set that *x* belongs to, and *Union* takes two set representatives as its arguments.

**d.** Let *G* = (*V*, *E*) be the graph where V is the set of vertices, E is the set of edges and |V|= *n*. The spanning tree *G*¢= (*V*, *E*¢) is a sub graph of *G* in which all the vertices of graph *G* are connected with minimum number of edges. The minimum number of edges required to correct all the vertices of a graph *G* in *n* – 1. Spanning tree plays a very important role in designing efficient algorithms.



**e.** Given a sorted array keys[0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts, where freq[i] is the number of searches to keys[i]. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

**f.** The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. It was invented by mathematician named Richard Bellman inn 1950s. The DP in closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. The DP differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these sub problems.

**g.** Let there are *n* number of objects and each object is having a weight and contribution to profit. The

knapsack of capacity *M* is given. The objective is to fill the knapsack in such a way that profit shall be maximum. We allow a fraction of item to be added to the knapsack.



Where *pi* and *wi* are the profit and weight of *ith* object and *xi* is the fraction of *ith* object to beselected.

**h.** [Implementation of 0/1 Knapsack](https://www.geeksforgeeks.org/branch-and-bound-set-2-implementation-of-01-knapsack/)

[Job Assignment Problem](https://www.geeksforgeeks.org/branch-bound-set-4-job-assignment-problem/)

[N Queen Problem](https://www.geeksforgeeks.org/branch-and-bound-set-4-n-queen-problem/)

[Traveling Salesman Problem](https://www.geeksforgeeks.org/branch-bound-set-5-traveling-salesman-problem/)

**i.** In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in case of **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs. In fact non-deterministic algorithms can’t solve the problem in polynomial time and can’t determine what is the next step. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.

**j. NP**-**complete** problem, any of a class of computational problems for which no efficient solution algorithm has been found. ... Many significant computer-science problems belong to this class—**e.g.**, the traveling salesman problem, satisfiability problems, and graph-covering problems.

**PART – C: (Long Answer Questions) (10 x 4 = 440 Marks)**

**3 (a)**

[Amortized Analysis](http://en.wikipedia.org/wiki/Amortized_analysis) is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a Particular expensive operation. The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

**1)**Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

**2)** The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called [***Accounting Method***](http://en.wikipedia.org/wiki/Accounting_method)and [***Potential Method***](http://en.wikipedia.org/wiki/Potential_method). We will be discussing the other two methods in separate posts.

**3)**The amortized analysis doesn’t involve probability. There is also another different notion of average-case running time where algorithms use randomization to make them faster and expected running time is faster than the worst-case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

**3 (b)**

##### BINARY SEARCH

1. Algorithm Bin search(a,n,x)
2. // Given an array a[1:n] of elements in non-decreasing
3. //order, n>=0,determine whether ‘x’ is present and
4. // if so, return ‘j’ such that x=a[j]; else return 0.
5. {
6. low:=1; high:=n;
7. while (low<=high) do
8. {
9. **mid:=[(low+high)/2****];**
10. if (x<a[mid]) then high;
11. else if(x>a[mid]) then

 low=mid+1;

1. else return mid;
2. }
3. return 0;
4. }

**3 (c)**

**Backtracking Algorithm for Subset Sum**Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.Assume given set of 4 elements, say **w[1] … w[4]**. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



**3 (d)**

A [*disjoint-set data structure*](http://en.wikipedia.org/wiki/Disjoint-set_data_structure) is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A [*union-find algorithm*](http://en.wikipedia.org/wiki/Disjoint-set_data_structure) is an algorithm that performs two useful operations on such a data structure: ***Find:*** Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
***Union:*** Join two subsets into a single subset.
In this post, we will discuss the application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.
*Union-Find Algorithm* can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [algorithm to detect cycle](http://www.geeksforgeeks.org/archives/18516). This is another method based on *Union-Find*. This method assumes that the graph doesn’t contain any self-loops.

**4 (a)**

**Algorithm**: Partition the array a[m:p-1] about a[m]

1. Algorithm Partition(a,m,p)
2. //within a[m],a[m+1],…..,a[p-1] the elements
3. // are rearranged in such a manner that if
4. //initially t=a[m],then after completion
5. //a[q]=t for some q between m and
6. //p-1,a[k]<=t for m<=k<q, and
7. //a[k]>=t for q<k<p. q is returned
8. //Set a[p]=infinite.
9. {
10. v=a[m];I=m;j=p;
11. repeat
12. {
13. repeat
14. I=I+1;
15. until(a[I]>=v);
16. repeat
17. j=j-1;
18. until(a[j]<=v);
19. if (I<j) then interchange(a,i.j);
20. }until(I>=j);
21. a[m]=a[j]; a[j]=v;
22. retun j;
23. }

 **Best case**

• Occurs when the subarrays are completely balanced every time.

• Each subarray has ≤ *n/*2 elements.

• Get the recurrence

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

*O*(log *n*) for keepingtrack of the recursion). A sorting algorithm is *stable* if duplicate elements

remain in the same relativeposition after sorting.

**Slow Algorithms:** Include BubbleSort, InsertionSort, and SelectionSort. These are all simple

Θ (*n*2)in-place sorting algorithms. BubbleSort and InsertionSort can be implemented as stable

algorithms,but SelectionSort cannot (without significant modifications).

**4 (b)**

Consider-

* Knapsack weight capacity = w
* Number of items each having some weight and value = n
* n = 4
* w = 5 kg
* (w1, w2, w3, w4) = (2, 3, 4, 5)
* (b1, b2, b3, b4) = (3, 4, 5, 6)
*

  

**4 (c)**

**let** dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)

**for each** edge (*u*, *v*) **do**

 dist[*u*][*v*] ← w(*u*, *v*) *// The weight of the edge (*u*,* v*)*

**for each** vertex *v* **do**

 dist[*v*][*v*] ← 0

**for** *k* **from** 1 **to** |V|

 **for** *i* **from** 1 **to** |V|

 **for** *j* **from** 1 **to** |V|

 **if** dist[*i*][*j*] > dist[*i*][*k*] + dist[*k*][*j*]

 dist[*i*][*j*] ← dist[*i*][*k*] + dist[*k*][*j*]

 **end if**

**4 (d)**

***1.****Sort all the edges in non-decreasing order of their weight.****2.****Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.****3.****Repeat step#2 until there are (V-1) edges in the spanning tree.*



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.



**5 (a)**

**breadth-first search** (**BFS**) is a strategy for searching in a graph when search is

limited to essentially two operations:

(a) visit and inspect a node of a graph;

(b) gain access to visit the nodes that neighbor the currently visited node.

The BFS begins at a root node and inspects all the neighboring nodes.

Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which

were unvisited, and so on.

Compare BFS with the equivalent, but more memory-efficient.

Any example can discuss.

**5 (b) Lower Bounds for Comparison-Based Sorting:**

Can we sort faster than *O*(*n* log *n*) time?

We will give anargument that if the sorting algorithm is based solely on making comparisons

between the keys in thearray, then it is impossible to sort more efficiently than (*n* log *n*) time.

Such an algorithm is called a*comparison-based* sorting algorithm, and includes all of the

algorithms given above.Virtually all known general purpose sorting algorithms are based on

making comparisons, so this isnot a very restrictive assumption. This does not preclude the

possibility of a sorting algorithm whoseactions are determined by other types of operations, for

example, consulting the individual bits ofnumbers, performing arithmetic operations, indexing

into an array based on arithmetic operations onkeys.We will show that any *comparison-based*

sorting algorithm for a input sequence *ha*1*; a*2*; : : : ; ani*must

make at least (*n* log *n*) comparisons in the worst-case. This is still a difficult task if you think

about it.It is easy to show that a problem *can* be solved fast (just give an algorithm). But to show

that a problem*cannot* be solved fast you need to reason in some way about all the possible

algorithms that might everbe written. In fact, it seems surprising that you could even hope to

prove such a thing. The catch hereis that we are limited to using comparison-based algorithms,

and there is a clean mathematical way ofcharacterizing all such algorithms.

**5 (c)** The Greedy approach constructs the solution through a sequence of steps. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

The choice of each step is a greedy approach is done based in the following:

· It must be feasible

· It must be locally optimal

· It must be unalterable

Therefore, the solution is said to be a feasible solution if it satisfies the following

constraints.

(i) **Explicit constraints**: - The elements of the output set must be taken from the input set.

(ii) **Implicit constraints**:-The objective function defined in the problem.

**5 (d)** In computing, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–**

**find set**, is a data structure that keeps track of a set of elements partitioned into a number

of disjoint (non-overlapping) subsets.

It supports the following useful operations:

· *Find*: Determine which subset a particular element is in. *Find* typically returns an item

from this set that serves as its "repr esentative"; by comparing the result of

two *Find* operations, one can determine whether two elements are in the same subset.

· *Union*: Join two subsets into a single subset.

· *Make Set*, which makes a set containing only a given element (a singleton), is generally

trivial. With these three operations, many practical partitioning problems can be solved.

In order to define these operations more precisely, some way of representing the sets is needed.

One common approach is to select a fixed element of each set, called its *representative*, to

represent the set as a whole. Then, *Find*(x) returns the representative of the set that *x* belongs to,

and *Union* takes two set representatives as its arguments.

**6 (a)**

A **deterministic algorithm** is simply an **algorithm** that has a predefined output. For instance if you are sorting elements that are strictly ordered(no equal elements) the output is well defined and so the **algorithm** is **deterministic**.

 In fact most of the computer **algorithms** are **deterministic**

* Given the same input, produces the same output every time.
* Given the same input, takes the same amount of time/memory/resources every time it is run.
* Problems of complexity class P that can be solved in polynomial time by a deterministic computer, as opposed to problems of complexity class NP which can be only solved in polynomial time using a *non-deterministic computer*.

The most simple deterministic algorithm is this [random number generator](http://xkcd.com/221/).

def random():

 return 4 #chosen by fair dice roll, guaranteed to be random

It gives the same output every time, exhibits known O(1) time and resource usage, and executes in PTIME on any computer.

deterministic algorithm that checks whether a given number is odd:

function is\_odd(n):

 if n mod 2 = 1

 then return true

 else return false

**6 (b)**

For example, consider the following binary tree. The smallest vertex cover is {20, 50, 30} and size of the vertex cover is 3.



The idea is to consider following two possibilities for root and recursively for all nodes down the root.
***1) Root is part of vertex cover:***In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).

***2) Root is not part of vertex cover:*** In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).

**6 (C)**

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

**RABIN-KARP-MATCHER (T, P, d, q)**

 1. n ← length [T]

 2. m ← length [P]

 3. h ← dm-1 mod q

 4. p ← 0

 5. t0 ← 0

 6. for i ← 1 to m

 7. do p ← (dp + P[i]) mod q

 8. t0 ← (dt0+T [i]) mod q

 9. for s ← 0 to n-m

 10. do if p = ts

 11. then if P [1.....m] = T [s+1.....s + m]

 12. then "Pattern occurs with shift" s

 13. If s < n-m

 14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q

**6 (d)**

The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard.

“The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.”

Ensure the following:

1. No two queens share a column

2. No two queens share a row

3. No two queens share a diagonal

4. No two queens share a top-right to left-bottom diagonal

1. For the 1st Queen, there are total 8 possibilities as we can place 1st Queen in any row of first column. Let’s place Queen 1 on row 3.
2. After placing 1st Queen, there are 7 possibilities left for the 2nd Queen. But wait, we don’t really have 7 possibilities. We cannot place Queen 2 on rows 2, 3 or 4 as those cells are under attack from Queen 1. So, Queen 2 has only 8 – 3 = 5 valid positions left.
3. After picking a position for Queen 2, Queen 3 has even fewer options as most of the cells in its column are under attack from the first 2 Queens.

We need to figure out an efficient way of keeping track of which cells are under attack. In previous solution we kept an 8­-by­-8 Boolean matrix and update it each time we placed a queen, but that required linear time to update as we need to check for safe cells.

--------------------------------- END--------------------------------------------------------------------------