**Analysis of Algorithms Questions and Solutions**

***Along with the lecture notes on E-learning consider these questions and answer to understand the exam structure:***

1. Consider the following Brute-force algorithm that find a match between a pattern P of length m in a text T of a length n such that m is smaller than n? [8 marks, 2 for each point]

**ALGORITHM** *BruteForceStringMatch(T* [0*..n* − 1]*, P*[0*..m* − 1]*)*

//Input: An array *T* [0*..n* − 1] of *n* characters representing a text and

// an array *P*[0*..m* − 1] of *m* characters representing a pattern

//Output: The index of the first character in the text that starts a

// matching substring or −1 if the search is unsuccessful

**for** *i* ←0 **to** *n* − *m* **do**

*j* ←0

**while** *j <m***and** *P*[*j* ]= *T* [*i* + *j* ] **do**

*j* ←*j* + 1

**if** *j* = *m* **return** *i*

**return** −1

If the average case scenario is when:

*T* = “N O B O D Y \_ N O T I C E D \_ H I M”

P = “**N** O T”

1. What is the best case scenario of this algorithm, give an example?
2. Show how to compute the complexity of the best case scenario.
3. What is the worst case scenario of this algorithm, give an example?
4. Show how to compute the complexity of the worst case scenario?

***Solution***:

1. The best case scenario of this algorithm is when the pattern P is “N O B”
2. The complexity of best case is O(1)
3. The worst case scenario of this algorithm is when the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n − m + 1 tries. This is when T = “M M M M M M M M M M M M M M M M M M M” and P=”M M T”
4. The complexity of the worst case scenario that make the algorithm makes m(n − m + 1) character comparisons, which puts it in theO(nm) class.
5. Set up and solve a recurrence relation for the number of times the following algorithm’s basic operation is executed. It computes the sum of the first n cubes: [6 marks]

S(n) = 13 + 23 + ... + n3.

Algorithm S(n)

//Input: A positive integer n

//Output: The sum of the first n cubes

if n = 1 return 1

else return S(n − 1) + n ∗ n ∗ n

***Solution*:**

The recurrence relation

C(n) = C(n − 1) + 1,

C(0) = 1 (there is a call but no multiplications when n = 0).

The solution of the recurrence

C(n) = C(n − 1) + 1

= [C(n − 2) + 1] + 1 = C(n − 2) + 2 = ...

= C(n − i) + i = ...

= C(0) + n

= 1+n.

1. State the Master Theorem then use it to find the O notation for the following recurrence relations. [6 marks]

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

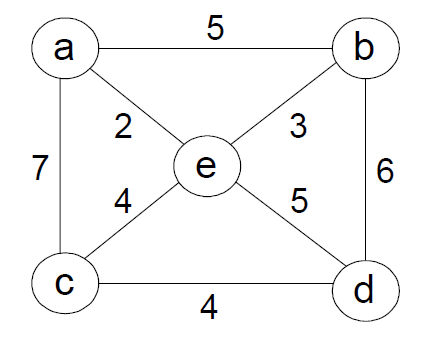
***Solution***:

f(n) = a f(n/b) + cnd

*a =* 4, *b* = 2, d=2 ⇒ a = 4 ? bd = 22=4 🡺 equal

Case 2: T(n) = O(n2 log2 n)

1. Use Prim algorithm to find the minimum spanning tree of the following graph, clarify the steps of the algorithm in your solution. [10 marks]



***Solution***

Prim Algorithm

*T* = ϕ; //Set of selected edges

*S =* ϕ; //Set of selected vertexes in MST

For every vertex *v* in *V*; key(*v*) = ∞; initial key value ∞

Pick random vertex *a* in *V*; key(*a*)= 0; initial key value 0

while (*S*≠*V*){

Get vertex *u* of minimum key value in *V*, such that *u*∉*S*

*T* = *T* ⋃ {(*u*, *a*)}, such that *a*∈*S* && weight(*u-a*)=key(*u)*

*S* = *S* ⋃{*u*}, and remove *u* from *V*

For every vertex *v* connected to *u* such that *v*∉*S*

If weight(*u*-*v*) weight is less than key(*v*)

Then key(*v*) = weight(*u*-*v*)

Tree vertices Priority queue of remaining vertices

a(-,-) b(a,5) c(a,7) d(a,∞) e(a,2)

e(a,2) b(e,3) c(e,4) d(e,5)

b(e,3) c(e,4) d(e,5)

c(e,4) d(c,4)

d(c,4)

The minimum spanning tree found by the algorithm comprises the edges

ae, eb, ec, and cd.

1. Consider the following Brute-force algorithm that finds distance between two closest points in the plane? [8 marks, 4 marks for each]

**ALGORITHM** *BruteForceClosestPair(P)*

//Input: A list *P* of *n (n* ≥ 2*)* points *p*1*(x*1*, y*1*), . . . , pn(xn, yn)*

//Output: The distance between the closest pair of points

*d*←∞

**for** *i* ←1 **to** *n* − 1 **do**

**for** *j* ←*i* + 1 **to** *n* **do**

*d* ←min*(d, sqrt((xi*− *xj )*2 + *(yi*− *yj )*2*))*

**return** *d*

1. Show how to compute the number of instruction executed in this algorithm to find the results?
2. What is the complexity of this algorithm?

***Solution***:

1. The number of computations with respect to size of the problem which is n is as follow:

= 2 \* [(n-1)+ (n-2)+…+1]

= (n-1)\*n

1. The complexity of this algorithm is O(n2)
2. Set up and solve a recurrence relation for the number of times the following algorithm’s basic operation is executed. [6 marks]

Algorithm Q(n)

//Input: A positive integer n

if n = 1 return 1

else return Q(n − 1) + 2 ∗ n − 1

***Solution***:

The recurrence relation

M(n) = M(n − 1) + 2,

M(1) = 0.

The solution of the recurrence

M(n) = M(n − 1) + 2

= [M(n − 2) + 2] + 2 = M(n − 2) + 2 + 2

= [M(n − 3) + 2]+2+2 = M(n − 3)+2+2 + 2

= ...

= M(n − i) + 2i

= ...

= M(1) + 2(n − 1)

= 2(n − 1).

1. State the Master Theorem then use it to find the O notation for the following recurrence relations. [6 marks]

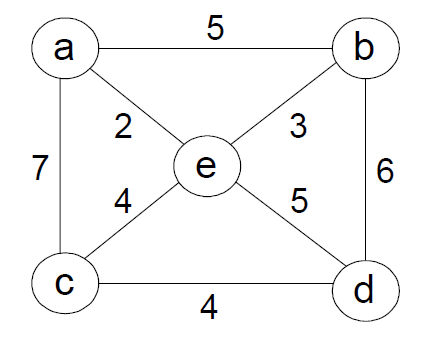
T(n) = 4T(n/2) + 2n

***Solution***:

f(n) = a f(n/b) + cnd

*a =* 4, *b* = 2 ⇒ Master theorem cannot be applied on this recurrence

1. Use Kruskal algorithm to find the minimum spanning tree of the following graph, clarify the steps of the algorithm in your solution [10 marks]



***Solution***

**Procedure** ***Kruskal (V***, ***E***)

**Begin**

***T***= {};

***n*** = |***V***|

**1. *Sort*** the edges of ***E*** in an ascending order according to weight ***w****;*

**2. While |T| < *n***

Remove (***u***,***v***)edge of lowest weight ***w*** from ***E***

**if**( ((***x***,***u***)edge ∈ ***T*** ) *&&* ((***y***,***v***)edge ∈ ***T***) )

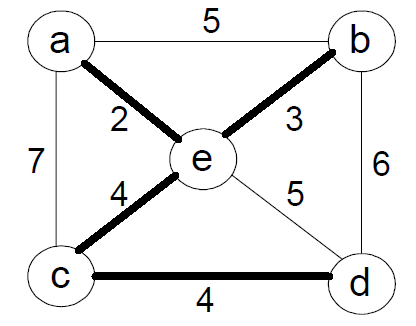
disregard (***u***,***v***)edge

**else**

add (***u***,***v***)edge to ***T***

return ***T***;

**End *Kruskal*;**



1. You have a set A of n nuts and a set B of n bolts, such that each nut in A has a unique matching bolt in B. Unfortunately, the nuts in A all look the same, and the bolts in B all look the same as well. The only kind of a comparison that you can make is to take a nut-bolt pair (a,b) such that aA, and bB, and test to see if the threads of a are larger, smaller or a perfect match with the treads of b. Describe an algorithm ,based on divide and conquer algorithm, to match up all of the nuts in set A with all of the bolts in set B. *(Hint: Express the essence of the algorithm using simple English words or pseudo code)*

***Solution***:

Match (A , B) {

1) If A and B contain one element, just match them and finish. Else continue.

2) Pick a nut at random, let this nut be a.

3) Compare each bolt to a.

i) Place each smaller bolt in a set BL.

ii) Place each larger bolt in a set BH.

iii) If it matches, set aside this particular bolt and label it b.

4) Compare every other nut (except a) to b.

i) Place each smaller nut in a set AL.

ii) Place each larger nut in a set AH.

5) recursively call Match(AL, BL)

6) recursively call Match(AH, BH)

}

This work is very similar to QuickSort

1. Does greedy produces optimal solution to the fractional knapsack problem? and what is the greedy choice? [5 Marks]

***Solution***:  
Yes , it produces optimal solution. The greedy choice is arranging items in descending order based on their value/weight

1. Given two strings ‘X’ and ‘Y’ with length “n” and “m” respectively. You are required to design a dynamic programming algorithm to find the length of the longest common substring between the two given strings. For example, if the given strings are “GeeksforGeeks” and “GeeksQuiz”, the output should be “5” as longest common substring is “Geeks”. Write down the recurrence equation (with base condition) , and define its major terms (don’t write the whole algorithm).  [15 Marks]

***Solution***:

The idea is to find length of the longest common suffix for all substrings of both strings and store these lengths in a table.

The longest common suffix has following optimal substructure property

LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1

( if X[m-1] = Y[n-1])

= 0 Otherwise (if X[m-1] != Y[n-1])

The maximum length Longest Common Suffix is the longest common substring.

LCSubStr(X, Y, m, n) = Max(LCSuff(X, Y, i, j)) where 1 <= i <= m

and 1 <= j <= n

1. Write pseudo code of floyd's algorithm to find all pairs shortest paths in a graph ( The paths themselves and their lengths)

***Solution***:

Algorithm ModifiedFloyd (W[1..n,1..n])

// input : the weight matrix W of the weighted graph

// output : the distance matrix of the shortest paths' lengths D & the Path matrix

D←W // it is not necessary if W can be overwritten

for each vertex pair (i,j)

path[i,j] ← null

for k ← 1 to n do

for i ←1 to n do

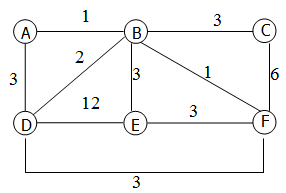
for j ← 1 to n do

if (d[i,k]+d[k,j] < d[i,j])

path[i,j] ←vertex k

d[i,j] ← d[i,k]+d[k,j]

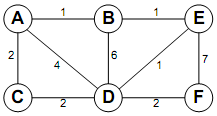
1. Consider the following graph:



* 1. What is the cost of the minimum spanning tree?
  2. How many minimum spanning trees does it have? Give a short argument to justify your answer.

***Solution***:

1. 10
2. **Two MST, The only choice arises when you have the edges BE & EF to choose between.**
3. Suppose Dijkstra’s algorithm is run on the following graph, starting at vertex "*A"*.



Fill out the following table showing the intermediate distance values of all the vertices after each iteration of the algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| iteration → | Initial | 1 | 2 | 3 |
| Tree  Q |  |  |  |  |
| A |  |  |  |  |
| B |  |  |  |  |
| C |  |  |  |  |
| D |  |  |  |  |
| E |  |  |  |  |
| F |  |  |  |  |

Where: *Q : Vertex priority queue, and Tree : Tree of shortest paths*

***Solution***:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **iteration →** | **Initial** | **1** | **2** | **3** |
| **Tree**  **Q** | 0 | A | B | C |
| **A** | 0 | 0 | 0 | 0 |
| **B** | ∞ | 1 | 1 | 1 |
| **C** | ∞ | 2 | 2 | 2 |
| **D** | ∞ | 4 | 4 | 4 |
| **E** | ∞ | ∞ | 2 | 2 |
| **F** | ∞ | ∞ | ∞ | ∞ |

1. Consider the following algorithm

Algorithm Enigma(A[0..n − 1, 0..n − 1]

1: {Input: A n × n matrix A[0..n − 1, 0..n − 1] of real numbers}

2: for i ← 0 to n − 2 do

3: for j ← i + 1 to n − 1 do

4: if A[i, j] = A[j, i] then

5: return false

6: end if

7: end for

8: end for

9: return true

1. What does this algorithm compute? [1 marks]

***Solution***: This program checks if the matrix is symmetric or not.

1. What is its basic operation? [1 marks]

***Solution***: The basic operation is the comparison of the matrix elements.

1. How many times is the basic operation executed and what is the efficiency class of this algorithm? [5 marks]

***Solution***: Depending on if the matrix is symmetric or not, the number of comparisons differ.

In the best case the first comparison shows that the matrix is not symmetric so in the best case we have Θ(1) efficiency class.

In the worst case we have to perform n2 /2 comparisons, so in the worst case we have Θ(n2) efficiency class.

In average we have to calculate the following:

1. Consider the following algorithm

Algorithm Riddle(A[0..n − 1])

1:{Input: An array A[0..n − 1] of real numbers

2:if n = 1

4: return A[0]

5:else

6: temp←Riddle(A[0..n − 2])

7: if temp ≤ A[n − 1]

8: return temp

9:else

10: return A[n − 1]

* 1. What does this algorithm compute?

***Solution***: The algorithm computes the value of the smallest element in a given array.

* 1. Set up a recurrence relation for the algorithm’s basic operation count and solve it.

***Solution***: The recurrence for the number of key comparisons is

C(n)=C(n−1) + 1 for n>1, C(1)=0.

Solving it by backward substitutions since master theorem is not suitable in this case:

*C(n)* = *C(n* − 1*)* + 1 substitute *C(n* − 1*)* = *C(n* − 2*)* + 1

= [*C(n* − 2*)* + 1]+ 1= *C(n* − 2*)* + 2 substitute *C(n* − 2*)* = *C(n* − 3*)* + 1

= [*C(n* − 3*)* + 1]+ 2 = *C(n* − 3*)* + 3*.*

A General formula for the pattern: *C(n)* = *C(n* − *i)* + *i.*

What remains to be done is to take advantage of the initial condition given. Since it is specified for *n* = 0*,* we have to substitute *i* = *n* in the pattern’s formula to get the ultimate result of our backward substitutions:

*C(n)* = *C(n* − 1*)* + 1= *. . .* = *C(n* − *i)* + *i* = *. . .* = *C(n* − *n)* + *n* = *n.*

Then the complexity C(n) of this Riddle function is *O(n)*

1. We have two input arrays, an array **A** with **m** elements and an array **B** with **n** elements, where **m** is less than or equal to **n**. There may be duplicate elements. We want to decide if every element of B is an element of A.

Describe an algorithm to solve this problem in O (**n** log **m**) worst-case time. [3 marks]

***Solution***:

First we sort **A** by ***MERGESORT*** (in **O(m log m)** time).

Then for each element of **B** we do a binary search in the sorted list of **A** (in O(**n** log **m**) time). The total worst-case running time is O(( **m** + **n** ) log **m**) = O(**n** log **m**).

1. Consider the ***BINARY SEARCH (A, v, low, high)*** takes a sorted array **A** in a range [low..high] to search for a value ***v***. The procedure compares ***v*** to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration. Both iterative and recursive versions return either an index ***i*** such that **A**[***i***]=***v***, or NIL if no entry of **A**[low..high] contains the value v.
2. Write the iterative version of this algorithm. [2 marks]
3. Analyze the complexity of the iterative version in the best and worst case scenarios. [2 marks]
4. Write the recursive version of this algorithm. [2 marks]
5. State the master theorem.
6. Analyze the complexity of the recursive version of the worst case scenario using the master theorem. [2 marks]

***Solution***:

1. The iterative version of the algorithm is as follows:

Algorithm iterativeBinarySearch

1: Input{ A, v, low, high}

2: while (low≤high)

3: mid = ⌊low+high⌋/2

4: if (v==A[mid])

5: *return* mid

6: else if (v> A[mid])

7: low=mid+1

8: else

9: high=mid-1

10: *return* NIL

1. The algorithm in the best case scenario is that v exists in the middle term in the array, in this case the complexity is O(1).

In the worst case scenario where v is the first term or the last term or not exists. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. Then the search is applied first on n/20, then on n/21, then on n/22.. . The division continues with the formula n/2x until n≥2x where the array contains 1 or no elements. This leads to the result that : log **n**≥log2 **2x = x.** This means that in the worst case, the number of steps x is has an upper bound log2 n. The complexity is O(log2 n)

1. The recursive version of the algorithm is as follows:

Algorithm recursiveBinarySearch

1: Input{ A, v, low, high}

2: if (low>high)

3: *return* NIL

4: mid = ⌊low+high⌋/2

5: if(v==A[mid])

6: *return* mid

7: else if (v>A[mid])

8: *return* recursiveBinarySearch(A, v, mid+1, high)

9: else

10: *return* recursiveBinarySearch(A, v, low, mid-1)

1. The master theorm reclusive case equation is as follows:

f(n) = a f(n/b) + cnd

Then the complexity of the recursive function is detected according to the values of a, b and c such that:

* + f(n) = O(nd) if a < bd
  + f(n) = O(nd log2 n) if a = bd
  + f(n) = O(n log a) if a > bd

1. In the worst case scenario where v is the first term or the last term or not exists.

The recurrence for these procedures is therefore T(n)=T(n/2)+c\*n0

Then a=1, b=2 and d=0

Therefore (a=1) = (bd=20=1)

Therefore T(n)=O(ndlog2n)=O(n0log2n)= O(log2n)

1. Write the difference between the Greedy method and Dynamic programming. [2 marks]

***Solution*:**

Greedy method

1. Only one sequence of decision is generated.
2. It does not guarantee to give an optimal solution always

Dynamic programming

1. Many number of decisions are generated.
2. It definitely gives an optimal solution always.
3. What are the features and the drawbacks of dynamic programming?

***Solution*:**

***Features***: Optimal solutions to sub problems are retained so as to avoid recomputing their values. Decision sequences containing subsequences that are sub optimal are not considered. It definitely gives the optimal solution always.

***Drawbacks***: Time and space requirements are high, since storage is needed for all level. Optimality should be checked at all levels.

1. Use the master theorem to solve the following recurrence
   1. T(n) = 2T(n/2) +n\*log(n)
   2. T(n) = 64T(n/8)−n2\*log(n)

***Solution*:**

1. T(n) = 2T(n/2) +n\*log(n) 🡺 O(n\*log2(n))
2. T(n) = 64T(n/8)−n2\*log(n) 🡺 Master Theorem does not apply
3. Consider the following instance of the knapsack problem: Capacity W=6

|  |  |  |
| --- | --- | --- |
| Item | weight | Value |
| 1 | 3 | $25 |
| 2 | 2 | $20 |
| 3 | 1 | $15 |
| 4 | 4 | $40 |
| 5 | 5 | $50 |

* 1. Apply the bottom-up dynamic programming algorithm on this instance to find the optimal subset of items.
  2. How many different optimal subsets does the instance of part (a) have?
  3. In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem’s instance?

***Solution*:**

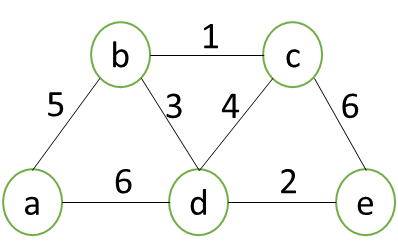
1. Apply the bottom-up dynamic programming algorithm on this instance to find the optimal subset of items.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| w1 = 3, v1 = 25 | 1 | 0 | 0 | 20 | 25 | 25 | 25 | 25 |
| w2 = 2, v2 = 20 | 2 | 0 | 0 | 20 | 25 | 25 | 45 | 45 |
| w3 = 1, v3 = 15 | 3 | 0 | 15 | 20 | 35 | 40 | 45 | 60 |
| w4 = 4, v4 = 40 | 4 | 0 | 15 | 20 | 35 | 40 | 55 | 60 |
| w5 = 5, v5 = 50 | 5 | 0 | 15 | 20 | 35 | 40 | 55 | 65 |

The maximal value of a feasible subset is V [5, 6] = 65.

The optimal subset is {item 3, item 5}.

1. The instance has a unique optimal subset.
2. An instance of the knapsack problem has a unique optimal solution if and only if the algorithm for obtaining an optimal subset, which retraces backward the computation of V [n, W], encounters no equality between V [i − 1, j] and v[i] + V [i − 1, j – w[i]] during its operation.
3. Consider the following graph :



* 1. Apply Kruskal’s algorithm to find a minimum spanning tree of this graph.
  2. Indicate whether the following statements are true or false:
     1. If e is a minimum-weight edge in a connected weighted graph, it must be among edges of at least one minimum spanning tree of the graph.
     2. If e is a minimum-weight edge in a connected weighted graph, it must be among edges of each minimum spanning tree of the graph.
     3. If edge weights of a connected weighted graph are all distinct, the graph must have exactly one minimum spanning tree.
     4. If edge weights of a connected weighted graph are not all distinct, the graph must have more than one minimum spanning tree.

***Solution*:**

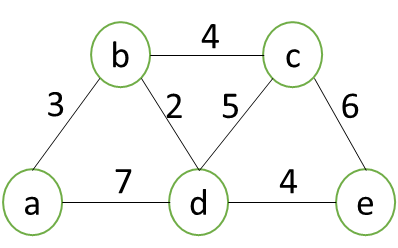
1. Apply Kruskal’s algorithm to find a minimum spanning tree of this graph.

|  |  |  |
| --- | --- | --- |
| Tree edge | Sort list of edges  Selected edges in bold | Illustration |
|  | **bc:1**, de:2, bd:3, cd:4, ab:5, ad:6, ce:6 |  |
| bc:1 | bc: 1, **de:2**, bd:3, cd:4, ab:5, ad:6, ce:6 |  |
| de:2 | bc: 1, de:2, **bd:3**, cd:4, ab:5, ad:6, ce:6 |  |
| bd:3 | bc: 1, de:2, bd:3, cd:4, **ab:5**, ad:6, ce:6 |  |
| ab:5 |  |  |

1. Indicate whether the following statements are true or false:
2. True: Otherwise, Kruskal’s algorithm would be invalid.
3. False: As a simple counter-example, consider a complete graph with
4. three vertices and the same weight on its three edges
5. True: one can always find a spanning tree with the smallest total weight among the finite number of the candidates.
6. False: The graph may contain only one spanning tree even if not all edges are distinct. The given graph in this question is a proof for this statement.
7. Does Kruskal’s or Prim's algorithms or both work correctly on graphs that have negative edge weights?

***Solution***: Both algorithms work correctly for graphs with negative edge weights. One way of showing this is to add to all the weights of a graph with negative weights some large positive number. This makes all the new weights positive, and one can “translate” the algorithms’ actions on the new graph to the corresponding actions on the old one. Alternatively, you can check that the proofs justifying the algorithms’ correctness do not depend on the edge weights being nonnegative.

1. Consider the following graph :



* 1. Use Dijkstra’s algorithm to solve the following instances of the single-source shortest-paths problem with vertex ***a*** as the source.
  2. Let T be a tree constructed by Dijkstra’s algorithm in the process of solving the single-source shortest-paths problem for a weighted connected graph G.
     1. Justify whether T is a spanning tree of G or not?
     2. Justify whether T is a minimum spanning tree of G or not?

***Solution***:

1. Apply Dijkstra’s algorithm with source vertex a:

|  |  |
| --- | --- |
| Tree edge | Remaining vertices |
| a(-,0) | b(-,∞) c(-,∞) d(a,7) e(-,∞) |
| d(a,7) | b(d,7+2) c(d,7+5) e(-,∞) |
| b(d,9) | c(d,12) e(-,∞) |
| c(d,12) | e(c,12+6) |
| e(c,18) |  |

The shortest paths (identified by following nonnumeric labels backwards from a destination vertex to the source) and their lengths are:

* from a to d: a − d of length 7
* from a to b: a − d − b of length 9
* from a to c: a − d − c of length 12
* from a to e: a − d − c − e of length 18

1. Let T be a tree constructed by Dijkstra’s algorithm in the process of solving the single-source shortest-paths problem for a weighted connected graph G.
   * 1. Justify whether T is a spanning tree of G or not?

***Solution***

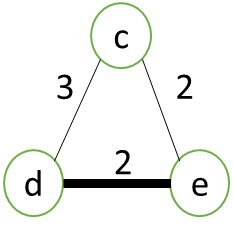
**True**: On each iteration, we add to a previously constructed tree an

edge connecting a vertex in the tree to a vertex that is not in the tree. So, the resulting structure must be a tree. And, after the last operation, it includes all the vertices of the graph. Hence, it’s a spanning tree.

* + 1. Justify whether T is a minimum spanning tree of G or not?

***Solution***

**False**. Here is a simple counterexample:



With vertex a as the source, Dijkstra’s algorithm yields, as the shortest path tree, the tree composed of edges (a, b) and (a, c). The graph’s minimum spanning tree is composed of (a, b) and (b, c).

1. Quick sort is a divide-and-conquer algorithm, answer the following questions:
   1. Give an example showing that quicksort is not a stable sorting algorithm.
   2. Are arrays made up of all equal elements the worst-case input, the best case input, or neither?
   3. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

***Solution***:

1. Quicksort is not stable. As a counterexample, consider its performance on a two-element array of equal values.
2. Arrays composed of all equal elements constitute the best case because all the splits will happen in the middle of corresponding subarrays.
3. Strictly decreasing arrays constitute the worst case because all the splits will yield one empty subarray. (Note that we need to show this to be the case on two consecutive iterations of the algorithm because the first iteration does not yield a decreasing array of size n − 1.)
4. Answer the following: [10 marks]
   1. State whether the following statements are true or false, justify your answers [6 marks]
      1. Every computational problem on input size n can be solved by an algorithm with running time polynomial in n.
      2. Every weighted graph has a unique Minimum Spanning Tree (MST).
      3. Given n integers a1, . . . , an, the third smallest number among a1, . . . , an can be computed in O(n) time.
      4. n(n + 1)/2 is O(n3).
      5. n(n + 1)/2 is O(n2)
      6. n(n + 1)/2 is Ω(n log n)
   2. In merge sort algorithm, [4 marks]
      1. How many such assignments does the recursive merge sort make when sorting an array of length *n*, assuming that *n*=2*k*?, make your answer in terms of k.  [2 marks]
      2. What is the complexity of the merge sort algorithm if it divides the array of n elements into three partitions and then applies the merge on these partitions? Assuming that the merge algorithm in this case is of O(n). [2 marks]

***Solution***:

* 1. State whether the following statements are true or false, justify your answers
     1. False, some problems are exponential.
     2. False, the case when two different edges are of equal weight.
     3. True, use extra memory and the time of searching will still be O(n)
     4. True, as this f(n) = n(n + 1)/2 is smaller than const\*n3, f(n) <= cn3
     5. True, as this f(n) = n(n + 1)/2 is smaller than const\*n2, f(n) <= cn2
     6. True, as this f(n) = n(n + 1)/2 is greater than const\*n\*logn, f(n) >= c\*n\*logn
  2. In merge sort algorithm, [4 marks]
     1. Σk=0..n 2k
     2. O(nlog3 n)

1. Answer the following: [10 marks]
   1. Solve the following recurrence using Master theorem. [3 marks]

T(n) = 1 if n = 1

T(n) = 5\*T(n/4) + n2 otherwise

* 1. Solve the following recurrence using induction method. [3 marks]

T(n) = 1             if n=1

T(n) = 2T(n/2) + n lg             otherwise

* 1. What is the complexity for the following piece of code where n < j ? [4 marks]

j = 20

While (j >= n)

for(i = 1 to j)

x = x+1

j = j / 3;

***Solution***:

1. Using Master theorem, Case 3 since A=5, b=4, d=2

Where, f(n)=n2>nlogba = nlog45 = n1.1 🡺 Case 3

Then, T(n) = O(n2)

1. using induction method, the guess, Hypothesis and induction are as follows:

Guess : T(n) = O(n2)

Hypothesis: T(n/2) <= c(n/2)2

Induction: T(n) <= c(n/2)2 +nlgn

<=(1/2) \* cn2+nlgn

<= cn2 - (cn2-nlgn)

Then T(n) = O(n2)

1. Since the loop is decreased to the 1/3 each time, then complexity of this code will be f(n) = O( *log3 (j-n)* )
2. Using dynamic programming, find the Least Common subsequence between the following two strings.

S1: ACVAGCAV

S2: VAGVACCA

***Solution***:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | A | C | V | A | G | C | A | V |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| V | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| G | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 3 |
| V | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 4 |
| A | 0 | 1 | 1 | 2 | 3 | 3 | 3 | 4 | 4 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 |
| C | 0 | 1 | 3 | 2 | 3 | 3 | 4 | 4 | 4 |
| A | 0 | 1 | 3 | 2 | 3 | 3 | 4 | 5 | 5 |

So the longest common subsequence is as follows: AVACA

\_ A C V A G C A V

| | | | |

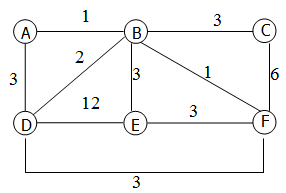
V A G V A C C A \_

1. Does greedy method produces optimal solution to the fractional knapsack problem? and what is the greedy choice? **[9 marks]**

***Solution***:

Yes , it produces optimal solution. The greedy choice is arranging items in descending order based on their value/weight

1. Consider the following graph:



* 1. What is the cost of the minimum spanning tree? [5 marks]
  2. How many minimum spanning trees does it have? Give a short argument to justify your answer. [5 marks]

***Solution***:

1. 10
2. Two MST, the only choice arises when you have the edges BE & EF to choose between.
3. For the partitioning procedure outlined in this section:
   1. Prove that if the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p.
   2. Prove that when the scanning indices stop, j cannot point to an element more than one position to the left of the one pointed to by i.
   3. Give an example showing that quicksort is not a stable sorting algorithm.

***Solution***

1. Let i = j be the coinciding values of the scanning indices. According to the rules for stopping the i (left-to-right) and j (right-to-left) scans, A[i] ≥ p and A[j] ≤ p where p is the pivot’s value. Hence, A[i] = A[j] = p.
2. Let i be the value of the left-to-right scanning index after it stopped. Since A[i − 1] ≤ p, the right-to-left scanning index will have to stop no later than reaching i − 1.
3. Quicksort is not stable. As a counterexample, consider its performance on a two-element array of equal values.

Stopping the scans after encountering an element equal to the pivot tends to yield better (i.e., more equal) splits. For example, if we did otherwise for an array of n equal elements, we would have gotten a split into subarrays of sizes n − 1 and 0.

1. For the version of quicksort given in this section:
   1. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?
   2. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

***Solution*:**

* 1. Arrays composed of all equal elements constitute the best case because all the splits will happen in the middle of corresponding subarrays.
  2. Strictly decreasing arrays constitute the worst case because all the splits will yield one empty subarray. (Note that we need to show this to be the case on two consecutive iterations of the algorithm because the first iteration does not yield a decreasing array of size n − 1.)

1. For quicksort with the median-of-three pivot selection,
   1. Are strictly increasing arrays the worst-case input, the best-case input, or neither?
   2. Answer the same question for strictly decreasing arrays.

***Solution*:**

The best case for both questions. For either an increasing or decreasing subarray, the median of the first, last, and middle values will be the median of the entire subarray. Using it as a pivot will split the subarray in the middle. This will cause the total number of key comparisons be the smallest.

1. Sort the following functions according their growth rate in an ascending order.

n, n3, 2n, n log n, n2, log n

***Solution*:**

log n, n, n log n, n2, n3, 2n

1. Set up a recurrence relation for the following algorithm?

*ALGORITHM* *GraphComplete(A[0..n − 1, 0..n − 1])*

*//Input: Adjacency matrix A[0..n − 1, 0..n − 1]) of an undirected graph G*

*//Output: 1 (true) if G is complete and 0 (false) otherwise*

***if*** *n = 1* ***return*** *1*

***else***

*if* ***not*** *GraphComplete(A[0..n − 2, 0..n − 2])* ***return*** *0*

***else for*** *j ←0* ***to*** *n − 2* ***do***

***if*** *A[n − 1, j]= 0* ***return*** *0*

***return*** *1*

***Solution*:**

**The recurrence relation is:** T(n)=T(n-1)+2n-1

1. Analyze the complexity of the following algorithm:

string s= ̎̎̎ ̎;

**while (***n >* 0){

**if(**n%2=0**)**

s= ̎ 0 ̎ +s;

**else**

s= ̎ 1 ̎+s;

n= ⌊n/2⌋;

}

**return** s;

***Solution*:**

The number of steps x of this algorithm is based on the value of the input n.

Is x a function of n, that is to say x=T(n)?? ***Yes***!!!!

[n/21], [n/22] , [n/23], … , [n/2x]

The division of n by 2x continues

Until 🡺 2x-1 ≤ n ≤ 2x 🡺(2x)\*0.5 ≤ n ≤ 2x

This means🡺 log2 2x+log 0.5 ≤ log2 n

This means🡺 x + -1 ≤ log2 n

This means 🡺 x ≤ 1+log2 n

This means🡺 x = T(n) ≤ 1+log2 n 🡺T(n)=O(log2n)

1. Analyze the complexity of the following algorithm:

int ***Fib***(**int n**){

if(n ≤ 1)

return n;

else{

return ***Fib***(n-1)+***Fib***(n-2);

}

}

***Solution*:**

* + To analyze this recursive function to find its big O
    - T(0)=T(1)=1 ≤ b
    - T(n) ≤ T(n-1)+T(n-2)+c , to reduce this expression, since T(n-1)>T(n-2)
    - T(n) ≤ 2T(n-1)+c ≤ 4T(n-2)+2c ≤ 8T(n-3)+3c
    - T(n)≤ 2kT(n-k)+kc
    - T(n) ≤ 2nT(0)+nc ≤ 2nb+nc ***at k=n*** 🡺 **T(n) = O(2n)**
  + Note here that this recursive function has many redundant steps.

1. Use Master theorem to find the Θ notation for the following recurrence relations.
   1. *T*(*n*) = 4*T*(*n*/2) + *n*
   2. *T*(*n*) = 4*T*(*n*/2) + *n*2
   3. *T*(*n*) = 4*T*(*n*/2) + *n*3
   4. *T*(*n*) = 4*T*(*n*/2) + *n*2/lg *n*

***Solution*:**

1. ***Ex.*** *T*(*n*) = 4*T*(*n*/2) + *n*

*a =* 4, *b* = 2, d=1 ⇒*.* a = 4 > bd = 21=2

**Case 3**: *f* (*n*) = *nlog a=nlog4=n2*

* + *T*(*n*) = Q(*n*2).

1. ***Ex.*** *T*(*n*) = 4*T*(*n*/2) + *n*2

*a =* 4, *b* = 2, d=2 ⇒*.* a = 4 = bd = 22=4

**Case 2**: *f* (*n*) = *n*d log*n*

* + *T*(*n*) = Q(*n*2lg *n*).

1. ***Ex.*** *T*(*n*) = 4*T*(*n*/2) + *n*3

*a =* 4, *b* = 2, d=3 ⇒ a=4 < bd=23=8

**Case 1**: *f* (*n*) = nd=n3

* + *T*(*n*) = Q(*n*3).

1. ***Ex.*** *T*(*n*) = 4*T*(*n*/2) + *n*2/lg *n*

*a =* 4, *b* = 2, d != constant

* + Master method does not apply.

1. Analyze the complexity of the following algorithm:

**ALGORITHM** *F(n)* //Computes *n*! recursively

//Input: A nonnegative integer *n*

//Output: The value of *n*!

**if** *n* = 0

**return** 1

**else**

**return** *F(n* − 1*)* ∗ *n*

***Solution*:**

* + Since the function *F(n)* is computed according to the formula

*F(n)* = *F(n* − 1*) . n* for *n >* 0*,*

* + Master theorem is not applicable in this form of the recursive function. Instead **method of backward substitutions** is utilized .

*M(n)* = *M(n* − 1*)* + 1 substitute *M(n* − 1*)* = *M(n* − 2*)* + 1

= [*M(n* − 2*)* + 1]+ 1= *M(n* − 2*)* + 2 substitute *M(n* − 2*)* = *M(n* − 3*)* + 1

= [*M(n* − 3*)* + 1]+ 2 = *M(n* − 3*)* + 3*.*

* + A General formula for the pattern: *M(n)* = *M(n* − *i)* + *i.*
  + What remains to be done is to take advantage of the initial condition given. Since it is specified for *n* = 0*,* we have to substitute *i* = *n* in the pattern’s formula to get the ultimate result of our backward substitutions:
  + *M(n)* = *M(n* − 1*)* + 1= *. . .* = *M(n* − *i)* + *i* = *. . .* = *M(n* − *n)* + *n* = *n.*

1. State the approach of the following paradigms:
   1. Brute force
   2. Greedy Algorithms
   3. Divide-and-Conquer
   4. Dynamic Programming
   5. Genetic algorithms

***Solution***:

* 1. ***Brute force***: is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. ***e.g.*** selection and insertion sort algorithms. [*All the algorithms in lecture* 1 *are of Brute force paradigm*]
  2. **Greedy Algorithms**: The solution is constructed through a sequence of steps. At each step the next optimal step is selected locally, without considering whether this step leads to the final global optimal solution or not. ***e.g.*** Minimal spanning tree and Knapsack problem.
  3. **Divide-and-Conquer**: Divide the problem instance to several smaller sub-instances then solve each one independently. Finally, the solutions of these sub-instances are combined to form final solution. ***e.g.*** Merge and quick sort algorithm.
  4. **Dynamic Programming**: Solve problem of divide-conquer where identical sub-instances are computed repeatedly. Smallest sub-instances are solved first and results are placed in table to be used to construct larger sub-instances solution. ***e.g.*** Longest Common Subsequence.
  5. **Genetic algorithms**: Compute the fitness of each solution and select a the solutions according to their fitness value. Then perform crossover (combine each two solutions) to obtain new solutions. Finally perform mutation (make random changes on the combined solutions) to obtain new solutions. e.g. Knapsack problem.

1. Discuss the worst and best case scenarios of the Quick Sort Algorithm.

***Solution***:

* ***Worst case*** has the most unbalanced partitions possible. The sequence of n elements S(n) in this case is partitioned to {pivot, S(n-1)}. So the recursive call on S(n) is of complexity cn, and the recursive call on S(n-1) is of complexity c(n-1), and the recursive call on S(n-2) is of complexity c(n-2), and so on.

**The algorithm formula is *f(n)=f(n-1) + cn***

***This recurrence can be solved by backward substitution as follows:***

***f(n)=f(n-1)+cn=f(n-2)+c(n-1)+cn=f(n-3)+c(n-2)..+cn***

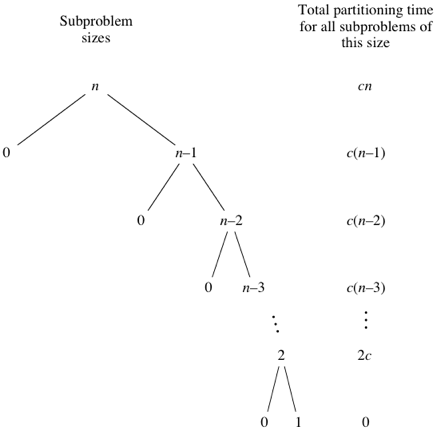
***c\*1+…+c\*(n-3)+c\*(n-2)+c\*(n-1)+c\*(n) =***

***c\*(1+.. +(n-3) +(n-2)+(n-1)+n)=***

***c\*((n+1)(n/2)-1)=***

***c\*n2/2+c\*n/2-1=***

***O(n2)***



* ***Best case*** has partitions are as evenly balanced as possible: their sizes either are equal of size ((n-1)/2) if the number of elements is odd or are within 1 of each other if the number of elements is even. So the recursive call on both partitions is of complexity 2\*c\*(n/2)=cn, then 22\*c(n/22)=cn, then 23\*c(n/23)=cn, and so on.

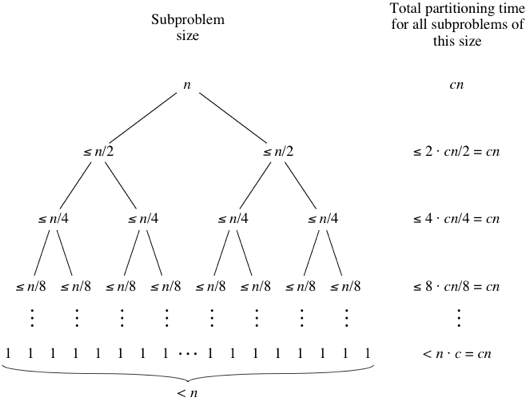
**The algorithm formula is *f(n)=2\*f(n/2)+cn***

This recurrence can be solved by second case of master theorem where a=2, b=2 and d=1.

The F(n) = O() =

O() =

***O()***



1. Does dynamic programming is applicable when sub-problems are independent?

***Solution***:

Dynamic programming is applicable when the sub-problems are dependent, that is, when sub-problems share sub-sub-problems.

1. Discuss the following statement: “Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms”

***Solution***:

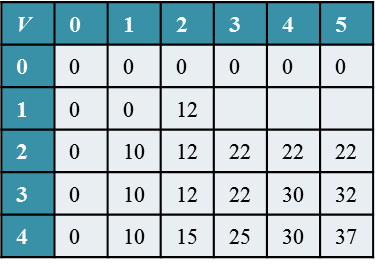
By “inefficient”, we mean that the same recursive call is made over and over.

If same subproblem is solved several times, we can use table to store result of a subproblem the first time it is computed and thus never have to recompute it again.

1. For the following table constructed by knapsack algorithm, state and use the backtracking algorithm to find the optimal solution:

***W***

***j***



***i***

***P*=0**

***P*=2**

***P*=3**

***n***

***P*=3**

***Solution***:

**Algorithm** ***knapsack-backtracking*** (***P*[*n, W***]) //*The function complexity is F(n)=n*

**Input** ***P*[*n, W***] describes the remaining number of items after testing this current item

**Output** Optimal set of items suitable for the knapsack

**set** ***j*** = ***W***;  ***//*** starting from item [i=n,j=W] bottom of the table P

**for *i*** = ***n*** **to** 0  ***// Complexity is n***

**if *P*[*i*, *j*] = *j***

item ***i*** is not selected where the remaining

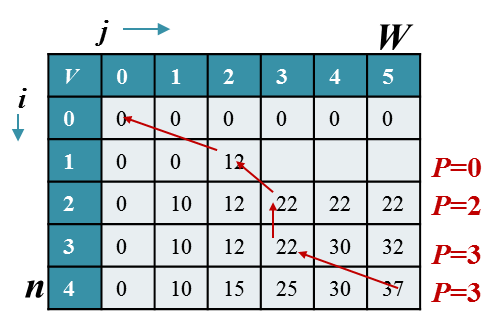
number of items to be selected will not be

affected after testing this item

**else**

item ***i*** is selected and added to optimal set

**set** ***j*** = ***P*[*i*, *j*]**



Initialize j=5

For i = 4

P[i,j]=P[4,5]=3 ≠ j=5

Item 4 is selected

j= P[i,j] = 3

For i = 3

P[i,j]=P[3,3]=3 = j=3

Item 3 is not selected

j= P[i,j] = 3

For i = 2

P[i,j]=P[2,3]=2 ≠ j=3

Item 2 is selected

j= P[i,j] = 2

For i = 1

P[i,j]=P[1,2]=0 ≠ j=2

Item 1 is selected

j= P[i,j] = 0

For i = 0

P[i,j]=P[0,0]=0 = j=0

Nothing is selected an algorithm is terminated

So the selected items are {4,2,1}

1. What are the steps involved in dynamic programming?

***Solution***:

* set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
* solve smaller instances once
* record solutions in a table
* extract solution to the initial instance from that table

1. Which problem does it address and which situations can it be used?

***Solution***   
Optimization problems, the situations we need to reach the optimal solution

1. Solve the following knapsack problem:

Selection of n=4 items, capacity of knapsack M=8

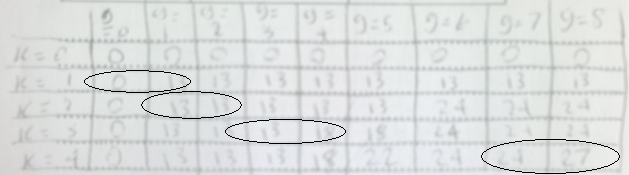
|  |  |  |
| --- | --- | --- |
| Item i | Value vi | Weight wi |
| 1 | 13 | 1 |
| 2 | 11 | 5 |
| 3 | 5 | 3 |
| 4 | 9 | 4 |

max {V[i-1,j], vi + V[i-1,j- wi]} if j- wi ≥ 0

V[i,j] =

V[i-1,j] if j- wi < 0

Initial conditions: V[0,j] = 0 and V[i,0] = 0



g=8, k=4, 27 != 24 🡺 x4=1 and g= 8-w4=8-4=4

g=4, k=3, 18 != 13 🡺 x3=1 and g= 4-w3=4-3=1

g=1, k=2, 13 = 13 🡺 x2=0

g=1, k=1, 13 != 0 🡺 x4=1 and g= 1-w4=1-1=0

Therefore the items selected are 1, 3 and 4

1. In the divide – and – conquer algorithms
   * 1. What is big O in the best case and the worst case of the merge sort and the quick sort?

Merge sort: best case: n lg n

Merge sort: worst case: n lg n

Quick sort: best case: n lg n

Quick sort: worst case: n2

* + 1. What makes the quick sort is better than the merge sort algorithm in some cases.

Quick sort doesn’t work on extra space like in merge sort, so it has less space complexity than merge sort.

1. Suppose that X and Y are two sorted sequences, containing m and n elements respectively. Design and write down the pseudo code of an efficient algorithm to search for any element (z) in the set of m+n combined elements.

***Solution***

First, the merge of the two arrays is of O(n+m)

So we are going to have a single sorted array of a size z = m+n

Second, the use of binary search is of order lg(z)=lg(m+n)

Note:

Any other solution rather than this will be deducted from the total mark.

The merge method is used in the merge sort algorithm in the lecture notes.

The binary search is used and clarified also in the lecture notes.

1. What is Genetics Algorithms? And what are the three main processes of the Algorithm? (Explain each one)

***Solution*:**

Genetics Algorithms:

* Uses concepts from Evolutionary Biology
* Initial generation is tested against the objective function
* Subsequent generation evolve from the 1st generation through Selection, Crossover & Mutation
* Provide efficient, effective techniques for optimization and machine learning applications
* Widely-used today in business, scientific and engineering circles

The three processes are:

Selection: **Selection** means to extract a subset of genes from an existing population, according to any definition of *quality*.

Crossover: Is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next.

Mutation: is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next.

1. Given 5 cities along with the distance of traveling between each pair of them, find the shortest path for visiting all the cities by applying the applicable Genetics Algorithms’ techniques
2. Cairo
3. Alexandria

|  |  |
| --- | --- |
| City | Cost |
| (2,1) | 200 Km |
| (2,3) | 875 Km |
| (2,4) | 370 Km |
| (2,5) | 730 Km |

1. Luxor
2. Suez
3. Sharm El Sahikh

|  |  |
| --- | --- |
| City | Cost |
| (1,2) | 200 Km |
| (1,3) | 655 Km |
| (1,4) | 140 Km |
| (1,5) | 504 Km |

|  |  |
| --- | --- |
| City | Cost |
| (3,1) | 655 Km |
| (3,2) | 875 Km |
| (3,4) | 670 Km |
| (3,5) | 400 Km |

|  |  |
| --- | --- |
| City | Cost |
| (4,1) | 140 Km |
| (4,2) | 370 Km |
| (4,3) | 670 Km |
| (4,5) | 388 Km |

|  |  |
| --- | --- |
| City | Cost |
| (5,1) | 504 Km |
| (5,2) | 730 Km |
| (5,3) | 400 Km |
| (5,4) | 388 Km |

CityList1 (3 5 2 1 4)

CityList2 (2 5 1 3 4)

***Solution*:**

First we apply Order 1 Crossover:

CityList1 (3 5 2 1 4)

CityList2 (2 5 1 3 4)

Child (3 5 2 1 4)

**The mutation phase is optional**

Then we start to sum the total path between those cities in the Child’s list

= 400 + 730 + 200 + 140

= 1440 Km

Then we apply Order 1 Crossover on Child and Citylist2 (Randomly):

CityList2 (2 5 1 3 4)

Child (3 5 2 1 4)

Child2 (2 5 1 3 4)

**The mutation phase is optional**

Then we start to sum the total path between those cities in the Child’s list

=730 + 504 + 655 + 670

=2559

Then we apply Order 1 Crossover on Child and Child2 (Randomly):

Child (3 5 2 1 4)

Child2 (2 5 1 3 4)

Child2 (3 5 2 1 4)

**The mutation phase is optional**

Then we start to sum the total path between those cities in the Child’s list

= 400 + 730 + 200 + 140

= 1440 Km

So, this means that the shortest path is 1440Km.

The user can keep going on as long as he needs to randomly but he has to provide a stopping condition (e.g. the stopping condition here is if the program found the same total number of kilometers again)