**A. Basics (10 marks)**

**A.1 Worst Case Time Complexity Analysis (10 marks)**

State the worst case time complexity for each operation/algorithm. Every correct answer is worth 1 mark.

The operations/algorithms referred below are the unmodified version, as per discussion in class, e.g. as currently discussed in VisuAlgo or as currently implemented in C++ STL. AM/AL/EL are the abbreviations for Adjacency Matrix/Adjacency List/Edge List, respectively.

For graph-related operations, use V to denote the number of vertices and E to denote the number of edges. Otherwise, use N to denote the size of data as usual.

|  |  |  |
| --- | --- | --- |
| 0. | Finding the shortest path a -> b in a weighted graph. | O( (V + E) log V ) |
| 1. | Detecting if the maximum item in the priority\_queue is unique. Note: top() returns the maximum item in O(1) time. | O( log N ) |
| 2. | Finding the *minimum* item in a *max* heap (array-based, not STL priority\_queue). | O( N ) |
| 3. | Finding the median item in a set. | O( N ) |
| 4. | Finding the key that contains the minimum value in a map. | O( N ) |
| 5. | Converting AM into EL. | O( V2 ) |
| 6. | Running depth-first traversal in a simple graph, stored in an *unsorted* EL data structure (instead of the default AL). | O( V+E ) |
| 7. | Finding the shortest distance from a -> b in a simple graph represented by a 2D grid, where you can only traverse up, down, left or right. | O( V ) |
| 8. | Detecting if a given node is part of a cycle in a directed weighted graph. | O(V+E) |
| 9. | Detecting if a given node has a shortest path of “negative infinity” in a directed weighted graph. | O(VE) |
| 10. | Finding the “center” vertex in a graph, where this vertex has the minimum total distance to every other vertices in the connected, undirected, weighted graph. | O( V3 ) |

**B. Analysis (15 marks)**

Prove (show that the statement is correct) or disprove (give a counterexample) the statements below.

1. In an AVL tree with at least 3 elements, |A - B| < 2, where A is the number of leaf nodes and B is the number of non-leaf nodes.

|  |
| --- |
| False. In this example, number of leaves = 4, number of non-leafs = 6. A - B = 2. |

2. The smallest value in the maxheap (size of at least 3) is always found in one of the leaves.

|  |
| --- |
| True. Suppose the smallest value, v\* is not found in one of the leaves.  Then it means that there must be at least 1 value, v, below this smallest value v\*.  This 1 value, v, must definitely be smaller than the smallest value v\* (in order to fulfill heap property).  This violates that our smallest value v\* is THE smallest value.  Thus, there is a contradiction. We conclude that smallest value must be found in one of the leaves. |

3. In a directed acyclic graph, the SSSP problem has a solution that runs as fast as O(V+E).

|  |
| --- |
| True. First run topological sort using DFS in O(V+E).  Next run 1-pass Bellman Ford by relaxing the edges in the topo sort ordering in O(E).  Overall time complexity O(V+2E) = O(V+E). |

4. In a directed weighted graph, we can run Dijkstra algorithm to find the longest path starting from a given source by changing the relax operation from min to max.

|  |
| --- |
| False.  If modified dijkstra: Will end up in an infinite loop if there is a positive weight cycle (which happens very often).  If original dijkstra: Will get wrong answer if there is a positive weight (which happens even more often). |

5. If speed is not a concern, map can support all operations that unordered\_map provides. In other words, we can generally replace all unordered\_map in our code with map and it will still compile and give the correct outputs.

|  |
| --- |
| False. If you dig C++ library, functions like bucket\_count in unordered\_map cannot be replaced using a map.  OR  True. map is ordered, and can support a superset of operations that unordered\_map can provide. |

**C. Alternative Implementations (15m)**

**C.1 BBST Tree std::set Implementation for Heap ADT**

In class, we have discussed the implementation of (Max) Heap ADT, using an array/std::vector as the underlying data structure. For this question, assume that we use std::set as the underlying data structure instead.

Someone suggested that we can potentially speed-up the search(i) operation from O(N) to O(log N), where N is the size of the heap.

Now, what are the implications of such implementation? One such implication has been listed below. Your job is to list down at least 5 (can be more...) other logical statements of this implementation. Your answer will be graded based on the soundness and the quality of the statements (3 marks for each valid statement; -1 mark for random/irrelevant statement; min 0 mark and max 15 marks).

0.

|  |
| --- |
| Using vector: Search operation is O(N) since we must sieve through the whole vector to locate the element in the worst case.  Using set: Search operation is O(log N) since the tree is balanced and the max height to locate the element is log N.  Time complexity improves. |

1.

|  |
| --- |
| Originally, insert operation is O(log N), to insert at the bottom and bubble upwards.  Now, insert operation is O(log N) to insert into a set.  Same time complexity. |

2.

|  |
| --- |
| Originally, delete operation is O(N), to find it, replace with the last element, then bubble down the last element.  Now, delete operation is O(log N) by erasing in a set.  Time complexity improves. |

3.

|  |
| --- |
| Originally, finding the next larger element is O(N) since we have to loop through the array to find it.  Now, finding the next larger element is O(log N) since we can find successor in O(log N) time by calling it++.  Time complexity improves. |

4.

|  |
| --- |
| Originally, finding the smallest element is O(N) since we have to loop through the array to find it.  Now, finding the smallest element is O(log N) since we can find any element in O(log N) time.  Time complexity improves. |

5.

|  |
| --- |
| Originally, extractMax is O(log N), to replace the last value with the current max and bubble down the last value.  Now, extract max is O(log N), since removing any element in the set is O(log N).  Time complexity stays the same. |

**C.2 New Adjacency List Implementation (9m)**

In class, we have discussed the implementation of adjacency list using a vector<vector<ii>> as the underlying data structure, where ii datatype refers to pair<int, int> and it contains information on {neighbour vertex, edge weight}. Such an adjacency list can support many graph operations (such as DFS/BFS) efficiently. **For this question, assume that we use unordered\_map<?, unordered\_map<?, int>> as the underlying data structure instead. ‘?’ can be any data-type you want it to be, as long as it makes sense.**

Now, what are the implications of such implementation? One such implication has been listed below. Your job is to list down **at least 3** (can be more...) other logical statements of this implementation. Your answer will be graded based on the soundness and the quality of the statements (3 marks for each valid statement; -1 mark for random/irrelevant statement; min 0 mark and max 9 marks).

0.

|  |
| --- |
| Originally, our adjacency list can only enumerate neighbours of vertex that are integers, by calling adjList[vertex].  Now, our adjacency list can support vertex that are strings/char as well, by calling adjList[vertex]. We just need to implement our adjacency list as unordered\_map<string, unordered\_map<string, int>>. |

1.

|  |
| --- |
| The time complexity of enumerating all neighbours is still O(N) for N neighbours since iterating an unordered\_map is O(N). |

2.

|  |
| --- |
| Converting adjacency list to adjacency matrix is still O(V+E) (if the matrix is already created, else O(V2) to create it). Enumerating each vertex is still O(V), enumerating all edges of each vertex is still O(E) due to my reason 1. |

3.

|  |
| --- |
| Originally: Finding for a particular edge between 2 vertex is O(N) for N neighbours from vertex1. for (ii neighbour: adjList[v1]) { if (neighbour.first == v2) return neighbour.second;}  Now: Finding a particular edge between 2 vertex is O(1), simply return adjList[v1][v2] if you confirm that that the keys are in this unordered\_map. |

4.

|  |
| --- |
| Originally, the data structure is compact (all vertex are within 1 index from each other).  Now, the data structure is **less** compact since unordered\_map deals with hashing and the vertex might not be near each other. |

5.

|  |
| --- |
| Do not have know the number of vertices beforehand unlike Adjacency List and Adjacency Matrix, when the only the number of edges and the edge list is provided.  -- alternatively --  If we do not know the number of vertices beforehand, we might need to process the edge list once to find out the number of vertices, before creating the AL or AM.  However, we do not need to do so in when using unordered\_map. |

**D. Applications (60m)**

**D.1 Bipartite Graph (5+10m)**

In class, we briefly mentioned that in a bipartite graph, the vertices can be decomposed into two sets, where there are no connection between any pair of vertices in the same set.

1. For a bipartite graph with V vertices, find the maximum possible number of edges in terms of V.

|  |
| --- |
| If V is even, then [If V = 12, then E = 36]  If V is odd, then [If V = 13, then E = 42] |

1. For a connected simple graph (represented using Adjacency List), check if the graph is bipartite. You may use global variables as needed.

|  |
| --- |
| typedef pair<int, int> ii;  // Fill in the blanks for this function  bool isBipartite(vector<vector<int> > AL) {  // Initialze visited array  // 0->unvisited, 1->group1, 2->group2  int visited[(int)AL.size()] = {0};  // Initialize queue with 1 item  queue<ii> q; // queue of (v, group) pairs  q.push(ii(0, 1));  // Process until queue is empty  while (!q.empty()) {  ii vGroup = q.front(); q.pop();  int v = vGroup.first; int group = vGroup.second;    // Check if bipartite condition is violated  if (visited[v] == 1 && group == 2) return false;  else if (visited[v] == 2 && group == 1) return false;  else if (visited[v] > 0) continue; // visited and correct    // Handles the condition where current vertex is unvisited  vector<int> neighbours = AL[v];  visited[v] = group; // Set current vertex as group 1 or 2  int newGroup = (group == 1) ? 2 : 1; // Toggle group num  for (int neighbour: neighbours) {  q.push(ii(neighbour, newGroup));// Should we update visited array here?  }  }    return true;  } |

The time complexity of this algorithm is O( V+E ).

**D.2 Air Travel (The Cheapo Way), Easier (15m)**

Air travel is (very) expensive. Steven wants to minimize the total cost that he have to pay in his journey. There are two types of costs: flight cost (incurred when travelling from one country to another country), and airport tax (incurred at each country).

This is a hypothetical scenario when Steven explored his air travel options to attend ACM ICPC World Finals 2018, 15-21 April 2018 @ Beijing, China (NUS Week 13, that's why Steven is on official Admin Leave that week).

**Input Description**

The first line of input contains two integers, F (1 ≤ F ≤ 100) and G.

The next F lines contains string CITY\_NAME (3 UPPERCASE ['A'..'Z'] characters) and integer AIRPORT\_TAX (less than 1000).

The next G lines contains string CITY\_NAME\_1, string CITY\_NAME\_2 and integer FLIGHT\_COST (less than 1000). This means that the cost from flying **to and from** CITY\_NAME\_1 and CITY\_NAME\_2 is FLIGHT\_COST.

Each pair of (ORIGIN, DESTINATION) only appears once in the input.  
It is guaranteed that “SIN” (Singapore) and “PEK” (Beijing) is mentioned as two of the cities in the input.

**Output Description**

Output the cheapest cost Steven has to pay to get from SIN (Singapore) to PEK (Beijing).

**Additional Input Constraint(s)**

Since this is an easy variant of the next question, we relax the constraint and **set all AIRPORT\_TAX and FLIGHT\_COST to 100**.

|  |  |
| --- | --- |
| **Sample Input** | **Sample Output** |
| 5 7  SIN 100  PEK 100  KUL 100  JKT 100  MEL 100  SIN KUL 100  SIN JKT 100  SIN PEK 100  KUL PEK 100  JKT MEL 100  JKT PEK 100  MEL PEK 100 | 300 |

Based on the sample input, the cheapest path would be to travel from SIN (Singapore) to PEK (Beijing) directly, with a cost of 100 + 100 + 100 = 300.

Design the best algorithm to solve this problem.

|  |
| --- |
| Run a usual BFS routine on an unweighted graph (and ignoring airport tax at each vertex), from source at SIN. This is the fastest algorithm for SSSP, so we get an array of “number of steps” from SIN to every other country. We get the “number of steps” from SIN to PEK.  If the shortest “number of steps” from SIN to PEK is length s, output 100 \* (2s + 1).  For this implementation, to make life easier for myself, I decided to use an unordered\_map with countryname as keys instead of mapping them to an integer and use the usual vector<vector<int>> AL implementation. This will not TLE (only in this case), since our country names are all 3 characters so searching for the countries by key is fast.  typedef pair<string, int> si;  int cheapestCost(unordered\_map<string, vector<string> > AL) {  // Initialize visited (country, distance) pairs to be empty  unordered\_map<string, int> countryDists;  // Initialize queue with 1 item  queue<string> q; // queue of (v, group) pairs  q.push(si(“SIN”, 0));  // Process until queue is empty  while (!q.empty()) {  si countryDist = q.front(); q.pop();  string country = countryDist.first;  int distance = countryDist.second;    // Check if visited and handle only unvisited vertex  if (countryDists.find(country) != countryDists.end()) {  continue;  }  // Mark current vertex as visited, then handle neighbours  countryDists[country] = distance;  vector<string> neighbours = AL[country];  for (int neighbour: neighbours) {  q.push(si(neighbour, distance+1));  }  }    return 100 \* (2 \* countryDists[“PEK”] + 1);  } |

The time complexity of this algorithm is O( V+E ).

**D.3 Air Travel (The Cheapo Way), Harder (10m)**

Now we remove the previous input constraint. **The AIRPORT\_TAX and FLIGHT\_COST can vary.**

|  |  |
| --- | --- |
| **Sample Input** | **Sample Output** |
| 5 7  SIN 100  PEK 50  KUL 200  JKT 250  MEL 300  SIN KUL 150  SIN JKT 50  SIN PEK 1000  KUL PEK 600  JKT MEL 200  JKT PEK 550  MEL PEK 100 | 1000 |

In this sample input, the cheapest path is from SIN (Singapore) to JKT (Jakarta) to PEK (Beijing). The total airport tax is 100 + 250 + 50 = 400, and total flight cost is 50 + 550 = 600, leading to a overall total cost of 1000.

There are two parts to this question.

1. Draw the sample input as a graph such that we can just use a pre-packaged algorithm (covered in CS2040C) without making any modifications to it.
2. State your best algorithm to solve this problem.

|  |
| --- |
| 1) Just set all vertex weights as incoming edge weights, and set a dummy node that points to SIN. Run SSSP algorithm from the dummy node (the source).    2) Dijkstra (either original or modified is fine). |

The time complexity of this algorithm is O( (V+E) log V ).

**D.4: Identical Subarrays (10+5m)**

*Warning: This is the toughest question in this paper. Do not attempt until you have finished all the other questions!*

You are given an array of **N** integers, arr and would like to support **Q** of the following operations:

1. Replace the value at index **X** with **V**. (i.e. arr[X] = V)
2. Check if the subarray starting at index **L** and ending at index **R** (inclusive) all contains the same value. (i.e. check if arr[L], arr[L+1], …, arr[R-1], arr[R] are all the same)

If they are all the same value, print that value. Otherwise, print “no”.

The input starts with a single integer **N**, followed by **N** integers describing the initial array arr on the next line. **N** will be up to 100,000.

This is followed by another integer **Q**, the number of operations that should be performed. **Q** will be up to 100,000.

**Q** lines will follow. The first integer of each of these lines will be either 1 or 2, denoting the type of operation that should be performed.

For operation (1), two more integers **X** and **V** will follow. It is guaranteed that **0** **≤ X < N**.

For operation (2), two more integers **L** and **R** will follow. It is guaranteed that **0** **≤ L ≤ R < N**.

For the easier version, the following constraints apply. (10m)

* All integers in the initial array **is guaranteed to be either 0 or 1.**
* All values of **V** in operation (1) **is also guaranteed to be either 0 or 1**.

For the harder version, the following constraints apply. (5m, no partial marks)

* All integers in the initial array will be from **0 to 1,000,000,000**.
* All values of **V** in operation (1) is also guaranteed to be from **0 to 1,000,000,000**.

Note: Solving the harder version will automatically award you points for the easier version. However, 0 marks would be awarded if the harder version is incorrect and there is nothing written for the easier version.

Sample Input for Easier Version

|  |  |
| --- | --- |
| **Sample Input** | **Sample Output** |
| 4  0 1 0 0  8  2 2 3  2 0 3  1 1 0  2 0 3  1 1 1  1 2 1  2 1 2  1 0 0 | 0  no  0  1 |

Explanation of Sample Input:

Initially the content of the array is as such: [0, 1, 0, 0].

The first operation asks if arr[2] to arr[3] have the same value. This corresponds to the underlined portion of the array: [0, 1, 0, 0]. Since the value is the same, we output that value, 0.

The second operation asks if arr[0] to arr[3] have the same value. This corresponds to the entire array: [0, 1, 0, 0]. As there are a mixture of ‘0’ and ‘1’ in this range, not all values are the same. Hence, we output ‘no’.

The third operation is to change arr[1] to a value of ‘0’. Now the array is: [0, 0, 0, 0].

The fourth operation asks if arr[0] to arr[3] have the same value. This corresponds to the entire array: [0, 0, 0, 0]. Since the value is the same, we output that value, 0.

The fifth operation changes arr[1] to a value of ‘1’.

The sixth operation changes arr[2] to a value of ‘1’. The array is now: [0, 1, 1, 0].

The seventh operation asks if arr[1] to arr[2] have the same value. This corresponds to the underlined portion of the array: [0, 1, 1, 0]. Since the value is the same, we output that value, 1.

The last operation changes arr[0] to ‘0’. Since arr[0] is already 0, this operation does not change the array at all. The array remains as [0, 1, 1, 0].

Sample Input for Hard Version: (no explanation)

|  |  |
| --- | --- |
| **Sample Input** | **Sample Output** |
| 10  3 0 1 1 1 2 1 1 3 3  5  2 2 7  1 5 1  2 2 7  1 7 3  2 7 9 | no  1  3 |

**Solution Sketch to Easier Variant**

Initialization

* Keep a copy of the array, arr.
* Create two set<int> data structures, called ‘zeros’ and ‘ones’.
* In the ‘zeros’ set, store the indexes where the array has value 0.
* In the ‘ones’ set, store the indexes where the array has value 1.

For every operation (1):

* Update arr[x] = v.
* If the value has been changed from 0 to 1, remove ‘x’ from the ‘zeros’ set and add ‘x’ to the ‘ones’ set.
* If the value has been changed from 1 to 0, remove ‘x’ from the ‘ones’ set and add ‘x’ to the ‘zeros’ set.

For every operation (2):

* Check arr[L]
* If arr[L] == 0, check whether ‘ones.lower\_bound(L) > R’. If so, then the entire subarray is zeros. Otherwise, output ‘no’.
* If arr[L] == 1, check whether ‘zeros.lower\_bound(L) > R’. If so, then the entire subarray is ones. Otherwise, output ‘no’.

**Solution Sketch to Harder Variant**

Initialization

* Keep a copy of the array, arr.
* Create one set<int> data structure, called ‘diff’.
* For every pair of adjacent values in array arr [eg: (arr[x-1], arr[x]) where 0 < **x** < **N**]
  + Insert x into ‘diff’ if arr[x-1] != arr[x]

For every operation (1):

* Update arr[x] = v.
* If arr[x-1] != arr[x], add ‘x’ into ‘diff’.
* Otherwise if arr[x-1] == arr[x], remove ‘x’ from diff.
* If arr[x] != arr[x+1], add ‘x+1’ into ‘diff’.
* Otherwise if arr[x] == arr[x+1], remove ‘x+1’ from diff.
* Implementation details: be careful when **x** is **0** or **x** is **N-1**.

For every operation (2):

* Check whether ‘diff.upper\_bound(L) > R’. If so, then the entire subarray is the same value. Hence, output arr[L].
* Otherwise, output ‘no’.

**[TAKEN OUT OF MOCK FINALS QUESTIONS, treat this as extra practice.]**

**B True/False?**

6. In a directed acyclic graph, there is always a unique path between any 2 vertices.

|  |
| --- |
| Highlight over the text to see the answers.  A->B->C  A->D->C  Thus, false. |

7. To find all-pairs shortest path for a tree, Floyd Warshall algorithm gives the best time complexity compared to other algorithms.

|  |
| --- |
| Highlight over the text to see the answers.  Floyd Warshall algorithm runs in O(V3)  Running BFS/DFS V times starting from each vertex as the source is O(V\*(V+E)). Since E = V-1 (tree property), the time complexity is O(V2)  Thus, Floyd Warshall is slower than V-times BFS/DFS to find APSP of a tree. False. |

8. The pre-order and in-order traversals of a rooted tree uniquely defined the tree. That means, given an array that is the pre-order of a tree, and an array that is the in-order of the same tree, we can construct only 1 possible tree that satisfy this constraint.

|  |
| --- |
| Highlight over the text to see the answers.  <https://www.geeksforgeeks.org/construct-tree-from-given-inorder-and-preorder-traversal/>  True. (Just learn how to do it, I don’t have a nice proof for this. The current ugly one spans pages, from stackoverflow <https://stackoverflow.com/questions/30556590/how-does-inorderpreorder-construct-unique-binary-tree>)  Interestingly:  There are (1) pre-order, (2) in-order, (3) post-order traversals of a tree. (1,2) and (2,3) uniquely defines a tree, but (1,3) does not. Can you come up with a counterexample to show why pre-order+post-order does not uniquely define a tree? |

**Section D**

1. **Read up on Lowest common ancestors (on binary tree, on binary search tree, on AVL tree) to learn a bit more about tree traversals and recursions.**
2. **Read up on Lego Mindstorms (Week 13 tutorial slides). Graph modelling might be tested. 2D Grid graph traversals might be tested too, so this is relevant.**
3. **Try to do more Kattis problems since Steven used 2 Kattis problems for this section last sem. You might encounter an exact copy. You can prepare Kattis questions+solutions into exam hall if you wish, since it’s open book…. (You can also google github solutions to kattis problems if you don’t know how to solve them…)**

**Continuation of D.2 and D.3**

Suppose Steven can begin his journey from any country near Singapore (SIN), because he can take a free bus from his house (in Sheares Hall) to these countries.

Suppose Steven can also end at any country near Beijing (PEK), because he can then take a free bus from these countries to ICPC competition venue.

To make things harder, **the graph is now directed** (so SIN PEK 100 means that it costs $100 to get from Singapore to Beijing, and this does not give any information on how much it costs to get from Beijing to Singapore instead.)

To simplify the problem, there is no more AIRPORT\_TAX. Thus, **all AIRPORT\_TAX is 0**.

|  |  |
| --- | --- |
| **Sample Input** | **Sample Output** |
| 7 11  SIN 0  PEK 0  KUL 0  JKT 0  MEL 0  JPN 0  KOR 0  SIN KUL 150  SIN JKT 50  SIN PEK 1000  KUL PEK 600  KUL JPN 500  JKT MEL 200  JKT PEK 550  MEL PEK 100  MEL KOR 50  KOR PEK 200 | 350 600 300 // Part (a) {SIN, KUL, JKT} respectively  250 // Part (b) JKT->MEL->KOR = 250 |

1. Describe your algorithm to find all cheapest costs that ends in PEK starting from these source vertices {SIN, KUL, JKT}. Analyse the time complexity.
2. Describe your algorithm to find the cheapest costs from any of {SIN, KUL, JKT} to any of {PEK, JPN, KOR}. Analyse the time complexity.

|  |
| --- |
| Highlight over the text to see the answers.   1. Reverse ALL edges first (i.e. if SIN PEK 1000, we change it to PEK SIN 1000). Then run Dijkstra (SSSP) starting from PEK. We then output dist[‘SIN’], dist[‘KUL’], dist[‘JKT’]. Time complexity of reversing all edges is O(E). Dijkstra is O((V+E)logV) Overall O((V+E) log V). 2. From the current graph, add a dummy source “STE” (for Steven’s House) that connects to SIN, KUL, JKT with cost 0. Also, add a dummy destination “ACM” (for ACM ICPC World Finals) and let PEK, JPN, KOR point to ACM with cost 0. Run the usual Dijkstra (SSSP) from STE to ACM and output dist[“ACM”]. Time complexity is O((V+E) log V). |

**D.5: Practical Exam Nightmare (10m, no partial credit)**

For one of the algorithm module in NUS (who shall not be named), there was an issue with the practical exam where many students were unable to login and/or submit to an online judge platform (again, it shall not be named). Thus, many student’s code are left unmarked/unsubmitted at the end of the practical exam (the horror!).

In light of such chaos, the Prof. S, the module coordinator, have to quickly find the shortest path to cycle through all the labs, so that he can instruct the students on which questions to do with minimum delay.

The input starts with three integers, 2 ≤ N ≤ 6 for the number of labs, and 1 ≤ R,C ≤ 20 for the total number of rows and columns of the map respectively.

The next R rows contains C characters each, where ‘#’ denotes a wall, ‘.’ denotes a floor tile, and ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’ denotes the lab numbers accordingly.

Prof. S will always start from lab 2 (since it is the biggest lab), and cycle through all N labs before ending at lab 2 again. It is allowed for the Prof. S to cycle through the same lab more than once during his walk (i.e. to move between two labs, sometimes we have to pass through an intermediate lab to reach it, and this is okay). You may assume that lab 2 in the input will always be reachable to every other labs, and thus there is always a valid path.

|  |  |
| --- | --- |
| **Sample Input** | **Sample Output** |
| 4  5 12  ############  #1...#2....#  #...3..##..#  #..###....4#  ############ | 24 |

By eyeballing the sample input, we see that the shortest cycle would be 2->4 (distance 6), 4->3 (distance 7). 3->1 (distance 4), 1->2 passing through 3 (distance 7). We thus output our answer as 24 (6 + 7 + 4 + 7).

Design an algorithm that can solve this problem in less than 1 second. **Just outline your idea or write pseudocode.**

|  |
| --- |
| Highlight over the text to see the answers.  Run BFS on from each of the N labs to get all-pairs-shortest-path. Memorize this result using an adjacency matrix AM.  Try all (N-1)! permutations that start and end with vertex 2. (using next\_permutation function in STL? :D)  For each permutation, obtain the total distance by summing the distances between each consecutive pair, which can be found in AM. (e.g. 2->3->4->1->2, add the distance from 2->3, 3->4, 4->1, 1->2)  Output the minimum total distance found by one of the permutation  Pseudocode:  vector<vector<int>> AM(N, vector<int>(N, 0));  for (i in 0 to N-1) { // i is the lab number  AM[i] = BFS\_SSSP(i); // suppose BFS\_SSSP returns a vector of  // shortest distance from lab i.  }  int getDistance(int lab1, int lab2) {  return AM[lab1][lab2];  }  int shortestDist() {  int bestDistance = 10e6;  for ((N-1)! permutations of all possible lab orderings  that start and end with vertex 2) {  // permutation is of the form {2, \_\_, \_\_, …, \_\_, 2},  // length of permutation = N+1  int totalDistance = 0;  for (i from 0 to N) {  totalDistance += getDistance(permutation[i],  permutation[i+1]);  }    bestDistance = min(bestDistance, totalDistance);  }  return bestDistance;  } |

The time complexity of this algorithm is O( N! + RCN2 ).