CS302 Final 2020

Questions

The exam was on Canvas, and I used question banks for a lot of questions, so I won't have the exact exam here, but | will describe it:

Questions 1-6

These were Big-O questions, worth *two points* each.

- Let G be a weighted, directed graph with X nodes and Y edges. What is the Big-O running time of finding the minimum-weight path between two arbitrary nodes in the graph?
- Let G be a weighted, undirected graph with X nodes and Y edges. What is the Big-O running time of finding a minimum spanning tree of the graph?
- Let G be an unweighted, undirected graph with X nodes and Y edges. What is the Big-O running time of finding the shortest path between two arbitrary nodes in the graph?
- Let G be a weighted, directed graph with X nodes and Y edges. What is the Big-O running time of determining whether two nodes in the graph are connected?
- Let G be a weighted, directed acyclic graph with X nodes and Y edges. What is the Big-O running time of determining the minimum weight path between two arbitrary nodes in the graph?
- Let G be a weighted, directed graph with X nodes and Y edges. What is the Big-O running time of finding an augmenting path through the graph when implementing network flow with the Edmonds-Karp algorithm?

Questions 7-12

These were true-false questions on NP-completeness, worth *1 point* each.

- If G is an NP Complete problem, then G may be solved in exponential time with enumeration.
- If G is an NP Complete problem, then G must be framed as a "yes/no" question, and a "yes" answer must be verifiable in polynomial time.
- Let G be an NP Complete problem. If you can solve G in time O(T), then you can solve 3-SAT in time O(Tp), where p is a polynomial of the input size of G and 3-SAT.
- If G is an NP Complete problem, then G may not be solved in polynomial time.
- If G is an NP Complete problem, then you prove that G is NP-Complete by taking an instance of G, mapping it to a known NP-Complete problem like 3-SAT in polynomial time, and showing that if you can solve 3-SAT in polynomial time, then you can solve G in polynomial time.
- If G is an NP Complete problem, then G must be framed as a "yes/no" question, and a "no" answer must be verifiable in polynomial time.

Questions 13 and 14

1. These were "identify the sorting algorithm" questions, randomly chosen from a question bank. *Four points* each.

Suppose I am sorting the following vector:

```
69 33 22 77 56 3 79 55 25 14
```

After two passes of my sorting algorithm, the vector is:

```
22 33 69 77 56 3 79 55 25 14
```

Which sorting algorithm am I using?

2. Suppose I am sorting the following vector:

```
73 61 20 84 81 37 57 88 77 1
```

After two passes of my sorting algorithm, the vector is:

```
1 20 61 84 81 37 57 88 77 73
```

Which sorting algorithm am I using?

Question 15

Quicksort Pivot Selection - *4 points*.
Suppose I am sorting the following vector using quicksort:

```
43 24 35 15 90 98 74 47 89 66 54
```

If I am using the median-of-three pivot selection algorithm, what is the pivot?

Question 16

Quicksort Partition - *8 points*.
You are sorting the following string using quicksort (those are lower-case L's, not ones):

```
l p g s v l f f f e b w l
```

If you use the first character as a pivot, what will the string be right before you make the first recursive call?

Question 17

Minimum Spanning Tree - *8 points*.

The following list specifies the edges in an undirected, weighted graph with 10 nodes labeled A through J. In the answer box, please enter the edges in the minimum spanning tree of the graph. Specify an edge from X to Y as "XY" (but no quotes). You can separate your edges with spaces or put them on separate lines. Please don't put anything else in your answer except the edges.

```
Edge: AF -- weight  1
Edge: AJ -- weight  2
Edge: EH -- weight  4
Edge: FJ -- weight  5
Edge: EG -- weight  6
Edge: GH -- weight  8
Edge: EF -- weight  9
Edge: BI -- weight 10
Edge: CD -- weight 12
Edge: GJ -- weight 14
Edge: FH -- weight 15
Edge: AE -- weight 17
Edge: HJ -- weight 18
Edge: AG -- weight 20
```

```
Edge: EJ -- weight 21
Edge: CI -- weight 22
Edge: BD -- weight 23
Edge: FG -- weight 25
Edge: AH -- weight 26
Edge: BC -- weight 27
Edge: DI -- weight 29
Edge: DF -- weight 31
Edge: AI -- weight 32
Edge: DE -- weight 33
Edge: HI -- weight 34
Edge: CE -- weight 36
Edge: AC -- weight 37
Edge: AD -- weight 38
Edge: FI -- weight 39
Edge: EI -- weight 40
```

Question 18

Topological Sort - *8 points*.

Below is a specification of a directed, unweighted graph using adjacency lists. In the answer box, please give a listing of the nodes in the order of a valid topological sort. There may be many valid answers -- you only have to give one valid answer. Please answer by simply listing the nodes. You don't need spaces or commas or newlines between the nodes, but it's ok to do to. In fact, if the browser lets you, I'd recommend cutting and pasting the adjacency lists into your answer and then working through it from there. But that's me.

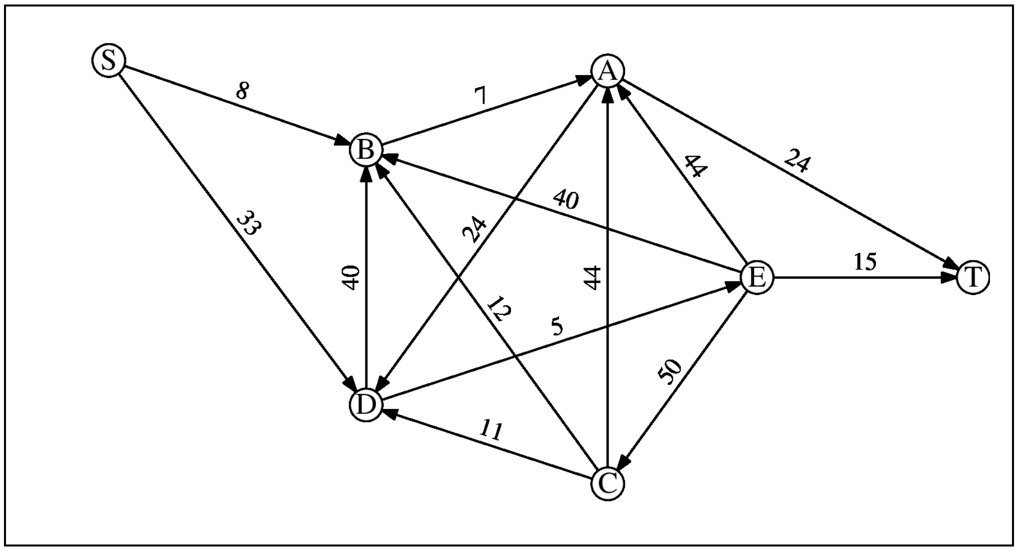
Please don't put any other stray stuff like numbers or comments in your answer. Remember that you can resize your entry window if it makes your life easier.

```
A: D C
B: D H A
C:
D:
E: C
F: D B
G: H C B
H:
I: D H C A
J: A I G E F
```

Questions 19-20

Network Flow - *6 points each*.

You are looking to find the max flow and min cut in the following graph:



In the box below, enter the maximum flow on the first line (just a number). And then enter the minimum cut on the next lines, one edge per line. No spaces; no punctuation. Use, for example, SA to represent the edge from S to A.

Question 21

Dijkstra's Algorithm - *7 points*.

You are working with a directed, weighted graph with the following adjacency matrix:

```
      A  B  C  D  E  F  G  H
A |   0 67 57 44 59 11 42 36
B |  68  0 50 13  2 67 55 30
C |  53 62  0  3 49 34  5 23
D |  55 18 46  0 13 25 30 26
E |  27 57 23 69  0 42  8 55
F |  59 50 61 65 45  0 66 39
G |  44  2 66 34 44 61  0 60
H |  63 25 38 10 68 11 18  0
```

You are in the middle of doing the shortest path calculation with Dijkstra's algorithms. You are maintaining a vector of the best known shortest paths, and the multimap that is central to Dijkstra's algorithm. The values of these data structures are as follows:

```
          A    B    C    D    E    F    G    H
Best:    68    0   50   13    2   67   55   30

Multimap:

key:   2    val: E
key:  13    val: D
key:  30    val: H
key:  50    val: C
key:  55    val: G
key:  67    val: F
key:  68    val: A
```

Your job is to tell me what the state of these two data structures is after the next pass of Dijkstra's algorithm.

Question 22

Programming BFS - *12 points*:

The *GF(64K)* graph is an unweighted, directed graph, where every node is labeled by a number from 0 to 0xffff. Every node with a label of *i* has two outgoing edges:
There is an edge to *(i+1)%0x10000*.

If *i* is less than 0x8000, then there is an edge to *i << 1*; otherwise, there is an edge to *(i << 1) ^ 0x1100b*. (in C++, the carat is XOR). You'll note that in either case, the edge is to a number that is less than or equal to 0xffff.

The graph does have two multi-edges. (Node 1 has two edges to 2, and Node 0xf006 has two edges to 0xf007). It's not really important, but I mention it in case you noticed it and were confused.

Write a procedure with the following prototype:

```
int shortest_path(int from, int to);
```

You don't have to error check -- you will be guaranteed that *from* and *to* are numbers between 0 and 0xffff. Your program should return the length of the shortest path from *from* to *to* in the *GF(64K)* graph. You'll note that there is always a path between every pair of nodes, so you don't need to worry about *from* and *to* being disconnected.

Question 23

Programming DP - *15 points*.

The following is a definition for a *B-String*:

- The following are *B-Strings*: `{}`, `[]`, `()`
- `{`*T*`}` is a *B-String* if and only if *T* is a *B-String* that does not start with '`{`'.
- `(`*T*`)` is a *B-String* if and only if *T* is a *B-String* that does not start with '`(`'.
- `[`*T*`]` is a *B-String* if and only if *T* is a *B-String* that does not start with '`[`'.
- `{`*TS*`}` is a *B-String* if and only if *T* and *S* are *B-Strings*.
- `(`*TS*`)` is a *B-String* if and only if *T* and *S* are *B-Strings*.
- `[`*TS*`]` is a *B-String* if and only if *T* and *S* are *B-Strings*.

So, for example, here are some *B-Strings:*

```
[]
[()]
[[][]]
[([]({}))]
```

These are not *B-strings*:

```
[][]            -- You can only have two concatenated strings inside a {, [ or (
[()()()]        --  You can't have three strings inside a []
[[]]            --  If you have one string inside a [], it cannot start with [
```

Write a dynamic program (obviously, in C++) that reads *n* from standard input, and prints the total number of *B-Strings* that are *n* characters in length. You only need to go to "step 2" in the four steps of dynamic programming.

Some examples:

```
UNIX> echo 1 | a.out
0
UNIX> echo 2 | a.out
3
UNIX> echo 3 | a.out
0
UNIX> echo 4 | a.out
6                # These are ([]), ({}), [()], [{}], {[]}, {()}
UNIX> echo 6 | a.out
39               # 13 begin with [, 13 begin with {, 13 begin with (
UNIX>
```

Please feel free to cut and paste the following to get you started:

```
#include <string>
#include <vector>
#include <iostream>
using namespace std;

class Bstring {
  public:
    vector < vector <long long> > Cache;
    long long bs(int len, int potential_starting_chars);
};

long long Bstring::bs(int len, int psc) {
// Do base cases.
// Create the cache if you need to.
// Check the cache
// You have two cases to count: strings like { T }, and strings
like { TS }.
}

int main() {
  int len;
  Bstring b;

  cin >> len;
  cout << b.bs(len, 3) << endl;
  return 0;
}
```