CS 141, Spring 2004 Practice Final

Name (first last): ____________________

Student ID: ____________________

- The exam will be closed book, closed notes, except for 1 page of notes.
- No electronic equipment allowed (cell phones, PDA's, computers...).
- Write legibly.
- Use pseudo-code or English to describe your algorithms.
- When designing an algorithm, you are allowed to use any algorithm or data structure we have covered in cs141 or in cs14 without giving its details, unless the question specifically asks for such details.

1. *+3 points for each correct answer, -2 points for each incorrect answer, 0 points for each question not answered.*

The running time of the following algorithm is $O(2^n)$: ☐ True ☐ False

```
1. void recurse(unsigned int n) {
2.   if (n <= 0) return;
3.   return recurse(n-1) + recurse(n-1) + recurse(n-1);
4. }
```

False, the recurrence tree has depth n and each node has 3 children, so the number of nodes in the tree is proportional to 3^n*.*

The running time of the following algorithm is $O(n \log n)$: ☐ True ☐ False

```
1. void loop(unsigned int n) {
2.   for (int i = 0;  i < n;  ++i)
3.     for (int j = 0; j < i*log(i);  ++i)
4.       print "Hi\n";
4. }
```

False. The running time is $\Theta(\sum_{i=0}^{n-1} i \log i) = \Theta(n^2 \log n)$*.*

The running time of the following algorithm is $O(n)$: ☐ True ☐ False

```
1. Array<int> F(-1);
2.
3. int fib(unsigned int n) {
4.   if (n <= 1) return n;
5.   if (F[n] == -1)  F[n] = fib(n-1) + fib(n-2);
6.   return F[n];
7. }
```

True. Suppose fib(n) is called. At most n calls to fib() in turn make recursive calls. Each of these calls makes at most 2 *calls to fib(). Thus, the total number of calls to fib() is at most* $3n$*.*

The largest number printed by mystery$(n, 0)$ is $\Theta(\log n)$: ☐ True ☐ False

```
1. void mystery(int n, int d) {
2.   std::cout << d << std::endl;
3.   if (n <= 1) return;
4.   mystery(n/2, d+1);
5. }
```

True. The recursion tree has depth $\Theta(\log n)$, and each call to mystery() that takes place at level i in the tree prints i.

Suppose $T(0) = 1$ and $T(n) = T(n-1) + T(n-3)$ for $n > 0$.
The recursion tree for T has $O(2^n)$ nodes. ☐ True ☐ False

True. The recursion tree has depth at most n and each node has at most 2 children, so the number of nodes is $O(\sum_{i=0}^{n} 2^i) = O(2^n)$.

Suppose $T(0) = 1$ and $T(n) = T(n-1) + T(n-3)$ for $n > 0$.
The recursion tree for T has $\Omega(2^{n/3})$ nodes. ☐ True ☐ False

True. Each node in the recursion tree at depth $n/3$ or less has at 2 children. Therefore the number of nodes is at least $\Omega(\sum_{i=0}^{n/3} 2^i) = \Omega(2^{n/3})$.

Suppose $T(0) = 1$ and $T(n) = 2 * T(n/2) + n$ for $n > 0$.
Then $T(n) = \Theta(n \log n)$. ☐ True ☐ False

True. The recursion tree has $\log n$ levels, and within each level the total work is n.

There exists a graph with 4 vertices such that every acyclic set of edges in the graph is a spanning tree. ☐ True ☐ False

False. Any graph with 4 vertices has an acyclic set of edges that is not a spanning tree. For example, the empty set of edges, or the set consisting of 1 edge.

Recall that an s, t-cut vertex is a vertex other than s or t whose removal separates s from t.
Every s, t-cut vertex in a graph is also a cut vertex. ☐ True ☐ False

True. A cut vertex is a vertex whose removal separates the graph (so that between some pair of vertices there is no longer any path). Removal of an s, t-cut vertex separates s from t. Therefore any s, t-cut vertex is also a regular cut vertex.

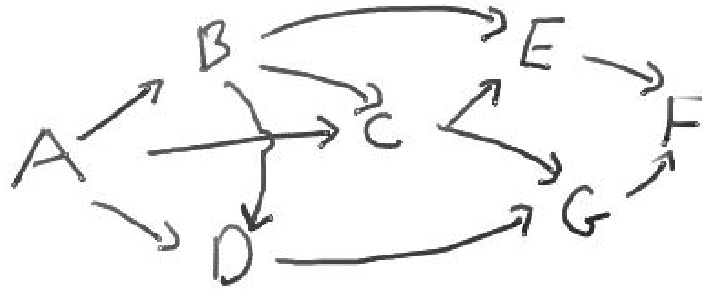
Dijkstra's algorithm runs in linear time. ☐ True ☐ False

False. All of the implementations that we have discussed take at least $\Omega(n \log n)$ time.

If the hash function is bad, accessing a key in a hash table can take $\Omega(n)$ time, where n is the number of items in the table. ☐ True ☐ False

True. If all n keys hash to the same bucket, then accessing the last key in that bucket takes $\Omega(n)$ time.

2.



Run depth-first search on the above graph. Start at A, and whenever you have a choice about which edge to explore next, choose the one that goes to the vertex that is earliest in the alphabet. Draw the DFS tree below, drawing the tree edges with solid lines and the non-tree edges using dashed lines. In your drawing, label each vertex with its dfs *post-order-number*.

The dfs tree has the following tree edges: A→B→C→E→F, C→G, B→D. The DFS-post-order numbers are F=1, E=2, G=3, C=4, D=5, B=6, A=7.

Below, give the corresponding topological ordering (the one that would be produced by the topological sorting algorithm based on DFS).

The reverse of the post-order numbering: A,B,D,C,G,E,F.

3. Describe the linear-time algorithm for finding shortest paths in an acyclic directed graph with edge weights.

   Illustrate the algorithm on an example.

   State the high-level idea of the algorithm, and explain why it takes only linear time. To be sure your answer is precise and detailed enough, you may want to give pseudo-code (if you have time).

   See ShortestPathsByDP on the course web page.

   The algorithm is to topologically sort the vertices, then consider the vertices in order, setting the shortest path distance from the source vertex s for each vertex v using the following recurrence:

$$D[s] = 0. \quad D[v] = \min_{u:(u,v)\in E} D[u] + wt(u,v).$$

4. Describe a linear-time algorithm for finding maximum-bottleneck paths in an acyclic directed graph with edge weights.

   Illustrate your algorithm on an example.

   State the high-level idea of the algorithm, and explain why it takes only linear time. To be sure your answer is precise and detailed enough, you may want to give pseudo-code (if you have time).

See homework 5 solutions. The algorithm is to topologically sort the vertices, then consider the vertices in topological order, then, for each vertex v, set the maximum-bottleneck of any path from s to v using the recurrence:

$$B[s] = \infty.$$

$$B[v] = \max\{\min\{B[u], wt(u,v)\} : (u,v) \in E\}$$

It takes linear time because we do $O(1 + degree(v))$ work for each vertex v, so the total work is $\sum_v O(1 + degree(v)) = O(n + m)$.

5. Describe a linear-time algorithm for finding a shortest-path tree from a given vertex s in any directed graph where every edge has weight 0 or 1.

   Illustrate your algorithm on an example.

   State the high-level idea of the algorithm, and explain why it takes only linear time. To be sure your answer is precise and detailed enough, you may want to give pseudo-code (if you have time).

Modify BFS as follows. When BFS is about to put a vertex w on the queue, if the vertex w was reached by a zero-cost edge from v, make the BFS put the vertex w on the front of the queue instead of the tail, and label distance[w] = distance[v] instead of distance[w] = distance[v]+1.

6. Given an array A[1..n] of integers (possibly negative), define partial sum

$$S(i,j) = \sum_{k=i}^{j} A[k] = A[i] + A[i+1] + \cdots + A[j].$$

We want a fast algorithm to compute the largest partial sum:

$$\max_{ij}\{S(i,j) : 1 \le i \le j \le n\}.$$

Define $M[j] = \max_i\{S(i,j) : 1 \le i \le j\}$. (That is, $M[j]$ is the maximum partial sum whose last term is $A[j]$.)

Prove the following recurrence: $M[j] = \max\{M[j-1] + A[j], A[j]\}$ for $j > 1$.

1. First, $M[j] \ge M[j-1] + A[j]$. This is because the maximum partial sum ending at $j-1$, together with j, is a partial sum ending at j.

2. Second, $M[j] \ge A[j]$. This is because $A[j]$ (by itself) is one of the partial sums ending at j.

3. Thus, $M[j] \ge \max\{M[j-1] + A[j], A[j]\}$.

4. Next, let $S(i,j)$ be a maximum partial sum ending at j, so $S(i,j) = M[j]$. If $i = j$, then $M[j] = A[j]$.

Otherwise, $S(i,j) = S(i,j-1) + A[j]$. Thus, there is a partial sum ending at $j-1$ that totals at least $M[j] - A[j]$. Thus, $M[j-1] \ge M[j] - A[j]$. Rewriting gives $M[j] \le M[j-1] + A[j]$.

Since either $M[j] = A[j]$ or $M[j] \le M[j-1] + A[j]$, it follows that $M[j] \le \max\{A[j], M[j-1] + A[j]\}$.

5. Combining the conclusions of parts (3) and (4) above, we get $M[j] = \max\{A[j], M[j-1] + A[j]\}$.

– *continued on next page* –

– continued from previous page –

Describe a linear-time algorithm, based on the recurrence, to compute $\max_{ij}\{S(i,j) : 1 \leq i \leq j \leq n\}$.

1. *M[1] = A[1]*
2. *For* $i = 2, 3, ..., n$ *do*
3. $\quad M[i] = \max\{A[i], A[i] + M[i-1]\}$
4. *Return* $\max\{M[i] : i = 1, 2, ..., n\}$