READ THIS NOW!

- Print your name in the space provided below.
- Unless a question involves determining whether given Java code is syntactically correct, assume that it is, and that any necessary `import`s have been made.
- There are 9 short-answer questions, priced as marked. The maximum score is 100.
- When you have finished, sign the pledge at the bottom of this page and turn in the test and your fact sheet.
- Aside from the allowed one-page fact sheet, this is a closed-book, closed-notes examination.
- No laptops, calculators, cell phones or other electronic devices may be used during this examination.
- You may not discuss this examination with any student who has not taken it.
- Failure to adhere to any of these restrictions is an Honor Code violation.

Name (Last, First) _____

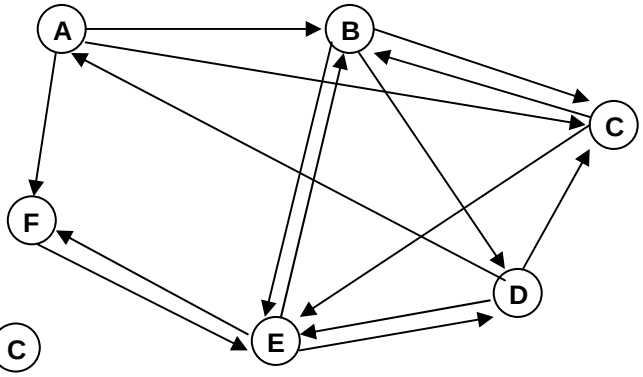
`printed`

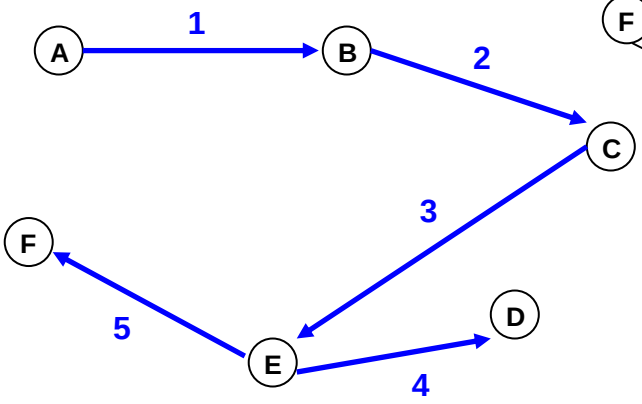
Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_____

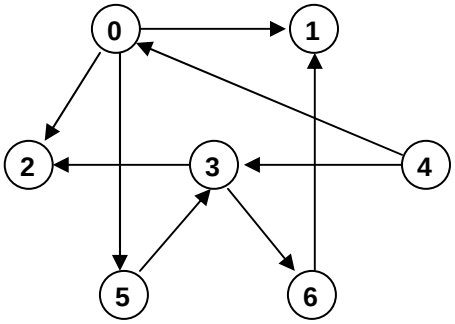
signed

1. [10 points] Draw a new graph showing the spanning tree for the graph below obtained by applying a depth-first traversal starting at node **A**. Visit neighbors in alphabetical order, and number the edges in the order they are added to the tree.





2. [10 points] The nodes in the graph below represent tasks that must be carried out. The edges represent prerequisite relationships; i.e., if there is an edge from i to j then i must be completed before j can be started. Use iterated depth first search to find a topological ordering of the nodes in the graph below; in your final answer, list the nodes linearly, in the order the corresponding tasks would be carried out. Show work if you want partial credit.



Let's say we start at node 0 and use a DFS approach:

Pass 1: 0 5 3 6 2 1

Pass 2: 4

Overall ordering: 4 0 5 3 6 2 1

Other solutions are possible.

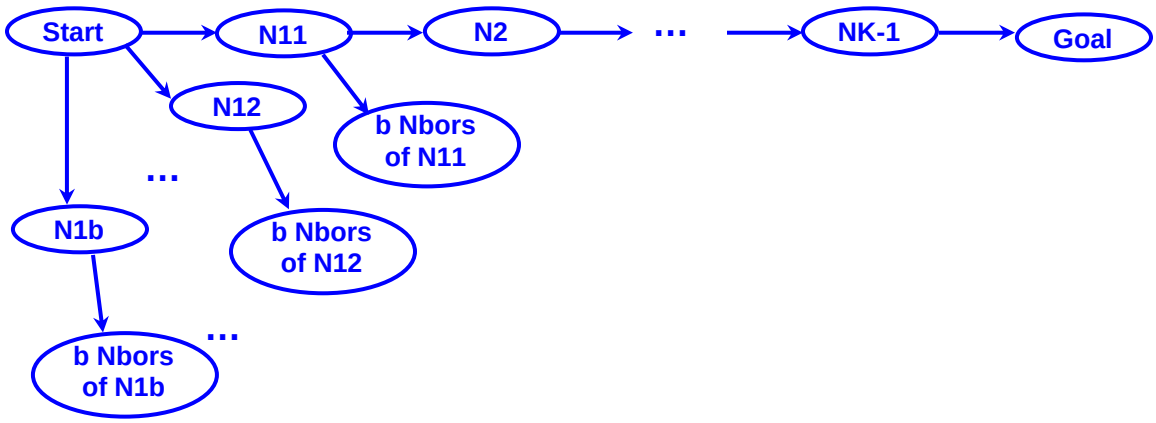
2. [16 points]] Many problems in Artificial Intelligence can be modeled as a search from a start vertex to a goal vertex in a very large graph. Consider a directed graph that contains *v* vertices, and each vertex in the graph has *b* outgoing edges. The **shortest path (i.e., minimum number of edges)** from the start vertex to the goal vertex contains *k* edges. The DFS or BFS algorithms can be used to search from the start vertex to the goal vertex. Express your answers to the following questions in terms of *v*, *b*, and *k*:

   a) What is the least number of vertices that could be visited by DFS (depth-first search) before reaching the goal vertex? Justify your answer.

From the problem statement, there IS a path from Start to Goal that contains *k* edges (and so *k*+1 nodes), and there is no shorter path from Start to Goal. So, in the best case, DFS would trace that path exactly and would visit *k* nodes before reaching Goal.

   b) What is the least number of vertices that could be visited by BFS (breadth-first search), assuming that no already visited nodes are encountered before finding the goal vertex?

Again, there IS a path from Start to Goal that contains *k* edges and *k*+1 nodes (like the diagram above). However, with BFS, we would visit every neighbor of Start before visiting any neighbors of N1, every neighbor of N1 and all the neighbors of all the other neighbors of Start before visiting any neighbors of N2, and so forth. So:



Remember we're assuming we don't see any repeated nodes until we reach Goal. So, before we reach Goal, we must have already visited every extended neighbor (k-1 levels deep) of the nodes Start through NK-2. (In the best case we'll get lucky and Goal will be the first neighbor of NK-1 that we visit.)

So, the smallest number of nodes we could have visited before reaching Goal is:

$$1+b+b^2+b^3+\cdots+b^{k-1}=\sum_{j=0}^{k-1}b^j=\frac{b^k-1}{b-1}$$

3. [16 points] Consider a hash table consisting of $N = 17$ slots, and suppose integer key values are hashed into the table using the hash function:

$$h(Key) = Key \quad \text{(the identify fn)}$$

Suppose that collisions are resolved using the probe function: $p(i) = i^2$

where i is the number of probes that have been attempted and the probe value is added to the base slot index (mod N).

Show the contents of the hash table after the following key values have been inserted in the given order:

29 16 12 34 21 46

If you want partial credit, show any calculations you perform.

Slot number	Contents
0	**34**
1	
2	
3	
4	**21**
5	
6	
7	
8	
9	
10	
11	**46**
12	**29**
13	**12**
14	
15	
16	**16**

```
29 → home slot is 29 % 17 == 12

16 → home slot is 16 % 17 == 16

12 → home slot is 12 % 17 == 12 full
      probe to 12 + 1 == 13

34 → home slot is 34 % 17 == 0

21 → home slot is 21 % 17 == 4

46 → home slot is 46 % 17 == 12 full
      probe to 12 + 1 == 13      full
      probe to 12 + 4 == 16      full
      probe to 12 + 9 % 17 == 4  full
      probe to 12 + 16 % 17 == 11
```

4. [8 points] Refer to the final table state in the previous question, and describe a specific scenario under which it would be logically necessary to use tombstones and explain why.

Suppose the value 21 is deleted from the table. If we simply marked slot 4 as EMPTY, then a subsequent search for 46 would fail, since the probe sequence for 46 would reach slot 4 and terminate.

In general, search in a hash table must terminate if an empty slot is found; otherwise looking for an element that was not in the table would degrade to Θ(table size), which is unacceptable.

So, simply converting a full slot to an empty slot when performing a deletion would break searches for any element that probed past that slot when it was inserted. Therefore, hash table slots must provide for at three distinct states (FULL, EMPTY, TOMBSTONE), where EMPTY indicates the slot has never contained and element and TOMBSTONE indicates the slot formerly contained an element that has been removed.

5. [8 points] Suppose that a hash table uses an array of size N, storing one element per array slot. If the search cost is too high, one might choose to increase the size of the array. Explain clearly what steps would be required to carry out this resizing operation, and why each step is necessary.

Step 1: Allocate a new, larger array to hold the data elements.

Step 2: Process the data elements from the old array, one by one, placing them into the new array.

BUT… this requires recomputing the hash value for each data element, unless the original hash values are stored in the table along with the data elements.

And, in any case, it requires recomputing the home slot for each data element since the hash value must be modded by the table size and that has changed.

6. [8 points] What is the single most important property of B-trees (including the variations) that make them more suitable for on-disk storage than the other kinds of trees we have discussed? Explain why.

Disk accesses are extremely slow, and it costs only a little more time to read a sector (or a few contiguous sectors) than to read a single byte.

Because B-tree nodes store a collection of logically related data values, the nodes are relatively "wide" and the tree is relatively short. This means that a traversal of the B-tree will require fewer, larger disk reads than if another kind of tree were used, but that each disk read will cost only a little more.

Hence, the overall cost of the disk I/O for a traversal will be greatly reduced.

---

7. [6 points] Consider building a skiplist of 16 nodes holding the integer values 0 through 15. What would the average search performance be like if every node in the skiplist was in level 3 (i.e., had 4 forward pointers)?

If every skip node is in level 3, then every node has only pointers to its immediate successor, so the performance would be identical to that of a simple, singly-linked list, with an average search cost of $\Theta(N)$.

---

8. [8 points] Suppose you have a max-heap containing *N* data records. In big-Θ terms, what is the worst-case running time for the `removeMax()` operation. Briefly explain the key reason why.

The `removeMax()` operation works by replacing the root value with the value in the last leaf, decrementing the heap size by 1, and sifting the new root value down until the max-heap property is restored.

Since the last leaf is always at index N-1, the only expensive part of this is the sifting down of the new root.

Since a heap is a complete binary tree, it is guaranteed that the height of the heap is no more than $\lceil \log(N+1) \rceil$.

So, it is not possible for the new root to sift down more than $\lceil \log(N+1) \rceil - 1$ levels.

Therefore, `removeMax()` is $\Theta(\log N)$ in the worst case.

9. [10 points] There are two different algorithms for performing some task; the first algorithm is $\Theta(N \log N)$ and the second algorithm is $\Theta(N^2)$, in the worst case. We would correctly expect that, for large values of N, the first algorithm would require the execution of fewer operations than the second algorithm. Is it possible, though, that for small values of N the first algorithm might actually require the execution of more operations than the second algorithm? Explain why or why not.

One answer:

If an algorithm is $\Theta(f(N))$, then, for large *N*, the actual number of operations that will be required in the worst case is bounded below by $kf(N)$, where *k* is some positive constant.

So, it is possible that the first algorithm has a much larger constant *k* than the second algorithm, and therefore that for small values of *N*, the lower bound for the first algorithm will actually be larger than the lower bound for the second algorithm.

In this case, even *k* were 10 for the first algorithm and 1 for the second, we see that $10N \log N$ is larger than N^2 for values of *N* less than 7.

Another answer:

The Θ-bound only captures the term that is dominant for large values of *N*. For small values of *N*, other terms may well be more important. Consider:

$$f(N) = N \log N + 1000N \quad \text{versus} \quad g(N) = N^2$$

For small values of *N*, the second term in $f(N)$ will be much larger than either $N \log N$ or N^2.