READ THIS NOW!

- Print your name in the space provided below.
- Unless a question involves determining whether given C++ code is syntactically correct, assume that it is. Unless a question specifically deals with compiler #include directives, you should assume the necessary header files have been included.
- There are 6 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) _____

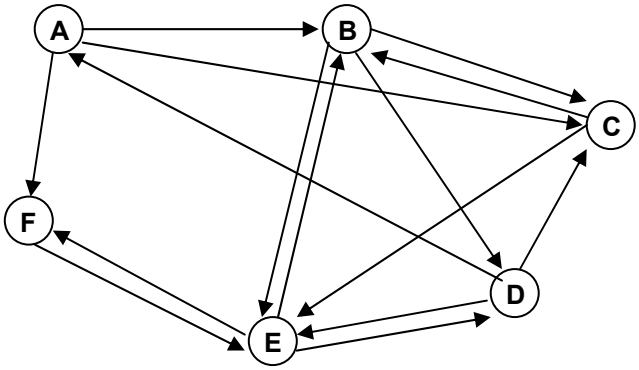
printed

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

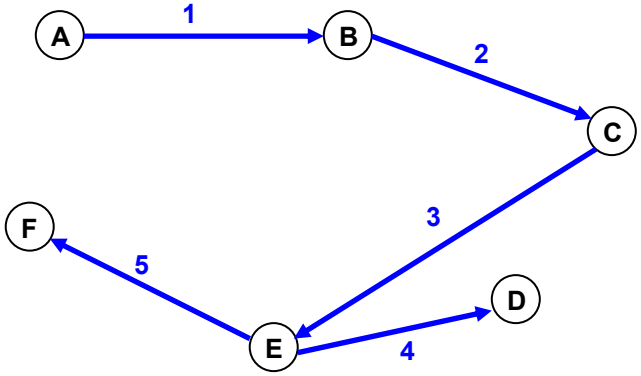
_____

signed

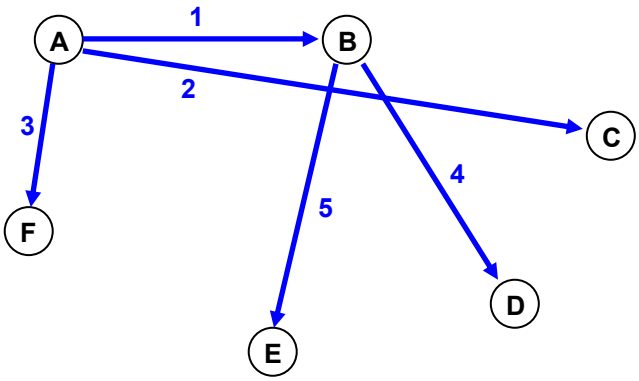
For questions 1 and 2, consider the following graph G_1:



1. [12 points] Draw the spanning tree for G_1 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. [12 points] Draw the spanning tree for G_1 obtained by applying a breadth-first traversal starting at node **A**. Visit neighbors in alphabetical order, and number the edges in the order they are added to the tree.



3. Suppose that a graph G has N vertices and E edges. Assume that pointers occupy four bytes and that vertex labels are 2-byte integers 0 through N-1.

   a) [10 points] Derive a formula for SA(N, E), the total space required to represent the graph using an adjacency matrix.

There's an N x N matrix; optimally each cell of that will occupy 1 byte, so that's N^2 bytes.

Evidently, the vertex labels must be stored somewhere, perhaps in an array, and that would be 2N bytes.

If you also count the two array pointers, that would be 8 more bytes, so the total would be:

$$SA(N,E) = N^2 + 2N + 8$$

   b) [10 points] Derive a formula for SL(N, E), the total space required to represent the graph using an adjacency list. (Assume that singly-linked lists are used.)

The adjacency list would be an array of N cells, each storing a vertex label and a head pointer for the adjacency list for that vertex. That amounts to 6N bytes.

For each edge, there will be a node in one adjacency list (if the graph is directed) or a node it two adjacency lists (if the graph is undirected). Each node must store a vertex label or index and a pointer, so each node is at least 6 bytes, and the nodes altogether occupy either 6E or 12E bytes, depending on whether the graph is directed or not.

So, we have:

$$SL(N,E) = 6N + 6E \text{ or } SL(N,E) = 6N + 12E$$

   c) [8 points] Assuming that the number of vertices is fixed (but not a specific value), determine how many edges the graph must have in order for the adjacency matrix to be more space-efficient than the adjacency list. (Your answer should be an inequality involving N and E.)

The answer here depends on which variation of each part above you give. For example, if you take my answer for part a (counting array pointers), and the first answer to part b, then you have:

$$SA(N,E) \le SL(N,E) \text{ iff } N^2 + 2N + 8 \le 6N + 6E$$
$$\text{iff } 6E \ge N^2 - 4N + 8$$
$$\text{iff } E \ge \frac{1}{6}N^2 - \frac{2}{3}N + \frac{4}{3}$$

4. Recall the two types of locality, *spatial* and *temporal*, that were discussed in class. Suppose a program accesses a largecollection of records stored in a file, and that the program exhibits both kinds of locality. Suppose also that the program incorporates a buffer pool.

   a) [12 points] Carefully describe how the program should use the buffer pool in order to take advantage of spatial locality.

   Spatial locality is exhibited if, when a particular record is accessed, there is a high probability that a nearby record will be accessed in the near future. Taking advantage of spatial locality requires that the program pre-fetch some nearby records when a record is retrieved from the file to satisfy a query. Naturally, the fetched records would be stored in the buffer pool. The replacement policy might seem to be a concern. However, nothing special needs to be done; if the client tells the buffer pool to load a collection of records, rather than just one, then each of those "extra" records will naturally reflect either a proper recent-use time (LRU) or a proper frequency count (LFU).

   b) [12 points] Carefully describe how the program should use the buffer pool in order to take advantage of temporal locality.

   Temporal locality is exhibited, if when a particular record is accessed, there is a high probability that same record will be accessed again in the near future. Simply using the buffer pool in the usual manner will take advantage of temporal locality so long as the replacement policy is reasonable.

5. [12 points] Aside from improving the selection of the pivot value, describe one technique for improving the performance of the quicksort algorithm and explain why it would make a difference.

   A number of suggestions were discussed in class, including:
   - **switch to a nonrecursive sorting algorithm on short sublists (say less than 16 elements); this will eliminate most of the recursive calls that would otherwise occur, and improve runtime efficiency**
   - **switch to an iterative implementation of the quicksort algorithm**
   - **find a more efficient algorithm for partitioning sublists; it is possible to reduce the number of copy operations to about 1/3 of those taken by the partitioning algorithm in the course notes**

6. [12 points] Suppose that you need to sort a list of N non-negative integer values. One objection to using binsort would be that it would require too many bins if the range of the data values was large. Describe a scenario in which, even if the number of bins is feasible, binsort would not really be Θ(N).

Note: the question is about binsort, not radix sort.

Let B be the number of bins; in any case B must be at least as large as N and typically B is much larger than N.

The pass of binsort through the data elements will leave then in sorted order, but scattered throughout the B bins. It is still necessary to collect the elements into a single list, and that will require a linear pass through the bins, which would be Θ(B).

That makes the total cost of binsort Θ(N + B). But this will only be Θ(N) if B is bounded by kN for some constant k. This, unfortunately, isn't guaranteed, even if the number of bins is feasible.

For a specific example, suppose that we are sorting a collection of N = 1000 integers in the range 0 to 1,000,000. We would need 10^6 bins, which might well be feasible, but the cost of the collecting pass would be on the order of N^2.

The size of the integers is of absolutely no concern here; binsort makes only one pass to sort the values and a second pass to collect them; and, if the integers are large but confined to a small range (e.g., 1000 integers between 900,000 and 1,000,000) then any intelligent use of binsort will take that into account and use a suitable number of bins (e.g., 100,000 instead of 1,000,000).

The occurrence of duplicate values is also irrelevant. Duplicates would simply go to the same slot; that's easily accommodated by making the slot a simple linked stack structure so insertion is still Θ(1). And, as far as binsort is concerned, it doesn't matter what order the duplicate elements occur in the slot. In the specific scenario described above, since the values being sorted are simple integers, all that's needed in each slot is actually just a counter.


Z
(SIGH)
IF IT'S NOT BEDHEAD, IT'S KEYBOARD FACE.