IE 170 Midterm Examination Practice Problems

Dr. T.K. Ralphs

1. You are given information that the running time of algorithm A is $O(n \lg n)$ and that the running time of algorithm B is $O(n^3)$. What does this tell you about the relative performance of A and B?

2. Show that $n \lg n \in O(n^{3/2})$. Is it also true that $n \lg n \in o(n^{3/2})$?

3. For each pair of functions f and g, state whether f is o, O, Θ, Ω, ω of g (choose as many as apply). Justify your assertion.

   (a) $f(n) = \lg n$, $g(n) = \ln n$.
   (b) $f(n) = n^{1/n}$, $g(n) = 1$.
   (c) $f(n) = n^{\lg n}$, $g(n) = n$.

4. Solve the following 3 recurrences. Assume that $T(1) = 1$.

   (a) $T(n) = T(n-2) + n/2$
   (b) $T(n) = 7T(n/3) + n^2$
   (c) $T(n) = T(\sqrt{n}) + 1$

5. Observe that the **while** loop of lines 5-7 of the insertion sort procedure on page 17 of CLRS uses a linear search to scan through the sorted subarray $A[1 \ldots j-1]$ to find the insertion point. Since the subarray is already sorted, can we improve the worst-case running time by using binary search to find the insertion point? If so, what would be the improved worst-case running time?

6. A *generalized queue* is a data structure that combines the functionality of a stack, in which the item most recently added to the list is the next to be removed, with that of a standard (FIFO) queue, in which the item least recently added to the list is the next to be removed. In a basic generalized queue interface, the following operations are supported.

   - `put`: add an item to the queue.
   - `getMostRecent`: return a copy of the most recently added item and then delete it from the queue.
   - `getLeastRecent`: return a copy of the least recently added item and then delete it from the queue.
   - `empty`: indicate whether the queue is empty or not.

   Note that we don't have operations to look at items in the queue without deleting them.

(a) What private members would be needed to implement a generalized queue **using an array** (not a linked list!) in order to allow all operations above to be performed in **constant time** (consider how you are going to implement the operations)?

(b) Write pseudo-code for a constant-time `put` operation.

(c) Write pseudo-code for a constant-time `getMostRecent` operation.

7. Suppose you are performing double hashing and you have a bug in your hash function code so that it always returns the same value. What happens when (i) the first hash value is wrong, (ii) the second hash value is wrong, (iii) both hash values are wrong.

8. Suppose you are given a binary search tree in which some of the nodes have equal keys. Describe an algorithm for locating all nodes with equal keys and analyze its worst case running time.

9. The following questions deal with recurrences. In all cases, assume $T(1) = 1$.

   (a) Solve the recurrence $T(n) = (T(\frac{n}{2}))^2$. Show your work.

   (b) Determine the solution to the recurrence

   $$T(n) = \alpha T(n/2) + n^2.$$

   for all values of $\alpha \geq 1$. In other words, state all solutions to this recurrence for different values of $\alpha \geq 1$ and state for which values of α each solution would arise. Justify your answer.

10. Consider the following function:

```
void foobar(int n)
{
   for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
         for (int k = j; k < n; k++){
            cout << "("<< i << ", " << j << ", " << k << ")" << endl;
         }
      }
   }
}
```

   (a) State succinctly what this function does.

   (b) What is the exact number of lines of text it outputs as a function of n? Show your work.

   (c) If the actual CPU time required to execute `foobar(10)` is 1 second, what is the predicted CPU time in seconds for execution as a function of n? Show your work.

11. The following questions test your knowledge of heaps. All three questions refer to an array-based implementation of a heap, like the one used in Laboratory 4.

   (a) Suppose the input array is already in sorted order. Is it also in heap order? In other words, will buildHeap() have any effect? Justify your answer (provide a counterexample if your answer to the first question is no).

(b) Recall that a stable sorting algorithm is one in which the relative order of items with duplicate keys is maintained after the list is sorted. Show by example that heap sort is not a stable sorting algorithm.

(c) The largest element in a max heap must be in position 0 and the second largest element must be in positions 1 or 2. Give a list of positions in a max heap of size 15 where the j^{th} largest element can and cannot appear, for $j = 2, 3, 4$. Justify your answer.

12. Consider the following algorithm for finding the median of an array of n integers. You may assume that the values in the array are distinct and that n is a power of 5.

- Arbitrarily insert the elements of the array into a $5 \times n/5$ matrix.
- Sort each column of the matrix.
- Recursively find the median of the middle row of the matrix.

(a) Is this algorithm correct?

(b) Write a recurrence for the worst-case running time of this algorithm and solve it. Justify all steps of your answer.

(c) Describe a version of the algorithm in which the array is first arranged into an $k \times n/k$ matrix and discuss how the worst-case running time changes.

13. The following questions refer to the mergesort algorithm. Assume that we are sorting from smallest to largest.

(a) Consider an implementation of mergesort that divides the array into k approximately equal parts, where k is a constant, instead of the usual two. Determine the worst-case running time of this algorithm if implemented using a straightforward recursion. Justify all your steps (Hint: Set up a recurrence and solve it using standard techniques.)

(b) The classical implementation of mergesort is considered *non-adaptive*, since the running time does not depend on the input array. In other words, every input array requires $\Theta(n \lg n)$ steps to sort, regardless of its order before sorting. Mergesort can be made partially adaptive by checking in the merge step whether the largest element of the first subarray to be merged is smaller than the smallest element of the second subarray. Use this idea to explain how to improve the best-case running time. What is the new best case and on what type of arrays is the best case achieved? Justify your answer.

(c) Consider a version of mergesort in which the array is divided into two pieces of random size, then recursively sorted as usual, and merged to form a completely sorted array. What is the worst-case running time of this algorithm? Justify your answer.

14. Consider the following recursive algorithm for sorting an array

```
sort(int *A, int n){
   if (A[0] < A[n-1])
      swap(A, 0, n-1);
   sort(A + 1, n-2);
   if (A[0] < A[1])
      swap(A, 0, 1);
   sort(A +1, n-1);
}
```

Here, `swap(A, i, j)` is a function that swaps the position of elements `i` and `j` in array `A`.

(a) Explain why this algorithm is correct.

(b) Write a recurrence for the worst-case running time. Justify your answer.

(c) Give a **lower bound** on the worst-case running time of this algorithm for an array of size n. Justify your answer. How does the running time compare to other algorithms we've discussed in class?

15. The following questions refer to the open addressing implementation of the bash table class from Laboratory 6.

(a) Below is the insertion function from the open addressing implementation of a hash table that you were given in Laboratory 6.

```
void HashTable::insert(Item* it, int& numComp)
{
  int i(0);                              // Count the number of comparisons
  int pos(it->hash());                   // Get the hash value

  // Try the sequence of addresses until an empty one is found
  do {
    if ( table_[pos] == 0){
      // Found an empty slot
      table_[pos] = it;
      break;
    }else if ( table_[pos]->getState() == Item::DELETED){
      // Found an empty slot
      delete table_[pos];
      table_[pos] = it;
      break;
    }else{
      // Go to the next slot
      ++i;
      pos = (pos + 1) % size_;
    }
  } while (i < size_);
```

```
  if (i < size_){
    ++numItems_;
  } else {
    std::cerr << "hash table overflow" << std::endl;
    abort();
  }
  numComp += i + 1;
}
```

Explain how this function would need to be modified in order to implement double hashing. Rewrite the exact lines of this function that need to be modified and also discuss any changes that would need to be made to other functions or classes to support double hashing.

(b) In double hashing, explain what would happen if the same function were used for both the first and second hash functions.