Worth: 11% **Due:** Monday February 1 (at class)

For each question, please write up detailed answers carefully. Make sure that you use notation and terminology correctly, and that you explain and justify what you are doing. Marks may be deducted for incorrect/ambiguous use of notation and terminology, and for making incorrect, unjustified or vague claims in your solutions.

1. [10 marks]
Consider the following two algorithms that find the two largest elements in an array $A[1..n]$, where $n >= 2$.

```
Algorithm 1
if A[1] >= A[2]
then L ← A[1]
     S ← A[2]
else L ← A[2]
     S ← A[1]
for i ← 3 to n
     if A[i] > L
     then S ← L
          L ← A[i]
     else if A[i] > S
          then S ← A[i]
end for
return(L,S)
```

```
Algorithm 2
if A[1] >= A[2]
then L ← A[1]
     S ← A[2]
else L ← A[2]
     S ← A[1]
for i ← 3 to n
     if A[i] > S then
          if A[i] > L
          then S ← L
               L ← A[i]
          else S ← A[i]
end for
return(L,S)
```

(a) Formally define an appropriate sample space and probability distribution with which to analyze the average case complexity of these algorithms.

Solution:

Let S_n be the set of all permutations of $\{1, \ldots, n\}$ and suppose that each is equally likely (i.e. has probability $1/n!$).

(b) Derive the expected number of element comparisons performed by Algorithm 1.

Solution:

Let X_i be the indicator variable that is 1 if $A[i] > \max\{A[j] \mid 1 \leqslant j < i\}$ and 0 otherwise. Recall from class that $E[X_i] = Prob[X_i = 1] = 1/i$.

In Algorithm 1, there is 1 element comparison before the loop and 1 element comparison (with L) each iteration of the loop. Furthermore, there is 1 element comparison (with S) each iteration of the loop in which the condition "$A[i] > L$" doesn't hold, i.e. in which $X_i = 0$.

Thus the expected number of element comparisons performed by Algorithm 1 is

$$1 + n - 2 + \sum_{i=3}^{n} E[1 - X_i] = 1 + n - 2 + (n-2) - \sum_{i=3}^{n} 1/i = 2n - 3 - \Theta(\log n).$$

(c) Derive the expected number of element comparisons performed by Algorithm 2.

Solution:

Let Y_i be the indicator variable that has value 1 if $A[i]$ is bigger than the second largest element of $\{A[j] \mid 1 \leqslant j < i\}$, and 0 otherwise. Then $E[Y_i] = Prob[Y_i = 1] = Prob\,[A[i]$ is the largest or second largest element in $A[1..i]] = 1/i + 1/i = 2/i$.

In Algorithm 2, there is 1 element comparison before the loop and 1 element comparison (with S) each iteration of the loop. Furthermore, there is 1 element comparison (with S) each iteration of the loop in which $A[i] > S$ i.e. in which $Y_i = 1$.

Thus the expected number of element comparisons performed by Algorithm 2 is

$$1 + n - 2 + \sum_{i=3}^{n} E[Y_i] = n - 1 + \sum_{i=3}^{n} 2/i = n + \Theta(\log n).$$

(Can you see how the gain of Algorithm 2 over Algorithm 1 obtained conceptually? Why does it make more sense to compare with S rather than with L?)

2. [10 marks]
Recall that the deterministic QuickSort algorithm we described in class chooses the first element of the input as pivot element. We have seen that this algorithm has worst-case running time of $\Omega(n^2)$ and an input showing this is one in which the input sequence is already sorted (increasing or decreasing). In this question we examine what happens when the input sequence is *almost* sorted.

We say that a sequence S is k-almost increasing if there are k elements, the exclusion of which leaves the sequence ordered in an increasing fashion. For example, the sequence 1,2,3,8,6,4,5,7,9,10 is 2-almost increasing as it is increasing once 8 and 6 are excluded. 4,3,2,1 is 3-almost increasing but not 2-almost increasing (check).

(a) Show that the best case running time of QuickSort on 1-almost increasing sequences is still $\Omega(n^2)$. In other words, there is a constant $c > 0$ so that for every sequence of size n that is 1-almost increasing, the number of comparisons that QuickSort performs on the sequence is at least cn^2.

We know that a sequence which is 1-almost increasing there is an elelment, let's say in location j, so that if we remove it the remaing sequence is increasing. We divide to cases.

- If $j = 1$ then it is the pivot in the qsort algorithm. Let's say that the rest of the elements split to sizes n_1 and n_2 and notice that $n_1 + n_2 = n - 1$. In addition we know that both parts of the partition are increasing sequences. So the running time is
$$n - 1 + \binom{n_1}{2} + \binom{n_2}{2}$$
(remember that $\binom{a}{2}$ is the number of pairs in a set of size a which is also expressed as $a(a-1)/2$). It can be easily seen that (at least) one of n_1 and n_2 is of size at least $(n-1)/2$ and so $\binom{n_1}{2} + \binom{n_2}{2} \geqslant (n-2)^2/8$, and so $n - 1 + \binom{n_1}{2} + \binom{n_2}{2} \geqslant n^2/8$.
- if $j > 1$ then we know that all elements, other than j perhaps, are bigger than the first element, which means that the partition will be to two parts of size 1 and $n - 2$ or to one part of size n -1and the bigger part is 1-almost increasing. We inductively assume that a sequence of length m where $1 < m < n$ which is 1-almost increasing requires number of comparisons which is at least $m^2/8$. By this assumption, we know have at least
$$n - 1 + (n-2)^2/8 = n^2 - n/2 + n - 1 + 1/2 \geqslant n^2/8$$
. Since for $m = 2$ once comparison is needed and $1 \geqslant 2^2/4$ this is the base case and we are done.

(b) (**bonus**) Show for a general k that the best case running time of QuickSort on k-almost increasing sequences is still $\Omega(n^2)$. Hint: Let $T(n, k)$ be the best case running time for k-almost increasing sequences. Show inductively that $T(n, k) \geqslant \frac{n^2}{2 \cdot 2^k}$.

To be added...

3. [10 marks]
Consider a binary tree T. Let $|T|$ be the number of nodes in T. Let x be a node in T, let L_x be the left subtree of x adn let R_x be the right subtree of x. We say that x has the "approximately balanced property", $APB(x)$, if $|R_x| <= 2|L_x|$ and $|L_x| <= 2|R_x|$.

(a) What is the maximum height of a binary tree T on n nodes where $ABP(root)$ holds?

Solution:

The worst case is when L_{root} and R_{root} are just single paths, so that $height(L_{root} = |L_{root}| - 1$ (and the same for R_{root}). We know $|L_{root}| + |R_{root}| = n - 1$, so it could be that $|L_{root}| = \frac{1}{3}(n-1)$ and $|R_{root}| = \frac{2}{3}(n-1)$ (or vice versa). Therefore, $height(R_{root}) = \frac{2}{3}(n-1) - 1$ and $height(T) = \frac{2}{3}(n-1)$.

(b) We call T an ABP-tree if $ABP(x)$ holds for every node x in T. Prove that if T is an ABP-tree, then the height of T is $O(\log n)$. More precisely, show that height(T) $\leqslant \log_2 n / \log_2 \frac{3}{2}$. **Solution:**

We'll prove that $|T| \geqslant \frac{3}{2}^{height(T)}$ (*) by induction on the height of T. If T has height 0 (it is a single node), then (*) certainly holds. Now consider T of height h. Assume, without loss of generality, that $height(L_{root}) \geqslant height(R_{root})$. Then $height(T) = height(L_{root}) + 1$. We know $|T| = |L_{root}| + |R_{root}| + 1$. Since we have $ABP(x)$ it follows that $|T| \geqslant \frac{3}{2}|L_{root}| + 1$. $L_{root} \geqslant (\frac{3}{2})^{h-1}$, so we get

$$n = |T| \geqslant \frac{3}{2}(\frac{3}{2})^{h-1} + 1 \geqslant (\frac{3}{2})^h.$$

Now that we hae proven (*), we just take the log of both sides:

$$h = height(T) \leqslant \log_{3/2} n = \log_2 n / \log_2 \frac{3}{3}$$

.

4. [10 marks]
Consider a binary search tree with *seven distinct* elements; the tree is perfectly balanced (that is, of height 2), and rooted at *root*. In this question, we will consider two slightly different methods for searching the tree for a key k. For each method, we will be interested in the *expected* time to search for k, when k is chosen at random from amongst the keys in the tree.

Solution Assumptions:

For Parts (a) and (b) of this question, assume (without loss of generality) that the 7 keys in the tree are $1, 2, 3, 4, 5, 6, 7$, where 4 is at the root, 2 and 6 are at depth 1, and 1,3,5,7 are at depth 2.

(a) Consider the following search method SEARCH1($root, k$):

```
SEARCH1(r,k) /*Return a pointer to a node with key k in subtree rooted at at r.*/
    if k = key(r) then
        return r
    elseif k > key(r) then
        return SEARCH1(rightchild(r),k)
    else
        return SEARCH1(leftchild(r),k)
    end if
end SEARCH1
```

Counting each "=" or ">" test as a comparison, what is the *expected* number of comparisons to do a search, where the expectation is over the random choice of k from amongst the keys in the tree, with all of them being equally likely? Briefly explain your reasoning and show your work.

Solution:

Let u_i be the number of comparisons made when searching for i using SEARCH1, where $1 <= i <= 7$. Since each number i is equally likely, the desired expected value is just $(1/7)\sum_{1<=i<=7} u_i$. It is easy to calculate that $u_4 = 1$, $u_2 = u_6 = 3$ and $u_1 = u_3 = u_5 = u_7 = 5$, so the answer is 27/7.

(b) Next consider the search method SEARCH2($root, k$):

```
SEARCH2(r,k)
/*Return a pointer to a node with key k in subtree rooted at at r.*/
  if k > key(r) then
    return SEARCH2(rightchild(r),k)
  elseif k = key(r) then
    return r
  else
    return SEARCH2(leftchild(r),k)
  end if
end SEARCH2
```

Again, suppose that each time we do a search, the element to be sought is chosen at random from amongst the seven elements in the tree. Compute the *expected* number of comparisons using SEARCH2.

Solution:

Let v_i be the number of comparisons made when searching for i using SEARCH2. Then $v_4 = 2$, $v_2 = 4$, $v_6 = 3$, $v_1 = 6$, $v_3 = 5$, $v_5 = 5$ and $v_7 = 4$, so the answer is 29/7.

(c) What if, instead of a perfectly balanced tree of height 2, we started with a perfectly balanced tree of much larger height, say 10, and considered the same two experiments above. Which of the two procedures, SEARCH1 or SEARCH2, would yield the smaller expected number of comparisons? Justify your answer.

Hint: Both algoithms have two possible comparisons (not counting the ones from the recursive calls). Consier the events that will lead to one or two comparisons in each of them and and their probabilities. Consider using recurrance relation for the average number of comparisions in the above algorithms.

Solution:

For a tree with only 7 nodes, SEARCH1 uses fewer (on the average) comparisons than SEARCH2. For large trees, however, we should expect SEARCH2 to do much better, since equality tests are very unlikely to be true most of the time.

More quantitatively, the behaviour of SEARCH1 can be described as follows:

- To search for the key at the root takes 1 comparison.
- Let x be an internal node of the tree. If it takes c comparisons to search for the key at x, then it takes $c + 2$ comparisons to search for key at the left child of x, and $c + 2$ comparisons to search for key at the right child of x.

The behaviour of SEARCH2 can be described as follows:

- To search for the key at the root takes 2 comparisons.
- Let x be an internal node of the tree. If it takes c comparisons to search for the key at x, then it takes $c + 2$ comparisons to search for key at the left child of x, and $c + 1$ comparisons to search for key at the right child of x.

To compare these two algorithms, say that the path from the root to node x contains l left edges and r right edges. Then to search for the key at x, SEARCH2 uses $2 + 2l + r$ comparisons while SEARCH1 uses $1 + 2l + 2r$ comparisons. When x is on the leftmost path (that is, $r = 0$), then SEACH1 uses 1 fewer comparison than SEARCH2. In all other cases, SEARCH2 is no worse than SEARCH1, and it is often much better.

Consider the perfectly balanced tree of height 10 with 2047 nodes. SEARCH1 saves 1 comparison on each of the 11 nodes on the leftmost path. However, in the tree rooted at the right child of the right child of the root (for example), there are 511 nodes, and for all of them, SEARCH2 is at least 1 better (and for most nodes much more than 1 better) than SEARCH1.