CS2223 Algorithms **B Term 2015**

Exam 1 **November 19, 2015**

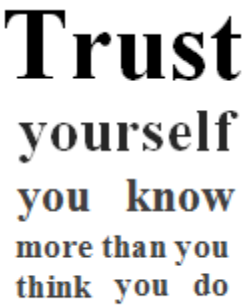
NAME: _RUBRIC FOR EXAM GRADERS_____

Instructions:

- Time allowed: 50 minutes
- Show your work and justify your answers
- Use the space provided to write your answers

Total	Q1	Q2	Q3	Q4	Q5

DO NOT OPEN EXAM UNTIL INSTRUCTED TO DO SO!!



Question 1. (20 pts.) Short Answer Questions

For each of the following statements:

+3 for correct answer

+2 for providing explanation

If the statement is *true*, circle **True** and explain why.
If the statement is *false*, circle **False** and explain *why the statement is false*.

Your explanations should be ***brief*** (using about *one sentence*), but complete.

(a) [5 pts.] **True / False** : Given an array of N unordered integers, you need at least N comparison operations to determine the maximum value.

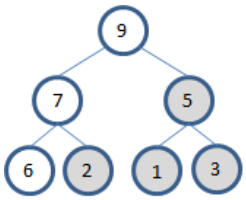
FALSE: You only need N-1 comparisons to find max in array of N

(b) [5 pts.] **True / False** : Given an array of N integers to sort, **Selection Sort** makes exactly N exchanges while sorting the array.

TRUE: Review Selection Sort algorithm and you will see that final step is always to exchange in place (p. 249)

(c) [5 pts.] **True / False** : In a max Heap containing seven distinct values, the bottom-most level contains its four smallest values.

FALSE: You know that each node in a Heap is greater than either of its two children. But with seven distinct values, you might have the following situation. Note that four smallest are highlighted in gray but are not ALL on final row.



(d) [5 pts.] **True / False** : When **QuickSort** sorts an array with an even number of values, it recursively calls **QuickSort** on two subproblems of exactly half the size.

FALSE: Subproblems are based on where pivot is placed. Alternatively, with an even number of values, the partitioning will place pivot in one place, which means an odd number of values form the sum of the two subarrays, so both can't be the same size!

Question 2. (20 pts.) Timing Analysis

Assume the existence of the **rank** function which returns the index location of target within a sorted array of elements (or returns **-1** if the target does not exist):

```
static int rank (int[] a, int target) {
  int lo = 0;
  int hi = a.length -1;
  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    if (target < a[mid])      { hi = mid - 1; }
    else if (target > a[mid]) { lo = mid + 1; }
    else                      { return mid; }
  }
  return -1;
}
```

Acceptable to use "1" as constant value for all time/cost which are constant. Sure you could use t0, t1, and so on, but in the end these will be approximated away. Note that the only non-constant operation is #9

(a) **[10pts.]** The **special(a)** method is executed on an array of size N. You can assume N is a power of 2. For each of the ten numbered statements below, determine the frequency of the statement and its cost (i.e., execution time).

```
      static int special (int[] a) {            frequency          time cost
1.      int N = a.length;                          _1_                _1_
2.      Queue<Integer> q = new Queue<Integer>();   _1_                _1_

3.      while (N > 1) {                          _log N_              _1_
4.        q.enqueue(N);                          _log N_              _1_
5.        N = N/2;                               _log N_              _1_
        }

6.      int value = 0;                             _1_                _1_
7.      while (!q.isEmpty()) {                   _log N_              _1_
8.        int x = q.dequeue();                   _log N_              _1_
9.        if (rank(a, x) != -1) {                _log N_          _t0 * log N_
10.         value++;                               _x_                _1_
          }
        }
        return value;                    x is unknown above
      }
```

(b) **[5 pts.]** What is the tilde approximation for the execution time of **special(a)** on an array of size N?

Now sum every product (freq*time) together and you get:

$3 + x + 5*\log N + t_0 * \log N * \log N$ which leads to $\sim t0*(\log N)^2$ since that is the most dominant exponent in the equation

(b) **[5 pts.]** What is the order of growth of the running time of **special(a)** on an array of size N?

Order of growth is $(\log N)^2$

Question 3. (20 pts.) Type Question

The Stack data type does not provide a **`boolean contains (Item it)`** operation. However, you are asked to implement the following static method to provide this functionality. I have given you an extra **`Stack`** object and an extra **`Queue`** object that you can use for extra storage in your answer.

(a) **[15 pts.]** You can provide Java code or describe your answer in pseudocode.

```
/** Determine whether stack contains the target integer. Upon return, stack must
  * contain its original contents in their original positions. */
 public static boolean contains (Stack<Integer> stack, Integer target) {
    Stack<Integer> extra = new Stack<Integer>();
    Queue<Integer> queue = new Queue<Integer>();
```

Multiple ways to handle this. Note that to make a copy of 'stack' you have to disturb it. Let's get started

```
+2      boolean found = false
+1      while (!stack.isEmpty()) {
+2       x = stack.pop()
+2       extra.push(x)
+2       if x == target {
+1         found = true;
+1         break;
         }
        }
+1      while (!extra.isEmpty()) {
+2       stack.push(extra.pop())
        }
+1      return found
```

Note: if you used a queue then you would have reversed the order of the elements on the stack. If you forgot to push back onto the stack the elements you popped, then you would not have satisfied the specification of 'contains'

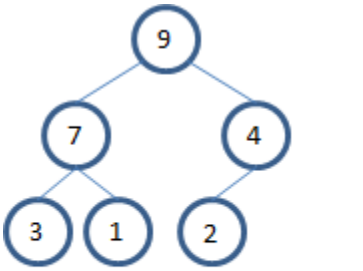
b) **[5 pts.]** If `stack` has N elements, compute the total number of `push` / `pop` calls needed in the worst case.

Question was a bit ambiguous. If you only count the operations on stack, then you push and pop N times each, for a worst case of 2N. If you also count the push and pop on extra, then you have to add an additional 2N for a total of 4N

Question 4. (20 pts.) Heap (you don't really need the code, but I'm providing just in case)

```
public class Heap {
  int[] pq; // Store in pq[1..N]
  int N;    // number of items in Heap

  public Heap (int initCapacity) {
    pq = new int[initCapacity + 1];
  }

  public void insert (int x) {
    pq[++N] = x;
    swim(N);
  }

  public int delMax() {
    int max = pq[1];
    exch(1, N--);
    pq[N+1] = null;
    sink(1);
    return max;
  }

  public boolean isEmpty() { return N == 0;  }
  public int size()        { return N; }

  void swim (int k) {
    while (k > 1 && less(k/2, k)) {
      exch(k, k/2);
      k = k/2;
    }
  }

  void sink (int k) {
    while (2*k <= N) {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
    }
  }
```

(a) **[12 pts.]** Assume you create **heap=new Heap(10)** and call **insert** with the following values in this order: **3**, **7**, **2**, **9**, **1** and **4**. Once all values are inserted, draw the tree representation of the final heap. Also show the array representation of **pq** that stores its values.

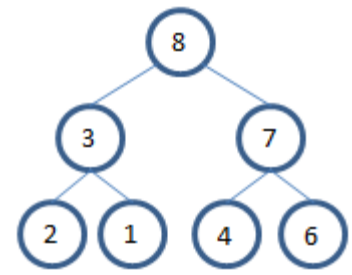


--	9	7	4	3	1	2	--	--	--	--

Note in the array representation you lose the 0th element, so it is shown as a --. Then the heap is constructed with 11 total spaces, the last four of which are unused. Finally you read off the heap left to right by level.

NOTE: There were two separate exams. The other exam had the same problem but all numbers were one greater (i.e., 4, 8, 3, 9, 2, 5 were added)

(b) **[8 pts.]** With this same heap, call **delMax** and then **insert** the values **6** and then **8**. Now draw the tree representation of the final heap. Also show the array representation of **pq** that stores its values.



--	8	3	7	2	1	4	6	--	--	--

Question 5. (20 pts.) Algorithm Question

The following is an example of an ***UpDown*** array of six unique values that contains a subarray **a[0..3]** of increasing integers followed by a subarray **a[3..5]** of decreasing values; note that **a[3]=12** is the maximum value in the array:

2	7	10	12	6	1
a[0]			a[3]		a[5]

To summarize, an ***UpDown*** array of size N has the following properties:

- **N ≥ 3** and all values in the array are different
- subarray **a[0..k]** is in ascending order and **k > 0**
- subarray **a[k..N-1]** is in descending order **k < N-1**

(a) **[15 pts.]** Design an algorithm that computes the index **k** containing the maximum value in an ***UpDown*** array. Performance must be ~ log N. **(To receive half credit on this part, the performance can be ~N).** Write your answer in pseudocode or Java.

(b) **[5 pts.]** In the worst case, what is the most number of comparisons of array items that your algorithm makes?

```
/** Return index k of maximum element a[k] in UpDown array. */
public static int maxUpDownArray (int[] a) {
```

There were at least five different ways of solving this problem, many of them discovered by students. Here is my implementation with points:

```
+1      int lo = 1                  // can avoid 0th and last one since they won't be largest
+1      int hi = a.length-2
+2      while lo <= hi              // must be <= otherwise strange case happens
+1        mid = lo + (hi-lo)/2
+3        if a[mid] < a[mid+1]      // means we are still ascending
+2          lo = mid+1              // can extend past mid, but also could have been just lo = mid
          else
+2          hi = mid -1             // interesting: no need to check again since must be descending
+3      return lo
```

Now how to compute number of comparisons? Each pass through, as with binary array search, there is a comparison, thus:

(1 + Floor(log n)) – and you can review the posted code solution to see that this is confirmed.

Note: FOR HALF CREDIT YOU COULD HAVE DONE SIMPLE LINEAR SEARCH UNTIL YOU SEE DESCENDING VALUES, WHICH WOULD REQUIRE N-1 COMPARISONS IN WORST CASE