Hwk #2  Fund Algs, Su03 , Dan Barrish-Flood
Due in class, hard copy only, date Thurs June 26 this seems very soon after your midterm, but working on this hwk counts as studying toward the exam!

Please use word processor, or *print neatly*. Of course, if you wish, you can word process all portions except those which require drawing of some sort, which you will please do neatly.

Steven will not accept your homework once they have been returned and the soln's posted. Usually, what this comes down to is, get them in by the due date in lecture. Recall the only way you can submit is by bringing hard copy to class, or handing hard copy to Steven *personally* at some other time (NOT in his mailbox, or under his door).

This hwk is 40 pts total

#1. [10 pts] Not only are heaps great for heapsort, but the heap data structure is perfect for implementing a priority queue (which maintains a set $S$ of elements, each associated with a value called a key). One application of a max-priority queue is to schedule jobs on a shared computer. This would use a max-priority queue, not a min-priority queue.

The 4 basic operations supported by a max-priority queue (as described in your text) are

INSERT(S, x) inserts the elements $x$ into the set $S$. This could be written as $S ← S U \{x\}$
MAXIMUM(S) returns the elements of $S$ with the greatest key.
EXTRACT_MAX(S) removes and returns the elements of $S$ with the greatest key.
INCREASE_KEY(S, x, k) increases the value of elements $x$'s key to the new value $k$,
which is assumed to be at least as large as $x$'s current value.

These routines all run in $O(\lg n)$ time, except MAXIMUM, which runs in $\Theta(1)$.

We use a max-heap data structure to support and implement those above routines, prefacing the names of those 4 routines above with "HEAP_", e.g, "HEAP_INCREASE_KEY", etc.

Check out this code for HEAP_INCREASE_KEY and MAX_HEAP_INSERT

```
HEAP_INCREASE_KEY(A, i, key)
1  if key < A[i]
2    then error “new key is smaller than current key”
3    A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5    do exchange A[i] ↔ A[PARENT(i)]
6    i ← PARENT(i)
```
Here is a picture of MAX_HEAP_INSERT in action (actually, the real work is done by HEAP_INCREASE_KEY.) The picture in your book looks better, I'm sure :)

Suppose the claim is made: "Here is a $\Theta(n \lg n)$ algorithm for eliminating duplicates from an n-element array: insert the elements into a heap one by one, using the MAX-HEAP-INSERT algorithm modified so that when a value equal to the one being inserted is seen, the algorithm stops without inserting that value, then moves on the next element."

Show by means of an example that this algorithm won't work (at least not in $\Theta(n \lg n)$ time). Give your answer by drawing a small heap, and a few sentences in English, no need to get down to level of pseudo-code (unless you really want to). And no need for any kind of mathematical justification or proof!

#2. [5 pts] Give a $\Theta(n \lg n)$ comparison-based algorithm for removing all duplicates from an array. You cannot make any assumptions about the order of the elements in the input array. Just a few sentences in English is all I'm looking for.

#3. [5 pts] Given an array of $n$ elements, each colored either red or blue, devise an "efficient" (as efficient as you can come up with) in-place algorithm to put all the red elements before the blue ones. Extend your approach to handle three colors. An answer in English sentences will do fine. Probably 5 sentences (more or less) is all you need.
#4. [10 pts] Consider the code for COUNTING_SORT (as seen in the slides from class or in your text). Suppose you change line 9 to be re-written as:

9  for j ← 1 to length[A]  // instead of   ...length[A] downto 1

This modified algorithm still works properly (i.e. it still results in a sorted output array). Show that this modified algorithm is not stable. Explaining this in English sentences is fine; but you may write pieces of pseudo-code and/or diagrams if you feel that will clarify your explanation.

#5. [10 pts]
Solve the following recurrence:

\[ T(2) = 2 \]
\[ T(n) = nT(n-1) + n, \quad n \geq 3 \]

**Hint:** use summation factor \(1/(n!)\) to get into telescoping form.

**Hint:** you should find yourself faced with a summation that contains a factorial in the denominator, which is difficult to solve exactly. But think about how to bound the sum. The lower bound is easy. For the upper bound, compare \(1/(n!)\) with \(1/(2^n)\), then you'll be able to arrive at a final solution of the form \(\Theta(f(n))\).