

Extra problems on **Order of magnitude and sorting**

Given February 21, due never.

1. We are taking a web based survey of popularity of songs. There are n songs in the contest. At any moment, a vote can come in for a song. Let $v(k)$ be the number of votes for song k at a particular time. When a new vote for k comes in, we do, among other things, $v[k]++$. We want always to know which song has the most votes, so we want a method `most()` that returns, in $O(1)$ time (i.e. no time) the song with the most votes at that moment. For this purpose, we need some other data structure such a heap or a sorted array. When a new vote comes in, we need to adjust the locations of the songs (or references to them) in this data structure. Your algorithm for doing this, called `vote(k)` should take $O(1)$ work if there are were no ties when the vote came in. Of course, there must at some point be ties, for example, in the beginning when every song has no votes. Design your algorithm so that if there are j songs tied with song k when a vote for k comes in, then the work is $O(\log(j))$ (also “no time”). *Harder:* Design a fancier algorithm that stores more information that can update the data structure in $O(1)$ work no matter what ties might be present.
2. Here is a combination of bubble sort and heapsort called beepsort:

```
a[n] is a list of n elements to be sorted
create heap h[k]; // a heap with k elements max.
inorder = FALSE;
while( not inorder ) { // if you don't know the elements are in order, make a pass
    inorder = TRUE; // if no element actually moves, the elements were in order
    for p = 0, k-1 // start by inserting the first k elements into the heap.
        h.insert(a[p]);
    for j = k, n-k-1 {
        t = h.deleteMax(); // take the largest element from the heap and . . .
        if ( t != a[j] ) inorder = FALSE // check whether an element moved . . .
        a[j] = t; // put it back in the array and . . .
        h.insert(a[p++]); } // refill the heap with the next element of a.
    for j = n-k, n-1
        t = h.deleteMax(); // same as before, but now there are
        if ( t != a[j] ) inorder = FALSE; // no elements left to refill the heap.
        a[j] = t; }
} // end while loop
```

- A.** Show that the algorithm terminates once the elements have gotten in order and not before.

- B.** Find the work per pass in the worst case as a function of n .
 - C.** Show that if every element is no more than k positions out of order then one pass will suffice. That is, if i is the index of an element in the original array and j is the index of the same element after sorting, then $|i - j| \leq k$.
 - D.** Show that at most n/k passes are needed.
- 3.** Any integer, x , in the range $0 \leq x \leq 2^n - 1$ may be written uniquely in terms of its binary “digits” (i.e. bits), $a(k)$, $k = 0, \dots, n - 1$.

$$x = a(0) + 2 \cdot a(1) + 4 \cdot a(2) + \dots + 2^{n-1} \cdot a(n - 1) ,$$

where each $a(k)$ is either 0 or 1.

- A.** If x is represented by a and y by b , write an algorithm that computes the representation, c , of $x + y$ from a and b . If n is at all large (say $n > 100$) you will not be able simply to add x to y and then convert to binary. Your algorithm should take $O(n)$ work.
- B.** Design an algorithm to find c representing $x \cdot y$ in $O(n^2)$ work using repeated addition. In view of this you may be impressed that there is an $O(n \log(n))$ algorithm for finding c (it’s in CLR somewhere), and that modern processor chips do a 32 bit integer multiply in one cycle and five hundred million such multiplies in a second.