

# MIDTERM SOLUTIONS

CIS 102 - Spring 03

Warmuth

NAME: \_\_\_\_\_

Student ID: \_\_\_\_\_

This exam is closed book and closed notes.

Show partial solutions to get partial credit.

Make sure you answered all parts of the question.

If your answers are not written legibly, you won't get full credit.

Clarity and succinctness will be rewarded.

Question 1: \_\_\_\_\_(out of 15)

Question 2: \_\_\_\_\_(out of 10)

Question 3: \_\_\_\_\_(out of 10)

Question 4: \_\_\_\_\_(out of 15)

Question 5: \_\_\_\_\_(out of 15)

Question 6: \_\_\_\_\_(out of 20)

Question 7: \_\_\_\_\_(out of 15)

Total: \_\_\_\_\_(out of 100)

1. What is the solution to the following recurrence in  $\Theta$  notation. Assume  $T(1) = 1$  in each case. Use the Master Theorem below or any other method. Sketch how you arrived at your answer

a)  $T(n) = 16T(n/4) + n^3$

$E = \frac{\log_2 16}{\log_2 4} = 2$ .  $n^3 \in \Omega(n^{2+\epsilon})$  - e.g. for  $\epsilon = 0.5$ .  $n^3 \in O(n^{2+\delta})$  - e.g. for  $\delta = 1.5$  and  $1.5 > 0.5$ . Case 3 of the Master Theorem applies, therefore  $T(n) \in \Theta(n^3)$

b)  $T(n) = 7T(n/2) + n^2$

$E = \frac{\log_2 7}{\log_2 2} = \log_2 7 \approx 2.8$ .  $n^2 \in O(n^{2.8-\epsilon})$  - e.g. for  $\epsilon = 0.3$ . Case 1 of the Master Theorem applies, therefore  $T(n) \in \Theta(n^{\log_2 7})$ .

c)  $T(n) = 2T(n/2) + \sqrt{n}$

$E = \frac{\log_2 2}{\log_2 2} = 1$ .  $\sqrt{n} = n^{0.5} \in O(n^{1-\epsilon})$  - e.g. for  $\epsilon = 0.2$ . Case 1 of the Master Theorem applies, therefore  $T(n) \in \Theta(n)$ .

d)  $T(n) = T(4n/5) + n$

$E = \frac{\log_2 1}{\log_2(5/4)} = 0$ .  $n \in \Omega(n^{0+\epsilon})$  - e.g. for  $\epsilon = 0.5$ .  $n \in O(n^{0+\delta})$  - e.g. for  $\delta = 1.5$ ,  $1.5 > 0.5$ . Case 3 of the master theorem applies, therefore  $T(n) \in \Theta(n)$ .

Master Th.: Assume  $T(n) = bT(n/c) + f(n)$  and  $T(1)$  a non-negative constant. Let  $E = \log_c(b) = \frac{\lg(b)}{\lg(c)}$  be the critical exponent.

i) If  $f(n) \in O(n^{E-\epsilon})$  for some positive  $\epsilon$ , then  $T(n) \in \Theta(n^E)$ .

ii) If  $f(n) \in \Theta(n^E)$ , then  $T(n) \in \Theta(f(n) \log n)$ .

iii) If  $f(n) \in \Omega(n^{E+\epsilon})$ , for some positive  $\epsilon$ , and  $f(n) \in O(n^{E+\delta})$  for some  $\delta \geq \epsilon$ , then  $T(n) \in \Theta(f(n))$ .

2. Using l'Hôpital rule, show that  $(\ln n)^2 \in O(n^{0.1})$ .

We need to find the limit  $\lim_{n \rightarrow \infty} \frac{(\ln n)^2}{n^{0.1}}$ . Both functions are differentiable and both go to infinity, therefore L'Hôpital's rule applies - the limit will be equal to the limit of their derivatives:

$$\lim_{n \rightarrow \infty} \frac{(\ln n)^2}{n^{0.1}} = \lim_{n \rightarrow \infty} \frac{((\ln n)^2)'}{(n^{0.1})'} = \lim_{n \rightarrow \infty} \frac{2 \cdot \ln n \cdot \frac{1}{n}}{0.1 \cdot n^{-0.9}} = \lim_{n \rightarrow \infty} \frac{2 \cdot \ln n}{0.1 \cdot n^{0.1}} =$$

Now we apply the L'Hôpital's rule again:

$$\lim_{n \rightarrow \infty} \frac{2 \cdot \ln n}{0.1 \cdot n^{0.1}} = \lim_{n \rightarrow \infty} \frac{(2 \cdot \ln n)'}{(0.1 \cdot n^{0.1})'} = \lim_{n \rightarrow \infty} \frac{2 \cdot \frac{1}{n}}{0.1 \cdot 0.1 \cdot n^{-0.9}} = \lim_{n \rightarrow \infty} \frac{2}{0.1^2 \cdot n^{0.1}} = 0$$

The limit is zero, therefore  $(\ln n)^2 \in O(n^{0.1})$ .

3. Show that  $\sum_{i=1}^n \log(i) = \Theta(n \log n)$ .

First approximate the sum from above, by setting all elements to the largest -  $\log n$ :

$$\sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n = n \cdot \log n$$

Then approximate the sum from below. First we split the sum into two:

$$\sum_{i=1}^n \log i = \sum_{i=1}^{\frac{n}{2}-1} \log i + \sum_{i=\frac{n}{2}}^n \log i$$

Then discard the first sum (its bigger than zero). In the second sum replace all elements with the smallest one -  $\log(n/2)$ :

$$\sum_{i=1}^n \log i = \sum_{i=1}^{\frac{n}{2}-1} \log i + \sum_{i=\frac{n}{2}}^n \log i \geq 0 + \sum_{i=\frac{n}{2}}^n \log\left(\frac{n}{2}\right) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \frac{1}{2} \cdot n \cdot \log n - \frac{n}{2}$$

So we have an upper and lower bound with the same asymptotic order -  $n \log n$ . Therefore  $\sum_{i=1}^n \log(i) \in \Theta(n \log n)$

4. Quicksort is not very fast for sorting a small number of keys. A variant of Quicksort does nothing (i.e. it simply returns) whenever the number of keys is  $\leq 10$ .

After the variant of Quicksort is called on the entire array, insertion sort is run once on the entire array.

Insertion sort uses  $\Theta(n^2)$  number of comparisons in the worst case. Why does this bound not apply?

What is the cost of insertion sort in this case and why?

Quicksort always partitions elements in this way - all elements in the subproblem (array range) on the left are smaller than all in the subproblem on the right. So if we stop the recursion at subproblems of size  $\leq 10$  we will have many small segments which are ordered among themselves, but the elements inside the segments are not ordered. The fact that the array is split into disjoint segments of size  $\leq 10$ , ordered among themselves means that any element is at most 10 positions away from its place in the sorted array (elements don't need to cross segment boundaries). In formal notation, after modified quicksort quits, we have:

$$S_i = A[l_i..r_i] = \{A[l], A[l+1], \dots, A[r]\}, l_i \leq r_i$$

$$A = S_1 \sqcup S_2 \sqcup \dots \sqcup S_k$$

$$\forall i : 1 \leq r_i - l_i + 1 \leq 10$$

$$S_i < S_j \equiv r_i < l_j$$

$$\forall S_i < S_j \forall x \in S_i, y \in S_j : x \leq y$$

Insertion Sort takes an element and starts going to the left in search of its place. We have that in order to insert an element in our almost-sorted array Insertion Sort has to do no more than 10 comparisons. Thus the total number of comparisons for Insertion Sort in this case is  $\leq 10n$  - which is linear.

In the worst case Insertion Sort is  $O(n^2)$ , because the element might be inserted anywhere in the sorted portion of the array which is growing -  $\sum_{i=1}^n i = \frac{n^2+n}{2}$ . In our case we have an additional restriction on where an element can go in the sorted portion of the array -  $i \leq 10$ .

5. For a max heap, give a high-level description of an algorithm for retrieving the second largest element from the heap.

Your algorithm should use as few comparisons as possible !!!

(Assume  $n$  is the number of elements in the heap.)

How many comparisons does your algorithm take?

Why is your algorithm correct?

(Exploit the partial order of the heap).

Hint: A “very very small” number of comparisons suffice!

A maximizing heap has the partial order property -  $Root \geq Left$ ,  $Root \geq Right$  if the appropriate child exists. By induction this means that root of the heap is maximum element in the heap. Root is the largest element - than the second largest element will be in either the left subheap or the right subheap. Since these subheaps are maximizing heaps too, we have that the second largest element is one of the immediate children of the root (Note: second largest element is the element that appears in position  $n - 1$  in the array sorted in increasing order). One comparison suffices to find it (assuming heap is stored in the array  $A$ , also assuming that -1 is a flag value indicating that no child exists):

```
if (A[2] == -1) return "There are less than 2 elements in the heap!"
if (A[3] == -1) return A[2];
if (A[2] >= A[3])
    return A[2];
else
    return A[3];
```

6. Give an adversary lower bound on the number of peeks required for the following problem:  
Given a string of  $n$  bits, determine whether the bit string has more zeros or has more ones (i.e. output the majority bit).  
For simplicity assume  $n$  is odd.  
Give an adversary strategy and prove your lower bound.  
The lower bound should be as large as possible !!!

The adversary strategy that forces the algorithm to peek at the entire array goes as follows. Adversary tracks the total number of questions it has answered so far. For the odd-numbered questions he answers 1, for the even-numbered questions he answers 0. If some question repeats the adversary answers in consistent manner and does not count the repeat question.

We know that  $n$  is odd. Assume that some algorithm stops before asking all possible  $n$  questions. If the algorithm has asked an even number of questions, then it has seen an equal number of ones and zeroes and there is an odd number of unchecked elements in the array. These elements can be set as to make the majority bit either zero or one - by adding an even number of ones and zeroes until only one unspecified bit remains. The value of that bit will then determine the majority bit. If the algorithm has asked an odd number of questions, than so far it has seen more ones than zeroes, but there still is an even number of unchecked bits in the array. The adversary can again assign them to make any possible outcome. One zero bit is added to even things out and then the previous strategy works.

Basically, since  $n = 2k + 1$  we make  $2k$  bits alternating zeroes and ones and then the last bit controls the outcome. So any algorithm that performs less than  $n$  peeks is incorrect. Thus  $n$  is the lower bound for this problem.

7. Given a list of  $n$  elements, an element of a list is a *majority* if it appears more than  $n/2$  times.

Design an algorithm that is linear in the number of comparisons in the worst case that will find a majority if one exists, and reports that there is no majority if no such element exists.

Give a high-level description of your algorithm.

Reason your time bound.

(possibly using the Master Th.).

Example: 3,6,8,3,3 has majority 3 and 4,3,2,3,4 has no majority.

Hint: You may assume any algorithm we covered in class as a submodule. Consider the median element! For simplicity assume that  $n$  is odd.

Let's consider the median for the array that has the majority element. Remember that median is an element that is greater than or equal to at least half of all elements and also less than or equal to at least half of all elements (alternatively, if the array is sorted in increasing order than median is  $A[\lfloor \frac{n}{2} \rfloor]$ ). If some element occurs more than  $n/2$  times, than immediately we have that this element has to be the median - in a sorted array there will be a continuous sequence of more than  $n/2$  of these elements and it has to touch the middle. E.g.:

3 1 3 2 3 3 3 5 7 - unsorted array

1 2 3 3 3 3 3 5 7 - sorted array

We know that there exists a linear-time algorithm for finding the median. Once we found it, all that remains is to verify that it actually is the majority element - by counting how many times the median value occurs in the array. This is a linear-time procedure - compare every element to the median value and count - takes  $n$  comparisons. If median occurs less than  $n/2$  times this means that there is no majority element in the array. Total number of comparisons required is about  $4n$  ( $3n$  is the current best algorithm for finding the median plus  $n$  for the check). Pseudocode follows ( $A$  is the array, `median` is an external function that finds the median):

```
m = median(A);
c = 0;
for (int i = 1; i <= N; i++)
    if (A[i] == m) c++;
if (c > N/2)
    return m;
else
    return "No majority element exists";
```