Massachusetts Institute of Technology
6.046J/18.410J: Introduction to Algorithms
Professors Piotr Indyk and Bruce Tidor

Quiz 1
4 March 2002

Quiz 1

- Do not open this quiz booklet until you are directed to do so.
- This quiz ends at 3:55 P.M.
- When the quiz begins, write your name on the top of ***EVERY*** page in this quiz booklet, because the pages will be separated for grading.
- Write your solutions in the space provided. If you need more space, write on the *back* of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem.
- Plan your time wisely. Do not spend too much time on any one problem. Read through all of them first and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- When describing an algorithm, describe the main idea in English. Use pseudocode only to the extent that it helps clarify the main ideas.
- Good luck!

Problem	Points	Grade
1	16	16
2	30	22
3	19	17
4	19	12
Total	80	67

Name: DAN PORTS

Please circle your TA's name and recitation:

Nitin **Brian** **Mihai** **Yoav**

10am **11am** **12pm** **1pm** **2pm**

Problem 1. Recurrences **[16 points]**

Solve the following recurrences (provide only the $\Theta()$ bounds). You can assume $T(n) = 1$ for n smaller than some constant in all cases. You *do not* have to provide justifications, just write the solutions.

4 • $T(n) = 9T(n/3) + n^{1.1}$

$\beta = \log_3 9 = 2$ $\quad n^{1.1} = O(n^{2-\varepsilon})$ – case 1

$T(n) = \Theta(n^2)$

4 • $T(n) = T(9n/10) + \log n$

$\beta = \log_{10/9} 1 = 0$ $\quad \log n = \Theta(n^0 \log^1 n)$ – case 2 extended

$T(n) = \Theta(\log^2 n)$

4 • $T(n) = T(\sqrt{n}) + \log n$

Let $n = 10^m \Rightarrow m = \log n$

$T(10^m) = T(\sqrt{10^m}) + m = T(10^{m/2}) + m$. Let $S(m) = T(10^m)$

$S(m) = S(m/2) + m$ $\quad \beta = \log_2 1 = 0$, $m = \Omega(m^{0+\varepsilon})$ and reg cond holds, case 3

$S(m) = \Theta(m)$

$\Rightarrow T(m) = \Theta(\log n)$

4 • $T(n) = 4T(n/3) + n^2$

$\beta = \log_3 4$ – between 1 and 2 $\quad n^2 = \Omega(n^\beta)$ $\quad 4\frac{n^2}{9} < \frac{1}{2}n^2$ reg cond ok

Case 3

$T(n) = \Theta(n^2)$

Problem 2. True or False, and Justify **[30 points]** (6 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. Your justification is worth more points than your true-or-false designation.

T F The solution to the recurrence

$$T(n) = 2^n T(n-1)$$

is $T(n) = \Theta((\sqrt{2})^{n^2+n})$ (assume $T(n) = 1$ for n smaller than some constant c).

$T(n) = 2^n \cdot 2^{n-1} \cdot 2^{n-2} \cdots 2$

$= 2^{n+(n-1)+(n-2)+\cdots+1}$

$= 2^{n^2 - \frac{n^2+n}{2}} = \Theta(2^{n^2/2 + n/2})$

$= \Theta(\sqrt{2}^{n^2+n})$

T F There exists a comparison-based sorting algorithm that can sort any 4-element array using at most 5 comparisons.

A 4 element array has $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ permutations

Each comparison can provide 3 results, so the decision tree can have $3^5 \geq 24$ leaves.

Even if each comparison only provided 2 outcomes, this would be $2^5 = 32 > 24$ leaves.

T F Checking if there is a pair of equal elements in an array $A[1 \ldots n]$ requires $\Omega(n^2)$ time in the comparison-based model, because we need to test equality for every pair of elements.

5 It can be done in $O(n \log n)$ time by sorting the array with merge-sort ($\Theta(n \log n)$) then iterating through the array checking each successive pair of elements for equality. This will identify a pair since the array is sorted, and it can be done in linear time, so this algorithm is $\Theta(n \log n)$

T F Consider an implementation of Paranoid Quicksort which accepts a partition of an array $A[1 \ldots n]$ as balanced only if the lengths of the two subarrays resulting from the partition differ by at most ± 3. The expected running time of this Quicksort implementation is $O(n \log n)$.

5 For a randomly selected pivot to fail at creating a balanced partition, it must be chosen from the interval $(1, \frac{n}{2}-2)$ or $(n/2+2, n)$ if n is even or $(1, \lfloor \frac{n}{2} \rfloor - 2)$ or $(\lfloor n/2 \rfloor + 2, n)$ if n is odd (anything but the middle 5 elements). The odds of failure are $\frac{n - n/2 - 2 + n/2 - 2 + 1}{n} = \frac{n-5}{n}$ so the probability of success is $5/n$, meaning the expected number of attempts will be $1/5n = n/5 = \Theta(n)$. Each attempt requires $\Theta(n)$ operations to partition the array, so finding a balanced partition takes $\Theta(n^2)$ time. The algorithm cannot have $O(n \log n)$ expected runtime.

T F An array $A[1 \ldots n]$ is called *bitonic*, if there exists t such that $A[1 \ldots t]$ is sorted in the increasing order, and $A[t \ldots n]$ is sorted in the decreasing order.
There is a linear time comparison-based algorithm for sorting bitonic arrays.

6 First, find t. This can be done by iterating through the array w/ i from 1 to n. When we find the first i such that $A[i] > A[i+1]$, then we have found $t = i$. This is $\Theta(n)$. Next reverse $A[t \ldots n]$, which can be done in $\Theta(n)$.

Finally, merge $A[1 \ldots t]$ and $A[t \ldots n]$ in linear time to produce the sorted array, using the same merge algorithm from merge-sort. The algorithm is $\Theta(n)$.

T **F** An array $A[1 \ldots n]$ is called *pytonic*, if every element stored at even position is smaller than its two neighboring elements (formally, for every integer $k > 0$, $A[2k-1] > A[2k]$ and $A[2k] < A[2k+1]$). E.g., the array $< 5, 1, 6, 2, 7, 3 >$ is pytonic.

1

There is a linear time comparison-based algorithm for sorting pytonic arrays.

0

Problem 3. Superselection **[19 points]**

Prof. Noitceles Repus proposes the following divide-and-conquer algorithm for selection, that he calls SUPERSELECTION. Suppose we have an array $A[1 \ldots n]$ of distinct elements and we want to select the element with rank i. We first apply SUPERSELECTION recursively on $A[1 \ldots n/2]$ to compute its median M_1. Then we apply SUPERSELECTION on $A[n/2+1 \ldots n]$ to compute its median M_2. Let $m = \min(M_1, M_2)$ and $M = \max(M_1, M_2)$, We use m and M to partition our array A into three parts:

1. Part B: contains all elements $< m$
2. Part C: contains all elements in $[m, M]$
3. Part D: contains all elements $> M$

Finally, we apply SUPERSELECTION recursively on one of the parts B,C or D in an "appropriate" way.

The following (incomplete) pseudocode defines the algorithm more formally:

```
    Superselection(A, start, end, rank)
1   if(start = end)
2         return A[start]
3   M_1 ← Superselection(A, start, ⌊(end + start)/2⌋, ⌊(end − start)/4⌋)
4   M_2 ← Superselection(A, ⌊(end + start)/2⌋ + 1, end, ⌊(end − start − 1)/4⌋)
5   m ← min(M_1, M_2)
6   M ← max(M_1, M_2)
7   (B_end, C_end) ← 3Partition(A, m, M)
8   newstart ← ... ; newend ← ... ; newrank ← ...
9   return Superselection(A, newstart, newend, newrank)
```

(a) **[4 points]** Specify what "appropriate" means. I.e., replace line 8 in the above code with new code, that properly specifies the variables *newstart*, *newend* and *newrank*.

(b) **[4 points]** Argue that the partitioning procedure can be implemented in linear time.

(c) **[11 points]** Analyze the worst-case running time of the algorithm.

B contains the first (at least n/4) elements, $< m$. C contains n/2 middle elements, D contains n/4 last elements $> M$,

a) if (start + rank) $\leq$ Bend;
newstart = start, newend = Bend, new rank = rank;

3/4 If rank > Bend, and rank $\leq$ Cend
newstart = Bend+1, new end = Cend, newrank = rank - Bend.
If rank > Cend
newstart = Cend + 1, new end = end, newrank = rank - Cend.

b) The partitioning procedure can be done by considering each element. Each element i can be compared against m and M_2. If A[i] < m, A[i] should be assigned to B; if not, then check $M < A[i]$. If this second comparison is true then A[i] is assigned to D; if not, A[i] → C. So it takes at most 2 comparisons plus the placement of the element in the appropriate subarray, which we can do in constant time. Partitioning the entire array requires this to be done n times; it is $\Theta(n)$.

3/4

c) Each super-selection requires 2 recursive calls to find the medians M_1 and M_2, a 3Partition call, and a recursive call to generate the return value. Each median-finding call searches $n/2$ elements, and the return-value recursive call searches either $n/4$ or $n/2$. With n as the length (end-start+1) of the array, we have the recurrence

11/11

$$T(n) = \underbrace{2T(n/2)}_{\text{median-finding}} + \underbrace{\Theta(n)}_{\text{3Partition}} + \underbrace{T(n/2)}_{\text{return value generation}} + \underbrace{\Theta(1)}_{\text{everything else}}$$

this will overwrite values in A!

→

Simplifying, $T(n) = 3T(n/2) + \underbrace{\Theta(n)}_{f(n)}$.

$\beta = \log_2 3 =$ something between 1 and 2.

$f(n) = O(n^{\beta-\epsilon})$, so Master Thm case 1 applies

and $T(n) = \Theta(n^{\beta}) = \Theta(n^{\log_2 3})$

Problem 4. General matrix multiplication **[15 points]**

From Lecture 3, we know that two $n \times n$ matrices A and B can be multiplied in time $O(n^{\log_2 7})$, beating the simple cubic-time algorithm. However, what if we need to multiply two non-square matrices ? For example, what if we are given two *rectangular* matrices:

- matrix A, with n rows and m columns, and
- matrix B, with m rows and n columns

such that $m \leq n$? The naive algorithm for computing $A \times B$ would take $n^2 m$ time. Is there a better algorithm ?

Your goal is to give an algorithm that has running time $o(n^2 m)$ (i.e., is asymptotically faster than the naive algorithm) whenever $m = \omega(1)$ (i.e., m is superconstant in n). You can use Strassen's algorithm as a block box.

Your solution should follow the following outline:

(a) [10 points] Give an algorithm as specified above, assuming that m divides n.

(b) [5 points] Show that your algorithm can be easily modified to handle the case where m does not divide n.

a) If m divides n, we can write A as a block matrix $\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_{n/m} \end{bmatrix}$ and B as $\begin{bmatrix} B_1 & B_2 & \cdots & B_{n/m} \end{bmatrix}$, where each block is $m \times m$. The output is $C = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1,n/m} \\ C_{21} & C_{22} & \cdots & \\ \vdots & \vdots & \ddots & \\ C_{n/m,1} & & & C_{n/m,n/m} \end{bmatrix}$ where $C_{ab} = A_a \cdot B_b$. Each element of C is a $m \times m$ matrix product that requires $O(m^{\log_2 7})$ via Strassen's algorithm. There are $\left(\frac{n}{m}\right)^2$ such elements, so the algorithm is $O\left(\frac{n^2}{m^2} m^{\log_2 7}\right)$

$= O(n^2 m^{\log_2 7 - 2}) = O(n^2 m)$

✓

b. In this case we can write the input matrices as $A = \begin{bmatrix} A_1 \\ A_2 \\ A_{\lfloor n/m \rfloor} \\ \hline a_{m \cdot \lfloor n/m \rfloor} \\ \vdots \\ a_m \end{bmatrix}$, $B = \left[B_1 \cdots B_{\lfloor n/m \rfloor} \;\vdots\; b_{m \cdot \lfloor n/m \rfloor} \cdots b_m \right]$

– that is, a set of $\lfloor n/m \rfloor$ $m \times m$ block matrices followed by the left-over $n - m\lfloor n/m \rfloor$ row or column ($m \times 1$ or $1 \times m$) vectors.

Then we compute the block products as before, and fill in the edges by multiplying the row and column vectors. ?

The block product portion takes $\Theta(n^2 m^{\log_2 7 - 2})$,

and there are less than m left-over vectors in each input matrix, which we are computing the dot product of.

These can be computed in $\Theta(m)$ time each.

Could take m^3

+

2