CS240 Assignment 1

Nissan Pow

May 25, 2006

Problem 2

2. (a)

$n^3 \in O(2^n)$

$$\begin{aligned}
\lim_{x\to\infty} \frac{n^3}{2^n} &= \lim_{x\to\infty} \frac{3n^2}{(\ln 2)2^n} && (L'Hôpital's\ Rule)\\
&= \lim_{x\to\infty} \frac{6n}{(\ln 2)^2 2^n} && (L'Hôpital's\ Rule)\\
&= \lim_{x\to\infty} \frac{6}{(\ln 2)^3 2^n} && (L'Hôpital's\ Rule)\\
&= 0 \\
\Rightarrow n^3 \in O(2^n)
\end{aligned}$$

□

2. (b)

$n \lg\lg n \in \Omega(n \lg n)$

$$\begin{aligned}
\lg n &< n && \forall n \geq 1\\
\Leftrightarrow \lg\lg n &< \lg n && \forall n \geq 1\\
\Leftrightarrow n\lg\lg n &< n\lg n && \forall n \geq 1\\
\Rightarrow n\lg\lg n &\notin \Omega(n\lg n)
\end{aligned}$$

□

2. (c)

$1 + n + n^2 + \ldots + n^{k-1} \in \Theta(n^k)$

$$\begin{array}{rcll} 1 + n + n^2 + \ldots + n^{k-1} & \leq & k \times n^{k-1} & \forall n \geq 1 \\ \Rightarrow 1 + n + n^2 + \ldots + n^{k-1} & \in & O(n^{k-1}) & \\ \Rightarrow 1 + n + n^2 + \ldots + n^{k-1} & \notin & \Omega(n^k) & \\ \Rightarrow 1 + n + n^2 + \ldots + n^{k-1} & \notin & \Theta(n^k) & \end{array}$$

□

2. (d)

$2^n \in O(3^n)$

$$\begin{array}{rcll} 2 < 3 & \Rightarrow & 2^n \leq 3^n & \forall n \geq 0 \\ & \Rightarrow & 2^n \in o(3^n) & \\ & \Rightarrow & 2^n \in O(3^n) & \end{array}$$

□

2. (e)

$2^n \in O(3^n)$

$$\begin{array}{rcll} 2^n & \in & o(3^n) & \text{(from 1.d)} \\ \Rightarrow 2^n & \notin & \Omega(3^n) & \end{array}$$

□

2. (f)

$2^n \in \Theta(3^n)$

$$\begin{array}{rcll} 2^n & \notin & \Omega(3^n) & \text{(from 1.e)} \\ \Rightarrow 2^n & \notin & \Theta(3^n) & \end{array}$$

□

2. (g)

Given three positive functions f, g, h,

$$f, g \in O(h) \Rightarrow max\{f, g\} \in O(h)$$

$$\begin{aligned}
f &\leq c_0 h \qquad c_0 > 0, n_0 \geq 0, n \geq n_0 \\
g &\leq c_1 h \qquad c_1 > 0, n_0' \geq 0, n \geq n_0' \\
max\{f, g\} &\leq f + g \qquad \text{(since f,g are positive functions)} \\
&\leq c_0 h + c_1 h \\
&= (c_0 + c_1) h \\
\Rightarrow max\{f, g\} &\in O(h)
\end{aligned}$$

□

2. (h)

Given a positive function h and k positive functions $f_1, \ldots, f_k \in \Omega(h)$, then

$$\sum_{i=1}^{k} (p_i f_i) \in \Omega(h) \qquad \forall p_i > 0$$

$$\begin{aligned}
f_1 &\geq c_1 h \qquad c_1 > 0, n_0 \geq 0, n \geq n_0 \\
f_2 &\geq c_2 h \qquad c_2 > 0, n_0' \geq 0, n \geq n_0' \\
&\vdots \\
f_k &\geq c_k h \qquad c_k > 0, n_0^{(k-1)} \geq 0, n \geq n_0^{(k-1)} \\
\Rightarrow p_1 f_1 &\geq p_1 c_1 h \qquad \text{(since } p_1 > 0\text{)} \\
\Rightarrow p_2 f_2 &\geq p_2 c_2 h \qquad \text{(since } p_2 > 0\text{)} \\
&\vdots \\
\Rightarrow p_k f_k &\geq p_k c_k h \qquad \text{(since } p_k > 0\text{)} \\
\Rightarrow \sum_{i=1}^{k} (p_i f_i) &\geq \sum_{i=1}^{k} (p_i c_i h) \\
&= h \sum_{i=1}^{k} (p_i c_i) \\
\Rightarrow \sum_{i=1}^{k} (p_i f_i) &\in \Omega(h)
\end{aligned}$$

□

2. (i)

Given two positive functions f and g,

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

$$\begin{aligned}
f \in \Theta(g) &\Leftrightarrow 0 \le c_1 g \le f \le c_2 g \\
&\Leftrightarrow 0 \le c_1 g \le f \text{ and } 0 \le f \le c_2 g \\
&\Leftrightarrow 0 \le g \le \frac{f}{c_1} \text{ and } 0 \le \frac{f}{c_2} \le g \qquad \text{since } c_1, c_2 > 0 \\
&\Leftrightarrow 0 \le \frac{f}{c_2} \le g \le \frac{f}{c_1} \\
&\Leftrightarrow g \in \Theta(f)
\end{aligned}$$

□

Problem 3

3. (a)

Given A (an array of k sorted arrays), the algorithm will output all the elements of A[1] that occur in at least t-1 other arrays from A[2] to A[k] inclusive.

3. (b)

In its worst case, for each element of A[1] the algorithm will need to search all the arrays from A[2] to A[k]. So the resulting worst case complexity is:

$$\begin{aligned}
n_1 \underbrace{(\lg n_2 + \ldots + \lg n_k)}_{k-1} &\le (n_1)(k-1)(\lg m), \qquad m = max\{n_2, \ldots, n_k\} \\
&\le (n)(k-1)(\lg n) \qquad n = max\{n_1, \ldots, n_k\} \\
&\in O(n \lg n)
\end{aligned}$$

□

Note: binary search $\in \Theta(\lg n)$

3. (c)

The algorithm can be improved by using a hashtable to facilitate element lookups in constant time (assuming that we can hash/dehash in constant time). Building the entire hashtable will take:

$$\begin{aligned}\sum_{i=2}^{k}(cn_i) &\leq \sum_{i=2}^{k}(cm) \qquad m = max\{n_2, \ldots, n_k\}, c \text{ is the time taken to hash a single element} \\ &= c(k-1)m \text{ time}\end{aligned}$$

Then for each element of A[1], we check to see whether the element occurs at least t-1 times in the other k-1 arrays. Using the previously built hashtable, this should take us at most k-1 checks. Therefore, for a single element we can determine whether it occurs at least t-1 times in O(1) time. Hence the resulting complexity in the worst case is as follows:

$$\begin{aligned}\underbrace{c(k-1)m}_{\text{build hash}} + \underbrace{(k-1)n_1}_{\text{check each element of A[1]}} &\leq c(k-1)n + (k-1)n \qquad n = max\{n_1, \ldots, n_k\} \\ &= (c+1)(k-1)n \\ &\in O(n) \qquad \text{since } c, k \text{ are constants}\end{aligned}$$

So we have improved the running time in the worst case to O(n).

Problem 4

4. (a)

For a single array, A, of size n, we can determine if $x \in A$ by using binary search, which has tight bound $\Theta(n)$. Therefore a lower bound on the worst case complexity of any deterministic algorithm is $\Omega(n)$, since binary search is among the fastest deterministic search algorithm for sorted arrays (as of today).

4. (b)

For k fixed, we will need to search every array, so the worst case complexity of any deterministic algorithm will be:

$$\begin{aligned} \sum_{i=1}^{k} \lg n_i &\geq \sum_{i=1}^{k} \lg m \qquad m = min\{n_1, \ldots, n_k\} \\ &= k \lg m \qquad m = min\{n_1, \ldots, n_k\} \\ &\in \Omega(\lg m) \qquad m = min\{n_1, \ldots, n_k\} \end{aligned}$$

Problem 5

5. (a)

```
main()
    seek(1, n, A)

seek(start,end,A)
    // start,end are integers, 1 ≤ start, end ≤ n
    // A is an array of n non-negative integers
    mid = ⌈(end/2)⌉
    x = A[start] * A[start+1] * ... * A[ mid ]
    The question states that we are only able to take the product of n/2 elements.
Thus if start-mid+1 ≤ ⌈(n/2)⌉ then simply fill in the outstanding elements with
some elements from the non-zero set. The non-zero set is the set of elements
whose product is non-zero, which can be obtained from previous computations.
    if (start = end) then
        if (x = 0) then output start // we have found the zero element
        else output NOT_FOUND // there isn't a zero element in A
        exit // we're done searching
    if (x = 0) then // zero element is somewhere between start-mid
        seek(start, mid, A)
    else // it's somewhere between (mid+1) - end
        seek(mid+1, end)
```

The algorithm is pretty much the same as binary search, except that we're using products to determine whether to search the lower half or the upper half. First we call seek(1,n,A) which starts the recurstion. Then in seek(), it will take the product of the lower half of the elements (A[start], ..., A[$\lceil(\frac{n}{2})\rceil$]). If this product is 0, then we know that the zero element lies somewhere between *start* and $\lceil(\frac{n}{2})\rceil$, since the product of a set of numbers is equal to zero iff one of the numbers is zero. So then we call seek recursively on *start* and $\lceil(\frac{n}{2})\rceil$. Otherwise we know that the zero element should lie in the upper half, so we call seek on $\lceil(\frac{n}{2})\rceil + 1$ and *end*. Eventually we will reach the point where we

have narrowed down the position to a single element (ie when $start = end$). Then we can simply check to see if the product is zero. If it is, then we have found the position of the element, otherwise zero doesn't exist in the array. The algorithm is guaranteed to take at most $O(\log n)$ products, since the height of the recursion is of order $O(\log n)$.

5. (b)

Assumptions: We have a bit-vector that will automagically store the results of the products for us, and we can compute binary to decimal in constant time (needed to output the index of the zero element in decimal). If the result of a product is 0, then the bitvector for that product# will be zero, otherwise it will be 1.

1. x = A[1] * ... * A[$\lceil(\frac{n}{2})\rceil$].

2. The other $\lceil(\frac{n}{2})\rceil$ products will be of the form:
   x = product of the left half of the elements previously computed AND the left half of the elements *not* previously computed.
   To illustrate: suppose we have n = 16, and we previously computed the products of elements 1-4 and 9-12. Then in the next run, we will compute the product of elements at indexes 1-2, 5-6, 9-10, 13-14. The table below shows a sample run of the algorithm for n=16. The X's mean that we're taking the product of those numbers.

#	1	2	3	4	0	6	7	8	9	10	11	12	13	14	15	16
1	X	X	X	X	X	X	X	X								
2	X	X	X	X					X	X	X	X				
3	X	X			X	X			X	X			X	X		
4	X		X		X		X		X		X		X		X	

   And the corresponding bit-vector:

0	1	0	0

   As seen from the table and the position of the zero element, $product_1 = 0, product_2 \neq 0, product_3 = 0, product_4 = 0$, thus we get the bit-vector shown above. In addition, 0100 is binary for 4 in decimal (+1 to get index in the array).

3. If the bitvector is not all 1's, then the 1+(binary representation of the bitvector) indicates the position of the zero element.
   Else we need to take the product of the right half of the array (ie from $\lceil(\frac{n}{n})\rceil$+1 to n) $\lceil(\log n)\rceil$ times, but store this product in some variable Z. If (Z = 0) then we know that the zero element is at position n, otherwise it is not in the array.

This algorithm will always run in at most 2 steps. In addition, the index of the zero element can be obtained from the bitvector. After one step, we have essentially figured out where the zero element is. In the worst case scenario when there is no zero element, the bitvector will contain all 1's, and we simply perform one more step on the upper half, which will indicate whether it is present or not.