CS2223 Algorithms D Term 2009
Exam 3 Solutions

By Prof. Carolina Ruiz
Dept. of Computer Science
WPI

May 4, 2009

PROBLEM 1: Asymptoptic Growth Rates (10 points)

Let A and B be two algorithms with runtimes $T_A(n) = \log_a n$ and $T_B(n) = \log_b n$, respectively, where a and b are two (possibly different) constants $a > 1, b > 1$.

Prove that $T_A(n) = \Theta(T_B(n))$.

Hint: use the fact that $\log_a n = \log_b n / \log_b a$

(DO NOT SPEND MORE THAN 5 MINUTES ON THIS PROBLEM.)

Solution:

Alternate Solution 1:

$T_A(n) = \log_a n = \log_b n / \log_b a = (1/\log_b a) * \log_b n = k * \log_b n$ where $k = 1/\log_b a$ is a constant. Note that $\log_b a > 0$ since $a, b > 1$. Hence $T_A(n) = \Theta(\log_b n) = \Theta(T_B(n))$

Alternate Solution 2:

$$T_A(n) = \log_a n,\ T_B(n) = \log_b n$$

$$\lim_{n\to+\infty} \frac{T_B(n)}{T_A(n)} = \lim_{n\to+\infty} \frac{\log_b n}{\log_a n} = \lim_{n\to+\infty} \frac{\log_b n}{\log_b n / \log_b a} = \lim_{n\to+\infty} \log_b a = \log_b a > 0 \text{ since } a, b > 1$$

Hence, $T_A(n) = \Theta(T_B(n))$

PROBLEM 2: Divide–and–Conquer (45 points + 5 bonus points)

Given a **sorted** array of **distinct** integers $A[1..n]$, your job is to find out whether there is an index i such that $A[i] = i$ (that is, whether there is a cell in the array whose content and whose index are the same). Here are a few examples:

If

A =	-7	-3	0	2	5	7	9	15	20	32	51	63
index:	1	2	3	4	5	6	7	8	9	10	11	12

then the answer is "$i = 5$".

If

A =	-17	-8	-1	0	7	13	82	236
index:	1	2	3	4	5	6	7	8

then the answer is "*no such i*"

If

A =	-17	2	3	4	9	13	82	236
index:	1	2	3	4	5	6	7	8

then any one of the following 3 answers would suffice: "i=2", "i=3", or "i=4".

Provide a divide–and–conquer algorithm to solve this problem in $O(\log n)$ time.

Hint: Consider an arbitrary index k, with $1 \leq k \leq n$. One of the following 3 cases holds:

- If $k = A[k]$, then k is the answer!
- If $k < A[k]$, then think of how $k+1$ compares to $A[k+1]$, $k+2$ compares to $A[k+2]$, ..., etc.
- If $k > A[k]$, then think of how $k-1$ compares to $A[k-1]$, $k-2$ compares to $A[k-2]$, ..., etc.

Construct your divide–and-conquer solution following the steps below.

1. **(20 Points)** Explain the design of your algorithm clearly (10 points) and prove carefully that the resulting algorithm is correct (10 points). Remember that your divide–and–conquer algorithm must run in $O(\log n)$ time. You can assume that the length of the array, n, is a power of 2.

   Solution:
   Following the hint provided above:

   - If $k < A[k]$, then $k + 1 < A[k] + 1$ and since the array is sorted in increasing order and doesn't contain repetitions, then $A[k] + 1 \leq A[k + 1]$. Hence, $k + 1 < A[k] + 1 \leq A[k + 1]$ and so $k + 1 < A[k + 1]$. The same argument holds for $k + 2, k + 3, \ldots, n$. In summary, for every k^+, $k \leq k^+ \leq n$, $k^+ < A[k^+]$.
     Hence, if $k < A[k]$ and there is some index i such that $A[i] = i$, then it must be the case that $i < k$.
   - If $k > A[k]$, then $k - 1 > A[k] - 1$ and since the array is sorted in increasing order and doesn't contain repetitions, then $A[k] - 1 \geq A[k - 1]$. Hence, $k - 1 > A[k] - 1 \geq A[k - 1]$ and so $k - 1 > A[k - 1]$. The same argument holds for $k - 2, k - 3, \ldots, 1$. In summary, for every k^-, $1 \leq k^- \leq k$, $k^- > A[k^-]$.
     Hence, if $k > A[k]$ and there is some index i such that $A[i] = i$, then it must be the case that $i > k$.

   So given an array $A[1..n]$ Our divide–and–conquer solution will select k as the midpoint of the array $k = n/2$. If $k = A[k]$ then we return k. If $k < A[k]$ then we recursive look for a solution in the first half of the array (i.e, $A[1..k - 1]$). If $k > A[k]$ then we recursive look for a solution in the second half of the array (i.e, $A[k + 1..n]$).

2. **(10 Points)** Write the algorithm in detailed pseudo-code.

   Solution:
   Here is our algorithm implementing the ideas above:

```
Binary-Search-for-fixpoint(A[1...n]) {
    Return Aux-Binary-Search-for-fixpoint(A, 1, n)
}
Aux-Binary-Search-for-fixpoint(A[u...v]) {
    If u > v
       Return "no such i"
    Else {
       Let k = floor((u + v)/2)
       If A[k] = k
          Return k
       Else If k < A[k]
          Return Aux-Binary-Search-for-fixpoint(A,u,k-1)
       Else // k > A[k] //
          Return Aux-Binary-Search-for-fixpoint(A,k+1,v)
    }
}
```

3. **(10 Points)** Write a recurrence for the runtime $T(n)$ of the algorithm. Explain your work.

   Solution:
   $T(n) = T(n/2)$.

4. **(10 Points)** Solve the recurrence to show that your algorith is $O(\log n)$. For this, either use the recursion-tree method (= "unrolling" the recurrence), the substitution method (= "guess + induction"), or the master theorem. Show your work and explain your answer.

   Solution:
   $T(n) = T(n/2)$. Since this recurrence has the form $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a = 1, b = 2$, and $d = 0$, and since $d = 0 = log_b a = log_2 1$, then the Master Theorem tells us that $T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$.

   For a solution using the substitution method (= "guess + induction"), see the Solved Exercise 1 in the course textbook (page 242).

PROBLEM 3: Dynamic Programming (45 points + 5 bonus points)

Suppose that your MP3 player has a disk capacity of M megabytes, where M is an *integer* (whole number). You have a list of n songs $S_1, S_2, S_3, \ldots, S_n$ that you would like to download onto your MP3 player. The i^{th} song S_i takes up m_i megabytes. The problem is that your MP3 player capacity, M, is less than the sum of all the m_i's. That is, $M < \sum_{i=1}^{n} m_i$.

Assume that you want to maximize the disk utilization of your MP3 player (that is, you want to use as many of your M megabytes as possible). We proved in Homework 4 that a greedy algorithm that selects songs to download in decreasing size order (from largest to smallest) will **NOT** produce an optimal solution.

Provide an optimal dynamic–programming solution to this problem following the steps below. *Note that the output of the algorithm should merely be the maximum disk utilization (a single numerical value, in megabytes). The subset of songs for which this maximum is attained, does not need to be returned.* Here is the input–output specification:

Maximum disk utilization$(m[1...n], M)$

Inputs: Array $m[1...n]$ of song sizes, and positive integer disk capacity M.

Output: Maximum disk utilization $\sum_{i \in S} m_i$ that can be attained over a subset $S \subseteq \{1..n\}$ of songs without exceeding the disk capacity M.

1. **(25 Points) Dynamic Programming Solution**.

   (a) **(20 points)** Design a dynamic programming solution to the disk utilization problem, using the subproblems suggested by the hint that follows.

   > **Hint:** Consider whether or not to add the i-th song to a disk of capacity j, assuming that the first $i-1$ songs and smaller disk capacities $j' < j$ have already been considered. To this end, define a 2-dimensional $(n+1) \times (M+1)$ matrix (array) $OPT[0..n][0..M]$, where $OPT[i,j] =$ maximum disk utilization over a subset of the first i songs $S_1..S_i$, assuming a maximum disk capacity of j megabytes.

   Find a recurrence relation that expresses the solution of $OPT[i,j]$ in terms of the solutions of slightly "smaller" problems (i.e., other cells in the matrix OPT whose values can be calculated before). Include suitable boundary conditions for the recurrence relation. At each stage argue decisively that your solution is correct.

   Solution:

   Define OPT as in the hint. This corresponds to considering, for each i between 0 and n, and each j between 0 and M, the subproblem consisting of finding the maximum disk utilization that is possible by selecting songs from among the first i only, assuming disk capacity j. In order to derive a recurrence relation for OPT, consider whether to add the i-th song S_i to a disk of capacity j, assuming that songs $S_1..S_{i-1}$ have already been considered. If song S_i doesn't fit on the disk at all, that is, if $m_i > j$, and assuming that only the first i songs are allowed (we're targeting $OPT[i,j]$ here), then of course the best that you can do is whatever the first $i-1$ songs allow:

   $$OPT[i,j] = OPT[i-1,j] \text{ if } m_i > j$$

   Otherwise, if song S_i fits, then you have two options: you can either add S_i to the disk, or not, and you will need to pick whichever of these two options provides a higher disk utilization. Note that if you *do* add S_i to the disk, then the remaining

disk space $j - m_i$ should be optimally utilized, and this must be done by picking from among the remaining songs, that is, only songs from among the first $i - 1$:

$$OPT[i, j] = \max\{OPT[i-1, j],\ m_i + OPT[i-1, j-m_i]\} \text{ if } m_i \leq j$$

Finally, if $i = 0$ (base case), there are no songs to consider at all, and so the maximum disk utilization possible is 0. Putting all of the above considerations together, we arrive at the following recurrence relation (base case included):

$$OPT[i, j] = \begin{cases} 0, \text{ if } i = 0 \\ OPT[i-1, j], \text{ if } m_i > j \\ \max\{OPT[i-1, j],\ m_i + OPT[i-1, j-m_i]\}, \text{ otherwise} \end{cases}$$

(b) **(5 points)** Based on the preceding part of this question, determine in what order you'll calculate the values of the cells in OPT. Explain.

Solution:
In light of the recurrence relation, all values $OPT[i-1, j']$ with $0 \leq j' \leq j$ must have been calculated prior to attempting to calculate $OPT[i, j]$. Therefore, we first calculate the values $OPT[0, j']$, for $0 \leq j' \leq M$, then the values $OPT[1, j']$ in order of increasing j', then the values $OPT[2, j']$, and so on.

2. **(12 Points) Algorithm**. Write a dynamic programming algorithm (in detailed pseudo-code) implementing the solution you designed above.

Solution:

Maximum disk utilization$(m[1...n], M)$
Inputs: Array $m[1...n]$ of song sizes, and positive integer disk capacity M.
Output: Maximum disk utilization $\sum_{i \in S} m_i$ that can be attained over a subset $S \subseteq \{1..n\}$ of songs without exceeding the disk capacity M.

```
{
    For j := 0 to M do
        OPT[0, j] = 0
    For i := 1 to n do
        For j := 1 to M {
            If m_i > j Then
                OPT[i, j] = OPT[i − 1, j]
            Else
                OPT[i, j] = max{OPT[i − 1, j], m_i + OPT[i − 1, j − m_i]}
        }
    return(OPT[n, M])
}
```

3. **(13 Points) Time Complexity**. Determine the asymptotic running time of your dynamic programming algorithm above, as a function of the input sizes n and M. Give the simplest possible big Theta expression for the running time.

Solution:
The initialization loop makes $M + 1$ passes, each of which involves a single assignment. Therefore, the total initialization time is $\Theta(M)$. A total of nM passes are made through the nested loops in which the values of the OPT matrix are updated. Each pass just

involves an if-else, an assignment, and possibly an addition and max evaluation. In either case, the total time per pass is bounded above and below by nonzero, finite constants (that do not depend on either n or M). It follows that the total time needed for the neted update loops is $\Theta(nM)$. Adding the time for the return statement at the end, we obtain a total running time $\Theta(nM)$.