CS350 - Algorithms: Homework #4

Due on February 26, 2015

for Professor Andrew Black

Kristina Frye

Problem 1

Proof of Master Theorem. Complete the proof, started in class, for the two cases of the Master Theorem where $a = b^d$ and $a > b^d$

Master Theorem: Let $T(n) = aT(\frac{n}{b}) + f(n)$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{log_b a}) & \text{if } a > b^d \end{cases}$$

Proof:
Solve for $T(n)$:

$$\begin{aligned} & T\left(\frac{n}{b}\right) = a \cdot T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right) \\ \implies & T(n) = a\left(a \cdot T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n) \\ \implies & T(n) = a^2 \cdot T\left(\frac{n}{b^2}\right) + a \cdot f\left(\frac{n}{b}\right) + f(n) \\ \implies & T(n) = a^k \cdot T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right) \end{aligned}$$

If $n = b^k$, then:

$$T(n) = a^{\log_b n} \cdot T(1) + \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

We know that $f(n) \in \Theta(n^d)$ and $a^{\log_b n} = n^{\log_b a}$, so:

$$T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n} a^i c\left(\frac{n}{b^i}\right)^d$$

for some constant c. Therefore:

$$T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n} c \cdot n^d \left(\frac{a}{b^d}\right)^i = n^{\log_b a} + c \cdot n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i$$

Case 1 ($a < b^d$) was proved in class. **Case 2** ($a = b^d$)**:**
If $a = b^d$, then $\log_b a = \log_b b^d = d \log_b b = d$. Therefore $n^{\log_b a} = n^d$. So:

$$T(n) = n^d + c \cdot n^d \sum_{i=0}^{\log_b n} 1 = \Theta(n^d \log_b n)$$

Case 3 ($a > b^d$)**:**

$$T(n) = n^{\log_b a} + c \cdot n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i$$

Since $a > b^d$, the last term of the summation will be the largest. By the theorem in the book on page 56,

Problem 1 continued on next page...

the efficiency class is equal to the largest efficiency class in the summation. Therefore:

$$\begin{aligned}
T(n) &\in max\left\{\Theta\left(n^{\log_b a}\right), \Theta\left(n^d\left(\frac{a}{b^d}\right)^{\log_b n}\right)\right\} \\
&\in max\left\{\Theta\left(n^{\log_b a}\right), \Theta\left(n^d\left(\frac{a^{\log_b n}}{b^{d\log_b n}}\right)\right)\right\} \\
&\in max\left\{\Theta\left(n^{\log_b a}\right), \Theta\left(n^d\left(\frac{n^{\log_b a}}{n^{\log_b b^d}}\right)\right)\right\} \\
&\in max\left\{\Theta\left(n^{\log_b a}\right), \Theta\left(n^d\left(\frac{n^{\log_b a}}{n^d}\right)\right)\right\} \\
&\in max\left\{\Theta\left(n^{\log_b a}\right), \Theta\left(n^{\log_b a}\right)\right\} \\
&\in \Theta\left(n^{\log_b a}\right)
\end{aligned}$$

Problem 2

Application of Master Theorem. Use the Master Theorem to find the order of growth for the solutions of the following recurrences:

(a) $T(n) = 4T\left(\frac{n}{2}\right) + n, T(1) = 1 \implies a = 4, b = 2, d = 1 \implies b^d = 2 \implies a > b^d \implies$
$T(n) \in \Theta(n^{\log_2 4}) \implies T(n) \in \Theta(n^2)$

(b) $T(n) = 4T\left(\frac{n}{2}\right) + n^2, T(1) = 1 \implies a = 4, b = 2, d = 2 \implies b^d = 4 \implies a = b^d \implies$
$T(n) \in \Theta(n^2 \log n)$

(c) $T(n) = 4T\left(\frac{n}{2}\right) + n^3, T(1) = 1 \implies a = 4, b = 2, d = 3 \implies b^d = 8 \implies a < b^d \implies$
$T(n) \in \Theta(n^3)$

Problem 3

Quicksort.

(a) Apply Quicksort, by hand, to sort the letters of the word ALGORITHM into alphabetical order. Draw the tree of the recursive calls that the algorithm makes, marking on the tree the values of l (the index of the left end of the sub-array), r (the index of the right end) and s (the index of the pivot after the partition) for each of the calls. For the purpose of this exercise only, assume that the first element of the sub-array, $A[l]$, is the pivot.

Problem 3 continued on next page...

$l = 0, r = 8, s = 0$ ALGORITHM

$l = 0, r = 0, s = 0$

Partition: IGH
Input: $l = 1, r = 3, p = I$
$i = 1, j = 4$
$i = 2..3, A[3] = H < I$
$j = 3, A[3] = H < I$
$swap(A[3], A[3])$
$3 \geq 3$
$swap(A[3], A[3])$
$swap(A[1], A[3])$
HGI
return 4

Partition: HG
Input: $l = 1, r = 2, p = H$
$i = 1, j = 3$
$i = 2, A[2] = G < H$
$j = 2, A[2] = G < H$
$swap(A[2], A[2])$
$2 \geq 2$
$swap(A[2], A[2])$
$swap(A[1], A[2])$
HG
return 2

Partition: RTOM
Input: $l = 5, r = 8, p = R$
$i = 5, j = 9$
$i = 6, A[6] = T > R$
$j = 8, A[8] = M < R$
$swap(A[6], A[8])$
RMOT
$i = 8, A[8] = T > R$
$j = 7, A[7] = O < R$
$swap(A[8], A[7])$
RMTO
$8 > 7$
$swap(A[8], A[7])$
RMOT
$swap(A[5], A[7])$
OMRT
return 7

Partition: OM
Input: $l = 5, r = 6, p = O$
$i = 5, j = 7$
$i = 6, A[6] = M < O$
$j = 6, A[6] = M < O$
$swap(A[6], A[6])$
$6 \geq 6$

$swap(A[6], A[6])$
$swap(A[5], A[6])$
MO
return 6

(b) Should an implementation of Quicksort use the first element of a sub-array for the pivot? Explain why, or why not.

Answer: The pivot should normally be a random element in the array. If each of the items in the array in random order, then it's okay to choose the first element. However, in many cases, the array will already have some sort of sorting. In that case, a random element of the array should be chosen instead of the first element, which has a higher probability of being a lower element of the sorted array.

Problem 4

Trees. Prove that Vertex u is an ancestor of vertex v in a rooted ordered tree T if and only if $preorder(u) \leq preorder(v) \wedge postorder(u) \geq postorder(v)$, where $preorder(x)$ and $postorder(x)$ are the numbers assigned to vertex x by the preorder and postorder traversals of T

Proof:
Left $\implies$ Right
Suppose Vertex u is an ancestor of vertex v in a rooted ordered tree T. Then, according to the definition of preorder traversal (Section 5.3 in the book), the root is visited before the left and right subtrees are visited (in that order). Therefore, the ancestor will be visited before the descendants, which implies that $preorder(u) \leq preorder(v)$. During postorder traversal, the root is visited after visiting the left and right subtrees (in that order). Therefore, the ancestor will be visited after the descendants, which implies that $postorder(u) \geq preorder(v)$

Right $\implies$ Left
Suppose $preorder(u) \leq preorder(v) \wedge postorder(u) \geq postorder(v)$ and suppose u is not an ancestor of v. If $preorder(u) \leq preorder(v)$, then u is visited before v in the preorder traversal. This occurs in two cases: u is an ancestor of v or u is in a left branch of the tree and v is in a right branch since the left branch is visited before the right branch in a preorder traversal. Since we are assuming u is not an ancestor of v, then u must be in the left branch and v must be in the right branch.

Now consider $postorder(u) \geq postorder(v)$. This occurs in two cases: u is an ancestor of v, or u is in the right branch of the tree and v is in left branch since, similar to the preorder traversal, the left branch is visited before the right branch in a postorder travel. Since we are assuming u is not an ancestor of v, then u must be in the right branch and v is in the left branch. But this contradicts with the earlier finding that u must be in the left branch and v must be in the right branch if u is not a ancestor of v. Therefore our assumption must be wrong and u must be an ancestor of v.

Problem 5

Horspool's Algorithm. Consider the problem of searching for genes in DNA sequences using Horspool?s algorithm. A DNA sequence is represented by a text on the alphabet A, C, G, T; the gene or gene segment is the pattern.

Problem 5 continued on next page...

(a) Construct the shift table for the following gene segment of your chromosome 10: TCCTATTCTT

A	C	G	T
5	2	10	1

(b) Apply Horspool's algorithm by hand to locate the above pattern in the following DNA sequence: TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
Your answer should draw each alignment of the pattern and the text, and indicate which characters are compared in that alignment.

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 1)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 2)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 1)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 5)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 1)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 1)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 1)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 5)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 1)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 2)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 2)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 1)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(shift by 5)*

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
TCCTATTCTT *(match!)*

Problem 6

Open Hashing for the input $30, 20, 56, 75, 31, 19$ and hash function $h(K) = K \bmod 11$

(a) construct the open hash table (draw it).

keys	30	20	56	75	31	19
hash addresses	8	9	1	9	9	8

0	1	2	3	4	5	6	7	8	9	10
	↓ 56							↓ 30 ↓ 19	↓ 20 ↓ 75 ↓ 31	- -

(b) find the largest number of key comparisons in a successful search in this table.
Answer: 3

(c) find the average number of key comparisons in a successful search in this table.
Answer: $(1 + 1 + 2 + 1 + 2 + 3)/6 = 1.67$

Problem 7

Closed hashing for the input $30, 20, 56, 75, 31, 19$ and hash function $h(K) = K \bmod 11$

1. construct the closed hash table (draw it).

keys	30	20	56	75	31	19
hash addresses	8	9	1	9	9	8

0	1	2	3	4	5	6	7	8	9	10
31	56	19						30	20	75

2. find the largest number of key comparisons in a successful search in this table.
Answer: 3

3. find the average number of key comparisons in a successful search in this table.
Answer: $(1 + 1 + 1 + 2 + 3 + 6)/6 = 2.33$

Problem 8

Unique Elements

(a) Design an algorithm that uses hashing to check whether all elements of a list are distinct.

```
procedure AreElementsUnique?(A[0..n − 1])                ▷ An array of elements
    for i = 1 to n − 1 do
        index ← GetHashIndex(A[i])                       ▷ Get the hash index for the current object
        t ← GetHash(index)                               ▷ Get the pointer to the hash table at index val
        while t ≠ null do
            if t.val = A[i] then                         ▷ If the pointer isn't null, check for equality
                return false                             ▷ This item already exists!
            end if
            t = t.next                                   ▷ Get the next item in the linked list
        end while
        PutHash(val, t)                                  ▷ Put the value t in the hash table at index val
    end for
    return true                                          ▷ No duplicates have been found
end procedure
```

(b) What is the asymptotic efficiency of your algorithm?

Answer: If you have an extremely bad hash table that maps everything to the same hash index, then you will need to check each value against every other value that has already been placed into the table. This is going to be $O(n^2)$ efficiency. However, if you have a reasonable hash table that only requires a constant (average) number of lookups for each value, the efficiency class is $O(n)$.

(c) How does it compare with the algorithm in which we first presort the list, and then compare adjacent elements?

Answer: If the sorting algorithm used is $O(n \log n)$, then the total efficiency for that algorithm is $O(n) + O(n \log n) = O(n \log n)$. This means our hashing algorithm is better (as long as we have a good hash function).