Last Name: *Chen* **First Name:** *Robert* **Email:** *rchen87@gatech.edu*

Problem 1, Analysis of Algorithm (10 points)
Where n is a power or 2 and $n > 8$, how many x's does the function Mystery(n) below print?
(a) Write and solve the recurrence exactly using substitution.
(b) State the solution in $O()$ notation.

```
Mystery(n)
if n > 8 then begin
        print("x")
        Mystery(n/2)
        Mystery(n/2)
        end
```

Answer:

a. The recurrence: $T(n) = 2T(n/2) + c$

The guess: $T(n) \leq an + b$

The inductive step is this:
$T(n) \leq 2T(n/2) + c$

To make substitution:
$T(n) \leq 2[a(n/2) + b] + c$
$= an + 2b + c$
to satisfy an upper bound of $an + b$, then the following need to be met: $c \leq a + b$ and $2b + c = c$, which is definitely possible e.g., $a = 2c$ and $b = -c$ which satisfies $T(n) \leq 2cn - c = an + b = O(n)$

b. $O(n)$.

Problem 2, Analysis of Algorithm (10 points)
Where n is a power or 2 and $n \geq 1$, how many x's does the function Mystery(n) below print?
(a) Write and solve the recurrence exactly using substitution.
(b) State the solution in $O()$ notation.

```
Mystery(n)
print("xx")
if n > 1 then begin
        Mystery(n/2)
        Mystery(n/2)
        Mystery(n/2)
        end
```

Answer:
a. Recurrence is:

$$T(n) = 3T(n/2) + 2$$

generally speaking it is:

$$T(n) = 3T(n/2) + c$$
$$= 3(3T(\frac{n}{2^2}) + c) + c$$
$$= 3^1(3T(\frac{n}{2^2}) + 3^1c + c$$
$$= 3^2(3T(\frac{n}{2^3} + c) + 3^1c + c$$
$$= 3^2(3T(\frac{n}{2^3} + 3^2c + 3^1c + c$$
$$...$$

...and so on and so forth...
So the guess will be:
$T(n) = 3^kT(\frac{n}{2^k}) + 3^{k-1}c$, for positive values of k.
We can verify this by induction:

$$T(n) = 3T(n/2) + c$$
$$T(n) = 3^{k-1}T(\frac{n}{2^{k-1}}) + 3^{k-2}c$$
$$= 3^{k-1}(3T(\frac{n}{2^k}) + 3^{k-1}c) + 3^{k-2}c$$

b. $O(n^{log_2(3)})$. If you express this recurrence as a tree there are $3^{log_2 n}$ leaves because the tree height is $log_2 n$. Therefore there will be $3^{log_2 n} - 1$ internal nodes and therefore we can arrive at $T(n) = 2(3^{log_2 n}) - 1 = O(3^{log_2 n}) = O(n^{log_2 3})$.

Problem 3, Analysis of Algorithm (10 points)
Where n is a power or 2 and $n \geq 1$, how many x's does the function Mystery(n) below print?
(a) Write and solve the recurrence exactly using substitution.
(b) State the solution in $O()$ notation.

```
Mystery(n)
for i := 1 to n^2
        print("xx")
if n > 1 then begin
                Mystery(n/2)
                Mystery(n/2)
                Mystery(n/2)
                end
```

Answer:

a. recurrence is: $T(n) = 3T(n/2) + n^2$.
initial guess we have is $O(n^2)$.
Therefore we assume that there is a c such that $T(n) \leq cn^2$, and we plug in $c(n/2)^2$ for $T(n/2)$ in the above recurrence.

$T(n) = 3T(n/2) + n^2$
$\leq 3[c(n/2)^2] + n^2$
$= \frac{3}{4}cn^2 + n^2$
$= n^2(\frac{3}{4}c + 1)$
$\leq cn^2$
if $c \leq \frac{4}{3}$

b. $O(n^2)$. We showed this in part a. You can also use master theorem: $2 > log_2(3)$ therefore it is $O(n^2)$.

Problem 4, Analysis of Algorithm (10 points)
Where n is a positive integer, how many x's does the function Mystery(n) below print?
(a) Write and solve the recurrence exactly using substitution.
(b) State the solution in $O()$ notation.

```
Mystery(n)
if n = 1 then print("x")
if n > 1 then begin
                Mystery(n − 1)
                Mystery(n − 1)
                end
```

Answer:

a. The recurrence is $T(n) = 2T(n-1)$.
Therefore:
$T(n-1) = 2T(n-2)$
$T(n) = 2(2T(n-2))$
$= 4T(n-2)$
Therefore:
$T(n) = 2^k T(n-k)$
NOw, we let $n - k = 1$
therefore:
$T(n) = 2^k T(1)$
$\Rightarrow T(n) = O(2^k)$
Running time is $O(2^n)$

b. $O(2^n)$.

Problem 5, Min and Max with Fewer Comparisons (10 points)
Let $a_1 \dots a_n$ be an input array of n unsorted distint integers, where n is an even number. Let $m_{\max}$ be the maximum value of the above integers, and let $m_{\min}$ be the minimum value of the above integers. Give an algorithm that findsboth $m_{\max}$ and $m_{\min}$ using at most $\frac{3n}{2} - 2$ comparisons.
Argue correctness and running time of your solution.

Answer:

```
def find_min_max(a):
    ## initialize new arrays
    l_smallers = []
    l_biggers = []


    ## we are going to make n/2 comparisons (every set of 2 consecutive elements,
    ## put the smaller of each pair in l_smallers, put the bigger of
    ## each pair in l_biggers)

    i=2
    while (i <= n):
        if a[i] > a[i-1]:
            l_smallers.append(a[i-1])
            l_biggers.append(a[i])
        else:
            l_smallers.append(a[i])
            l_biggers.append(a[i-1])

    ## now, find the min, by simply finding the smallest in l_smallers
    ## this will require n/2-1 comparisons
    curr_smallest = infinity
    for a_k in l_smallers:
        if a_k < curr_smallest:
            curr_smallest = a_k


    ## now, find the max, by simply finding the biggest in l_biggers
    ## this will require n/2-1 comparisons
    curr_biggest = -infinity
    for a_k in l_biggers:
        if a_k > curr_biggest:
            curr_biggest = a_k

    return {'max': curr_biggest, 'min': curr_smallest}
```

To argue correctness: you need n/2 comparisons to create l_smallers and l_biggers. Then, within each of l_smallers and l_biggers you need n/2-1 comparisons (of that element in the array vs. the running biggest/smallest). Total you have:

$$n/2 + n/2 - 1 + n/2 - 1 = \frac{3n}{2} - 2$$

The runninng time is:
$O(n/2 + (n/2 - 1) + (n/2 - 1)) = O(n)$

Problem 6, Sorting Faster than $O(n \log n)$ in Special Case (10 points)
Let $a_1 \ldots a_n$ be an input array of n unsorted and not necessarily distint integers. Let $m_{\max}$ be the maximum value of the above integers, and let $m_{\min}$ be the minimum value of the above integers. Let $M = m_{\max} - m_{\min}$. Give an $O(n + M)$ comparison algorithm that sorts the input array.
Argue correctness and running time of your solution.

Answer:

```
def sort(input_arr):
    count_arr: initialize array w/ size m_max #although we only need indexes m_min to m_max

    for a = 1 .. n       # initialize a OUTPUT array
        output_arr[a] = NIL
    for i = m_min .. m_max       #  initialize a "counting" array with M elements
        count_arr[i] = 0
    for j = 1 .. n              # loop thru input_arr and increment the counting array
                                #  this loop is O(n)
        count_arr[input_arr[j]] = count_arr[input_arr[j]] + 1 #increment by 1
    for i = m_min+1 .. m_max             #
        count_arr[i] = count_arr[i] + count_arr[i-1] # record the number of elts
                                                     # in input_arr that are <=i
                                                     # this is O(M)
    for j = n .. 1 #count decrementing by 1
        output_arr[count_arr[input_arr[j]]] = input_arr[j]
        count_arr[input_arr[j]] = count_arr[input_arr[j]] - 1

    return output_arr
```

Argue: this is sometimes known as "counting sort". It's going to run in O(n+M). It is O(n) for the looping through the input array and assigning the corresponding element in the counting array. It is O(M) for the part where you loop through the counting array [just the M elements of the counting array that correspond to the range of m_{min} thru m_{max}], and accordingly updating each element of the counting array with the number of elements that were less than the current one.

Problem 7, Use of Pointers (10 points)
Let $a_1 \dots a_n$ be n sorted distint integers, and let τ be an additional given integer. Give an $O(n)$ comparison algorithm that decides if there exist distinct indices i and j, ie $1 \leq i < j \leq n$, such that $a_i + a_j = \tau$.
Argue correctness and running time of your solution.

Answer:

```
def determine_if_sum_T_in_arr(sorted_arr, T):
    exists = False;
    i = 1
    j = n

    while (i < j): # while the condition holds that i<j
        if sorted_arr[i] + sorted_arr[j] > T: # if its too big
            j = j - 1
        elif sorted_arr[i] + sorted_arr[j] < T: # if its too small
            i = i + 1
        else:
            exists = True

    return exists
```

To argue correctness: The algo is correct. Eventually if the pointers are pointing to two elts that add to T then the "if" and the "elif" statements are false and the "else" statement exectued returning true. Because the "if" and the "elif" statements are measuring sum greater than T and less than T respectively. If sum is not graeter than and not less than then the sum is equal to.

Running time is O(n): we move the pointers not more than n times. at each iter of the while loop you either move the left pointer or the right pointer. If the left pointer has to be moved to collide with the right pointer than the while loop would be terminated and algo returns False.

Problem 8, Searching a Tree (10 points)

You are given a complete binary tree on n nodes, where each node has a distinct value w_i, $1 \leq i \leq n$. The input representation is as follows:

(1) Index 1 is the root of the tree.

(2) For $1 \leq i \leq \frac{n-1}{2}$, the left child of i is $2i$ and the right child of i is $2i+1$.

(3) For $2 \leq i \leq n$, the parent of i is $\lfloor \frac{n}{2} \rfloor$.

Say that k is a *local minimum* if and only if:

(1) If $k = 1$, then w_1 is smaller than both its children.

(2) If $k \geq \frac{n-1}{2}$, then w_k is smaller than its parent.

(3) If $2 \leq k \leq \frac{n-1}{2}$, then w_k is smaller than both its children, and w_k is also smaller than its parent.

Give an $O(\log n)$ comparison algorithm that finds a local minimum of the binary tree.
Justify correctness and running time.

Answer:

Problem 9, Sorting in Linear Time in Special Case (10 points)
Suppose that n is a perfect square and let $N = n + \sqrt{n}$.
You are given an array of $a_1 \dots a_N$ distinct integers, where the first n integers are sorted $a_1 < a_2 < \dots < a_n$, but the last $\sqrt{n}$ integers are not sorted. Give an $O(N)$ comparison algorithm that sorts the entire input array $a_1 \dots a_N$.

Answer:

Problem 10, Comparing Algorithmic Performance (10 points)
Suppose you are choosing between the following three algorithms:
• Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining solutions in linear time.
• Algorithm B solves problems of size n by recursively solving two subproblems of size $(n-1)$ and then combining the solutions in constant time.
• Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.
Solve each recurrence by substitution, and give the running times of each of these algorithms in $O()$ notation. Which one would you choose as the asymptotically fastest?

Answer:

Algo A:
The recurrence: $T(n) = 5T(n/2) + n$
Becomes

$$T(n) = 5^T(n/2^k) + cn((5/2)^{k-1} + (5/2)^{k-2} + ... + (5/2) + 1)$$

...applying the use of power series for $x > 1$, we get that:

$$= 5^k T(n/2^k) + cn(\frac{(5/2)^k - 1}{(5/2) - 1})$$

at this point we know base case T(1), so we want $\frac{n}{b^k} = 1 \iff k = log_2 n$.
Therefore the recurrence becomes:

$$= 5^{log_2 n} T(1) + \frac{cn2}{5-2}(\frac{5^{log_2 n}}{b^{log_2 n}} - 1)$$

$$= 2^{log_2 5 log_2 n} + \frac{cn2}{5-2}\frac{2^{log_2 5 log_2 n}}{2^{log_2 n}} - \frac{cn2}{5-2}$$

$$= 2^{log_2 n log_2 5}(1 + \frac{c2}{5-2}) - \frac{cn2}{5-2}$$

$$= n^{log_2 5}(1 + \frac{c2}{5-2}) - \frac{cn2}{5-2}$$

$$= O(n^{log 25}) = O(n^2.3)$$

Algo B: this is a classic tower of hanoi

$$T(m) = 2T(m-1) + c$$

making subsitution we get:

$$T(m) = 2(2T(m-2) + c) = 2^2 T(m-2) + 2^1 c + 2^0 c$$

making the next substition we get:

$$T(m) = 2^2(2T(m-2) + c) + 2^1 c + 2^0 c = 2^3 T(m-3) + 2^2 c + 2^1 c + 2^0 c$$

We can look at the pattern and **guess** that if we substitute k times we will have:

$$T(m) = 2^{k+1} T(m - (k+1)) + 2^k c + ...2^0 c$$

If we substitute $k = m - 2$ times we have an idea of a base case scenario:

$$T(m) = 2^{m-1} T(1) + ... + 2^0 c$$

thus $T(m) = \sum_{i=0}^{m-1} 2^i c = \frac{1-2^m}{1-2} c = c(2^m - 1)$. This is exponential: $O(2^m)$

Algo C:

We make guess of $n^2 logn$.

$$T(n) = 9T(n/3) + n^2$$

then

$$\begin{aligned} T(n) &\leq 9[c(n/3)^2 log(n/3) + (n/3)^2)] + n^2 \\ &= cn^2 log(n/3) + n^2 + n^2 \\ &= c(n^2 logn - n^2 log3) + n^2 \\ &\leq cn^2 logn \end{aligned}$$

for certain $c > 0$ small enough and n large enough. $O(n^2 logn)$.

Algo C is the one to choose. Algo B is just bad. You don't even need to do anything and you know its exponential and thus worse than A and B. Between A and C, A is better. Algo C is $O(n^2 logn)$. Algo A is $O(n^{log_2 5}) = O(n^{2.3})$. $O(n^2 log_9 n)$ grows slower asymptotically than $O(n^{2.3})$.