CS375-A0 Assignment 1 (Answer)

The assignment is designed to enhance the concept of time complexity, instruction counts, mathematics computation, and to implement an efficient program to solve a simple problem.

There are two parts in this assignment: (A) Theory part and (B) programming part

[Part A] Theory [85%]:

1. [6%] What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

   Answer: n >=15

2. [24%] What is the count for the instruction CountMe as a function of n for the fragments below?

   a. [8%]

```
Line 1: for (i=0; i<n^3; i++ ){
Line 2:     for (j=0; j<n^4/5; j++){
Line 3:         CountMe
            }
        }
```

   CountMe is executed $n^7/5$ times OR (n^3) * Floor[n^4 / 5]

   b. [8%]

```
Line 1: j = n
Line 2: while (j > 0) {
Line 3:     CountMe
Line 4:     j = j-2;
```

```
        }
```

The loop is executed for i= n, n - 2, n - 4,…., 1(if n was odd or 2 if n was even)

It is executed n/2 times for even n, and (n+1)/2 for odd n.

c. [8%] For simplicity you may assume that $n=2^k$ for some positive integer k.

```
Line 1: j = 1

Line 2: while (j<n) {

Line 3:    CountMe

Line 4:    j = 2*j;
        }
```

Countme is executed for j = 1, 2, 4, …, 2^{k-1}.

So k times (i.e., 2^0, 2^1,…, 2^{k-1}) k= ($\log_2$ n) times.

3. [12%] Prefix averages

Given an array X storing n numbers, we want to compute an array A such that A[i] is the average of the elements X[0], …, X[i] for i=0, …, n-1, or

$$A[i] = \frac{1}{i+1}\sum_{j=0}^{i} X[j].$$

For example if X = [1, 5, 10, 100], A[0]=1, A[1] = (1+5) / 2= 3, A[2] = (1+5+10) / 3= 5.333, and A[3] = (1+5+10+100) / 4 = 116/4 =29.

Computing prefix averages has many applications in economics and statistics. For example given year-by-year returns of a mutual fund, an investor will typically want to see the fund's average annual returns for the last year, the last three years, the last 5 years, and the last 10 years.

(1) [8%] Design your algorithm to calculate the prefix average values A[i]. Write the pseudo-code or C-code to demonstrate the procedure. Analyze the every-case time complexity by counting the barometer operation (e.g., "+" operation).

(2) [4%] If your algorithm is in $\Theta(n^2)$, is there any way to improve the pseudo-code to make the algorithm in $\Theta(n)$ time complexity? If so, show your code and count the number of execution of barometer operation "+".

Answer:

a. [8%]

```
Line 1: for i = 0 to n - 1 do
Line 2:    a = 0
Line 3:    for j = 0 to i do
Line 4:          a = a + X[j]
Line 5:    A[i] = a/(i+1)
Line 6: return array A
```

We choose the + of line 4 as a barometer operation. It is executed:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i}1=\sum_{i=0}^{n-1}(i+1)=\sum_{q=1}^{n}q=n(n+1)/2\,.$$

b. [4%]

```
Line 1: s=0
Line 2: for i=0 to n-1 do
Line 3:     s= s + X[i]
Line 4:     A[i] = s /(i+1)
Line 5: return array A
```

We choose the + of line 3 as a barometer operation. It is executed n times.

NOTE: if students report the O(n) algorithm directly (without O(n^2) algorithm), the part (a) can be skipped. It is Ok to report (b) only.

4. [8%] Compute the following sums:

a. $\sum_{i=1}^{n}(\frac{i}{3}+3)$ [4%]

$$\sum_{i=1}^{n}(i/3+3)=(\sum_{i=1}^{n}i)/3+3n=\frac{n(n+1)}{6}+3n$$

b. 2/3 + 2/9 + 2/27+ 2/81 +... What is the sum when *n* goes to infinity? [4%]

$$2/3(\sum_{i=0}^{\infty}1/3^{i})=\frac{2}{3(1-1/3)}=1$$

5. [8%] Show by using limits that:

a. $3^{n+1}\in\Theta\left(3^{n}\right)$ [5%]

b. $n^3 - 2n = \omega(n)$ [5%]

a) $3^{n+1} \in \Theta(3^n)$

$\lim_{n\to\infty} \frac{3^{n+1}}{3^n} = 3$. Therefore $3^{n+1} \in \Theta(3^n)$

b) $n^3 - 2n = \omega(n)$

$\lim_{n\to\infty} \frac{n^3 - 2n}{n} = \infty$. Therefore $n^3 - 2n = \omega(n)$

6. [6%] Use the original definitions to show that $n^2 - 10n + 1 = \Theta(n^2)$

a. We will first show that $n^2 - 10n + 1 = O(n^2)$

We need to find a $c > 0$ and $N > 0$ such that $0 \le n^2 - 10n + 1 \le cn^2$ for all $n \ge N$.

Dividing the inequality by $n^2 > 0$ we get, $0 \le 1 - \frac{10}{n} + \frac{1}{n^2} \le c$. Clearly, for $n \ge 10$, $0 \le 1 - \frac{10}{n} + \frac{1}{n^2}$. $0 \le 1 - \frac{10}{n} + \frac{1}{n^2} \le 2$ for all $n \ge 10$. Choosing $N = 10$ and $c = 2$, we have $0 \le n^2 - 10n + 1 \le 2n^2$ for all $n \ge 10$. So $n^2 - 10n + 1 = O(n^2)$.

b. We will now show that $n^2 - 10n + 1 = \Omega(n^2)$

We need to find a $c > 0$ and $N > 0$ such that $0 \le cn^2 \le n^2 - 10n + 1$ for all $n \ge N$.

Dividing the inequality by $n^2 > 0$ we get, $c \le 1 - \frac{10}{n} + \frac{1}{n^2}$ and $c > 0$. Clearly, for $n \ge 20$, $\frac{1}{2} \le 1 - \frac{10}{n} + \frac{1}{n^2}$.. Choosing $N = 20$ and $c = 1/2$, we have $0 \le \frac{1}{2} n^2 \le n^2 - 10n + 1$ for all $n \ge 20$. So $n^2 - 10n + 1 = \Omega(n^2)$.

7. [6%] Using the definitions of O and Ω , show that
$6n^2 + 20n + 1 \in O(n^3)$ but $6n^2 + 20n + 1 \notin \Omega(n^3)$

Answer: $6n^2 + 20n + 1 \le cn^3$, $6\frac{1}{n} + 20\frac{1}{n^2} + \frac{1}{n^3} \le c$, N=2, C=8.125, so
$6n^2 + 20n \in O(n^3)$
$6n^2 + 20n + 1 \ge cn^3$, $6\frac{1}{n} + 20\frac{1}{n^2} + \frac{1}{n^3} \ge c$, cannot find a positive value of c which satisfies the inequality when n goes to sufficient large. In other words, there always exists a large N which makes the left side of the inequality smaller than c. So $6n^2 + 20n + 1 \notin \Omega(n^3)$

8. [8%] Given a list of distinct positive numbers and the average or mean of those numbers, following pseudo-code is to determine whether there are more numbers above the average than below.

```
MoreAbove( list, average, N )
   countAbove = 0
   for j = 1 to N do
         if list[ j ] > average then
             countAbove = countAbove + 1
             if countAbove > N/2 then  return true

    return false
```

Let's take the ">" as the barometer operation. What is the count for the best case, and what is the count for the worst case? Give your explanation.

Example Answer:

We could describe the input classes based on the location where the variable `countAbove` becomes greater than *N*/2. We know that the first place this could occur is in location *N*/2, and the last place it can occur is in location *N*. We also know that there will be exactly *N*/2 successful comparisons of the list element and the average, which means that the comparison of `countAbove` and *N*/2 will be done exactly that many times. But since "if countAbove > N/2 then return true" used ">" (rather than ">="),

Therefore:

The best case will be (N/2 + 1) + (N/2+1) = N + 2; (case: return TRUE)

The worst case will be N + (N/2 + 1) = 3N/2 + 1 (case: return TRUE)

Students will get credits with the following answers as well.

The best case will be N+1 (case: return FALSE)

The best case will be N (case: all the keys are identical)

Discussion for further consideration (NOT Required):

Above answers assume the case that the code returns TRUE.

If the return value may not be true, there are two cases that we could see for the best case:

One exceptional case: the first N-1 keys smaller than the average, while the last key (Nth key) is greater than the average, e.g., [1, 1, 1, 1, 6], the average is 2. The number total comparison is N+1 (=5+1 = 6), which is better than N+2;

If the keys may not be distinct and the return may not be true:

Another exceptional case: all the keys have a same value. E.g., [1, 1, 1, 1, 1], the average is 1. Then the number of comparison for the above code is "N".

9. [7%]

Alternating disks: You have a row of 2n disks of two colors, n dark and n light. They alternate: dark, light, dark, light, and so on. You want to get all the dark disks to the right-hand end, and all the light disks to the left-hand end. The only moves you are allowed to make are those which interchange the positions of two neighboring disks.

Design an algorithm for solving this puzzle and determine the number of moves it makes. (You algorithm should not be worse than $O(n^2)$. Note: describe your approach without pseudo-code.

Answer:

Here is a simple and efficient (in fact, optimal) algorithm for this problem: Starting with the first and ending with the last light disk, swap it with each of the i $(1 \le i \le n)$ dark disks to the left of it. The ith iteration of the algorithm can be illustrated by the following diagram, in which 1s and 0s correspond to the dark and light disks, respectively.

$$\underbrace{00..0}_{i-1}\underbrace{11..1}_{i-1}\mathbf{1}0\mathbf{1}0..10 \quad \Rightarrow \quad \underbrace{00..0}_{i}\mathbf{0}\underbrace{11..1}_{i}\mathbf{1}10..10$$

The total number of swaps made is equal to $\sum_{i=1}^{n} i = n(n+1)/2$.

NOTE: n(n+1)/2 is the O(n^2). it's ok if the answer is O(n^2).

[Part B]: Programming (warm-up) (15%)

1. Given an array containing n keys, design an algorithm to determine whether there exists such a key which is equal to the summation of other two keys in the array. Explain the worst-case time complexity. (no sequential search is allowed).

Input data (For example: Array A[ ])

A[ ] = 18 23 4 35 99 67 198 20 38 55 2 18 19 487 11 40 10 13 27 22

2. Write-up:

(a) Provide a readme file to explain how to run your program, and show your code and results. (10%).

(b) Explain your implemented algorithm and its worst-case time complexity (5%)

Extra points (5%): Print out all the keys that are equal to the sum of the other two keys in the array, and print out the corresponding two keys.

3. Compress your code package (including code files, Makefile, etc.) and write-up into a .ZIP file (using the name: Your_Last_Name_Programming_Language_Used_Assignment1.zip), and upload it to the blackboard using the digital drop-box. Examples names include Smith_C++_Assignment1.zip or Smith_C_Assignment1.zip.

Upload your code package to the blackboard.

4. The hardcopy of the Part-A must be dropped to the drop-box of CS375-A0 cabinet (The third floor of the computer science department).

Possible Answer (as example) for reference:

Method 1: (1) sorting from small to big; (2) pick up a key, do the summation of the all two-values in the left of the key; (3) sorting again; (4) binary search

(1) Nlgn+ (2) [n^2 + nlgn + lgn]
In total; (1) + (2) *n = n^3

Method 2: (1) sorting from small to big (array A) ; (2) pick up a key, do the subtraction with all the values in the left of the key, save the result in a new array B; (3) search if any of the value in the array B could be same as a value in the A (left part of the key).

(1) Nlgn + [(2) n + (3) nlgn]*n = n^2lgn

Method 3: brute force approach: do the all possible two-value summation N=(n(n-1)/2); sorting (M) ; then for each value of key, do the binary search (lgN)

N(n-1)/2 + M + n (lgN).
Note: M=n^2lgn^2 (because the length of M could be n^2)

So its time complexity could be 2n^2lgn