Homework 2 Solutions

Due: Thursday 22 September 2022 by 11:59 PM

Instructions. Type your answers to the following questions and submit as a PDF on Gradescope by the due date and time listed above. (You may write your solutions by hand, but remember that it is at your own risk as illegible solutions will not receive credit.) Assign pages to questions when you submit. **For all questions, you must show your work and/or provide a justification for your answers.**

Note on Academic Dishonesty. Although you are allowed to discuss these problems with other people, your work must be entirely your own. It is considered academic dishonesty to read anyone else's solution to any of these problems or to share your solution with another student or to look for solutions to these questions online. See the syllabus for more information on academic dishonesty.

Grading. Some of these questions may be graded for completion and some for accuracy.

Question 1.

Classify each of the following functions or summations by order of growth. Show any work you do to simplify and make sure you put items that grow at the same rate in the same classification. Put them all in order from slowest growing to fastest growing. Note: $log(n!)$ is $\theta(nlogn)$.

(a) $4nlogn^2 + log(16^n) = 8nlogn + 4n$

(b) $8nlogn^4 + 4^{logn+1} = 32nlogn + 4n^2$

(c) $log(\frac{32}{16}) = 1$

(d) $3n^2 log(4^n) + log\left(\frac{4n}{12}\right)^2 = 6n^3 + 4 + 2logn - 2log12$

(e) $\sum_{k=1}^{n}(4logk + 1) = 4logn! + n$

(f) $6 + 12 + 24 + 48 + ... + n = 6\left(2^{log(n/6)+1} - 1\right) = 2n - 6$

(g) $6 + 10 + 14 + 18 + ... + n$

$= 2(3 + 5 + 7 + 9 + ... + n/2) = 2\sum_{k=1}^{n/4-1/2}(2k + 1)$

(h) $6 + 12 + 24 + 48 + ... + 6(2)^n$

(i) $\sum_{k=1}^{logn} nk = n\sum_{k=1}^{logn} k = n\left(\frac{logn(logn+1)}{2}\right)$

(j) $\sum_{k=1}^{logn} (n + k) = nlogn + \frac{logn(logn+1)}{2}$

Classification	Items
$\theta(1)$	(c)
$\theta(n)$	(f)
$\theta(nlogn)$	(a), (e), (j)
$\theta(nlog^2 n)$	(i)
$\theta(n^2)$	(b), (g)
$\theta(n^3)$	(d)
$\theta(2^n)$	(h)

Question 2.

For each piece of pseudocode, determine the final value of *sum* and the best and worst case runtimes using appropriate notation. Show your work.

(a) You can assume that N is a multiple of 3. Give a clear justification for your answer.

```
int sum = 0
for(int i = 0; i <= N; i = i + 3)
    for(int j = 1; j <= N; j++)
        sum++
    end for
end for
```

The best and worst case are the same.

i: $0, 3, 6, ..., N$

sum: $N + N + N + N + ... + N = \sum_{k=1}^{N/3+1} N = N(N/3 + 1)$

best/worst case runtimes: $\theta(N^2)$

(b)

```
int sum = 0
for i from 1 to N:
        for(int j = 1; j <= 2i; j++)
                sum++
        end for
end for
```

The best and worst case are the same.

i: $1,\ 2,\ 3,\ \ldots,\ N$

sum: $2 + 4 + 6 + \ldots + 2N = \sum_{k=1}^{N} 2k = 2\left(\frac{N(N+1)}{2}\right) = N(N + 1)$

best/worst case runtimes: $\theta(N^2)$

(c) You can assume that N is a power of 4.

```
int sum = 0
for(int i = 0, i < N; i++)
        for(int j = 1; j <= N; j= j * 4)
                sum++
        end for
end for
```

The best and worst case are the same.

i: $0,\ 1,\ 2,\ \ldots,\ N - 1$

sum: $(logN + 1) + (logN + 1) + \ldots + (logN + 1)$

$sum = \sum_{k=0}^{N-1} (logN + 1) = NlogN + N$

best/worst case runtimes: $\theta(NlogN)$

Question 3.

For each piece of pseudocode, determine the final value of *sum* and the best and worst case runtimes using appropriate notation. Show your work.

(a) You can assume that *N* is a power of 3.

```
int sum = 0
for(int i = 1; i <= N; i = i * 3)
        for(int j = 0; j < 3i; j++)
                sum++
        end for
end for
```

The best and worst case are the same.

i: $1,\ 3,\ 9,\ \ldots,\ N$

sum: $3 + 9 + 27 + \ldots + 3N$

$$sum = 3\sum_{k=0}^{log_3 N} 3^k = 3\left(\frac{3^{log_3 N+1}-1}{2}\right) = \frac{9N-3}{2}$$

best/worst case runtimes: $\theta(N)$

(b) You can assume that *N* is a power of 4.

```
int sum = 0
while N > 0:
    N = N/4
    for i from 1 to N:
        sum++
    end for
end while
```

The best and worst case are the same.

sum: $N/4 + N/16 + ... + 1$

$$sum = \sum_{k=0}^{log_4(N/4)} 4^k = \frac{4^{log_4(N/4)+1}-1}{3} = \frac{N-1}{3}$$

best/worst case runtimes: $\theta(N)$

(c)

```
A = an integer array of size N
sum = 0
for(int i = 0; i < N-1; i++)
     if(A[i] > A[i+1])
          sum++
     end if
end for
```

The value of *sum* depends on the input. It could be as little as 0 (when the array is sorted in non-decreasing order) and as large as *N-1* (when the array is sorted in decreasing order). However, the runtime is the same for both cases, so the best/worst case runtime is $\theta(N)$ because the code loops through the array once either way.

Question 4.

Consider the following pseudocode.

```
Algorithm doSomething
Input: An integer array A of size N and an integer m
Output: ???
if m > A.length then return 0
x = ∞
```

```
s = 0//global variable
p = 0
while (s+m <= A.length):
     p = foo(A, m, p)
     if p > x then x = p
end while
return x

procedure foo(Array A, int m, int p):
     if p != 0 then
          if (s+m-1 < A.length) then
               p = p/A[s-1]
               p = p*A[s+m-1]
          end if
          s++
          return p
     end if
     p = 1
     for i from s to s+m-1:
          p = p*A[i]
          if(A[i] == 0) then
               s = i+1
               return p
          end if
     end for
     s++
     return p
end foo
```

(a) What does this pseudocode do?
(b) Analyze the runtime in terms of N and m. You should consider various inputs that could affect the runtime–best and worst case. Your answer must show that you fully understand what the code is doing and that you have thought about various input scenarios.

(a) It is determining the largest product of consecutive *m* integers in the array.
(b)
Note that foo essentially calculates the product for a window of size *m* starting at index *s*.
Since the code for *foo* works differently depending on whether or not the product is 0 or not, we should consider the extremes.

1. The array is all zeroes: In that case, *foo* will go to the second loop, hit the first element, see that it is 0, and return. That means it will do O(1) work, which will be repeated O(N) times. So the runtime is O(N).
2. The array has no zeroes: Since the product value starts as 0, the second loop in *foo* will run the first time, doing O(m) work for the first window, and *s* will go up by 1. After that, the product will be non-zero, so the first foo-loop will run. It does O(1) work for the division and multiplication, and *s* is increased by 1. So this will happen O(N) times. That is a total of O(m+N) work.
3. The array is of the form [a, b, c,...,0, d, e, f,..., 0…] where the letters represent non-zero integers and the 0's appear at indexes m-1, 2m-1, 3m-1, and so on. This would mean that the second loop in foo would run each time it is called, and it would do O(m) work before hitting the 0. However, *s* will then be increased to be after that window, so *foo* will only be called about *N/m* times. That gives a total O(N) work.

Question 5.

Hint: For these questions, it sometimes helps to consider a *twoSum* problem first–that is, first consider how you might determine the number of pairs in an array that add to 0.

(a) In the slides, you saw an algorithm called *threeSum* that counted the number of triples in an integer array that summed to 0. Here, you should describe a solution to a variation of that problem. We are assuming that the input array is sorted. Describe and analyze an algorithm that solves the threeSum problem with a better runtime than the brute-force approach.
Analyze the best/worst case runtimes of your algorithm and give a brief explanation for why your algorithm is correct.

We can do a brute-force search of all the pairs (x, y) and then use a binary search to find -(x+y).

```
Input: an array A of N non-decreasing integers
Output: the number of triples that sum to 0
count = 0
for(int i = 0; i < N; i++)
     for(int j = i+1; j < N; j++)
          k = binarySearch(A[j+1:], -(A[i]+A[j]))
          if(k != -1)
               count++
          end if
     end for
end for
return count
```

Analysis: The two for-loops run O(N^2) times because there are N(N-1)/2 pairs in the array. For each one of those pairs, we do a binary search, which has an O(logN) runtime. So that gives us a runtime of $O(N^2 logN)$.

(b) Now let's assume that the numbers in the input array are sorted AND unique. Describe an algorithm that solves the *threeSum* problem with a runtime that is even better than the one in Question 11. Analyze the best/worst case runtimes and give a brief explanation for why your algorithm is correct.

```
Input: an array A of N unique non-decreasing integers
Output: the number of triples that sum to 0
for(int i = 0; i < N-2; i++)
     j = i+1
     k = N-1
     while(k > j)
          if(A[j] + A[k] == -A[i])
               count++
               j++
               k--
          else if(A[j] + A[k] < -A[i])
               j++
          else
               k-
          end if
     end while
end for
```

Analysis: The inner while loop runs through all the elements from j to k, a range that gets smaller as i gets larger.
When i = 0, that runtime is N-1.
When i = 1, that runtime is N-2.
When i = 3, that runtime is N-3.
…
When i = N-3, that runtime is 2.

Summing that up would give a runtime of $O(N^2)$.