COL106: Solutions for Quiz-2

(October 8, 2020)

Question 1. What is the worst case time complexity of the modified version of insertion sort algorithm where we use binary search for finding the correct position for inserting the next number?

(A) $O(n)$

(B) $O(n \log n)$

(C) $O(n^2)$

(D) $O(n^2/\log n)$

Answer: (C). Irrespective of the time taken for searching for finding the correct position, it will take $j-1$ swaps in the worst case to insert the j^{th} element in the sorted list. Hence the total time will be $\sum_{j=2}^{n}(j-1)$ in the worst case, which is $O(n^2)$.

Question 2. A stack sortable permutation is defined as a permutation whose elements may be sorted by an algorithm whose internal storage is limited to a single stack data structure. For example, the permutation (2,1,3) can be sorted as follows:

```
Push 2 from input to stack
Push 1 from input to stack
Pop 1 from stack to output
Pop 2 from stack to output
Push 3 from input to stack
Pop 3 from stack to output.
```

However, the permutation (2,3,1) cannot be sorted in this manner. Which of the following is not a stack sortable permutation?

(A) (5,1,4,2,3)

(B) (3,4,5,1,2)

(C) (5,4,1,2,3)

(D) (1,2,3,5,4)

Answer: (B). Try out the examples yourself. It is interesting to note that a stack sortable permutation always takes the following form: If we consider the sequence in sorted non-decreasing order: let this sequence be $<s_1, s_2, s_3, \ldots, s_n>$. Then the permutation is stack

sortable if and only if there exists an integer i: $1 \leq i \leq n$, such that, it can be obtained by any interleaving of the two sequences: $<s_1, s_2, \ldots, s_i>$ and $<s_n, s_{n-1}, \ldots, s_{i+1}>$.

Question 3. Compute the time complexity of the following code :

```
int sum = 0, j = 1;
int n = <some value>

for (int i =0; i<n; i++)
{
    while (j<i)
    {
        sum += 1;
        j *= 2;
    }
}
```

(A) $\Theta(n \log n)$

(B) $\Theta(n^2)$

(C) $\Theta(\log n)$

(D) $\Theta(n)$

Answer: (D). The for loop is executed $\Theta(n)$ times. Thus, the while loop condition will be checked $\Theta(n)$ times. Note that j is only initialized once in the beginning. It is not reinitialized every time in the i loop. The while loop will only be entered when $j = i - 1$ and it is doubled in the loop. Thus the loop statements will never execute more than once in any given iteration of the i loop. Hence the while loop will be executed $O(n)$ times (actually one can argue that they will only be executed $O(\log n)$ times, but that does not change the overall complexity).

Question 4. What exactly is average case time complexity?

(A) The average of best case and worst case time complexity.

(B) The time complexity averaged over all possible input instances.

(C) The worst case time complexity averaged over all possible values of n, the input size

(D) It is worst case time complexity itself

Answer: (B). By definition.

Question 5. Which of the following statements is/are correct?

1. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$, then $f(n) + d(n) = O(min(g(n), e(n))$
2. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$, then $f(n) \cdot d(n) = O(g(n) \cdot e(n))$

3. If $f(n)$ is $O(g(n))$, then $g(n)$ is $\Omega(f(n))$

4. If $f(n) = \Theta(g(n))$, then $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$

(A) 1,3,4

(B) 1,2,4

(C) 1,2,3

(D) 2,3,4

Answer: (D). It is easy to see that Statement 1 is not true by taking $f(n) = n$ and $g(n) = n^2$. The other statements can be verified by applying the definitions. A formal proof is illustrated here for Statement 4. Try to prove the other statements formally yourself.

Proof. $f(n) = \Theta(g(n)) \implies$ there exist constants c_0, c_1, n_0, n_1 such that

$$f(n) \leq c_0 \cdot g(n) \text{ for all } n \geq n_0 \text{ and}$$

$$f(n) \geq c_1 \cdot g(n) \text{ for all } n \geq n_1.$$

Let $n_2 = \max\{n_0, n_1\}$. Then

$$c_1 \cdot g(n) \leq f(n) \leq c_0 \cdot g(n) \text{ for all } n \geq n_2.$$

In other words,

$$\frac{1}{c_0} \cdot f(n) \leq g(n) \leq \frac{1}{c_1} \cdot f(n) \text{ for all } n \geq n_2.$$

This means that $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$. □

Question 6. Following is an incorrect pseudocode for the algorithm which is supposed to determine whether a sequence of parentheses is balanced:

```
declare a character stack
while ( more input is available )
{
    read a character
    if ( the character is a '(' )
        push it on the stack
    else if ( the character is a ')' and the stack is not empty )
        pop a character off the stack
    else
        print "unbalanced" and exit
}
print "balanced"
```

Which of these unbalanced sequences does the above code think is balanced?

(A) ((())

(B) ())(()

(C) (()()))

(D) (()))()

Answer: (A). Try all the inputs. It is interesting to note that the program accepts all inputs that are prefix of a sequence of parentheses that are balanced. A correct implementation would just need to check at the end that the stack is empty.

Question 7. In the lecture, we have been taught two strategies for increasing Stack size, namely, Growth Strategy and Tight Strategy. Now, we want to analyze what-if we set the size of the array to the square of the current Phase (as in lecture) every time we reach the end. This way, we can start with an array size of 1, then square 2 in phase 2 to get an array of size 4, then 9, then 16 and so on.

What will be the complexity, in this case, to fill the stack up to n ?

(A) $O(n^2)$

(B) $O(n \log n)$

(C) $O(n^{3/2})$

(D) $O(n^2 \log n)$

Answer: (C). In phase i, $(i-1)^2$ elements need to be copied and then the stack can be filled with another $i^2 - (i-1)^2$ elements. Thus the complexity of the i^{th} phase is $\Theta(i^2)$. In all to fill n elements there will be $\sqrt{n}$ phases. Hence the total complexity will be:

$$\sum_{i=1}^{\sqrt{n}} \Theta(i^2) = \Theta(\sqrt{n}^3) = \Theta(n^{3/2}).$$

Hence the best answer is $O(n^{3/2})$.

Question 8. How many swaps are required to sort the following array using insertion sort?

{ 15, 1, 10, 2, 5, 4, 8 }

(A) 13

(B) 15

(C) 11

(D) 18

Answer: (C). You should be able to check yourself. The number of swaps in iterations 1,2,3,4,5,6 are 1,1,2,2,3,2 respectively for a total of 11.

Question 9. The characters of a string are pushed into a stack. Consider an optimal algorithm to check whether the string is palindrome using stacks only. What is the minimum number of **EXTRA** stacks that will be needed? Recall that a palindrome is a string that reads the same backwards, e.g. "racecar".

(A) 0

(B) 1

(C) 2

(D) Palindrome cannot be checked using Stacks only.

Answer: (B). Suppose the size of the string is n. There is a simple algorithm to check for a palindrome using one extra stack. We pop out $\lfloor n/2 \rfloor$ elements and push them into the extra stack one by one. We then repeatedly pop out 1 element from both the stacks and check if they are the same till no more elements are let in the stacks. Note that there will be one extra element in case the total size of the string is an odd number; this can easily be handled by discarding the extra (middle) element once $\lfloor n/2 \rfloor$ elements have been moved to the extra stack as this element does not need to be matched.

Question 10. Let F be the set of all functions from $N \leftarrow \Re^+$ and $f(n), g(n) \in F$. Consider the statements:

1. $f(n) = O(g(n))$
2. $f(n) = \Theta(g(n))$
3. $f(n) = \Omega(g(n)$

What can you say about these statements ?

(A) At least one of the above statements must be true for any given pair of functions $f(n)$ and $g(n)$

(B) For some functions $f(n)$ and $g(n)$, none of the statements need to be true

(C) More than one of the statements is true for any given pair of functions $f(n)$ and $g(n)$

(D) All statements are true for any given pair of functions $f(n)$ and $g(n)$

Answer: (B). Consider the functions:

$$f(n) = \begin{cases} n & \text{if } n \text{ is prime} \\ n^3 & \text{if } n \text{ is not prime} \end{cases}$$

and $\quad g(n) = n^2.$

Question 11. Let F be the set of all functions from $N \leftarrow \Re^+$ and $f(n), g(n) \in F$. Then which of the statements is true?

(A) $f(n) + g(n) = \Theta((max(f(n), g(n)))$

(B) $f(n) + o(f(n)) = \Theta(f(n))$

(C) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$

(D) $f(n) = \Theta(f(n/2))$

Answer: (A). Statements (A) and (B) are easy to check. Let us see why (C) and (D) are not true.

(C) Consider the functions $f(n) = 2n$ and $g(n) = n$. Then $2n$ is $O(n)$, but 2^{2n} is not $O(2^n)$.

(D) Consider the function $f(n) = 2^n$. Then 2^n is not $\Theta(2^{n/2})$.

Question 12. let $W(n)$ be worst case running time of an algorithm, $A(n)$ be it's average case running time and $B(n)$ be best case running time of algorithm. Then which of the following is true

(A) $A(n) + B(n)$ is $O(W(n))$

(B) $A(n) + B(n)$ is $\Theta(W(n))$

(C) $B(n) + W(n)$ is $\Theta(A(n))$

(D) $B(n) + W(n)$ is $O(A(n))$

Answer: (A). Clearly $B(n) \leq A(n) \leq W(n)$. Thus (A) is true. To see that the other options are not true, consider a program that takes n binary inputs. Thus there are 2^n input instances. Suppose that the algorithm runs in time $\Theta(1)$ for $2^n - 1$ of these input instances and takes time $\Theta(2^n)$ for one instance. Then $B(n) = \Theta(1)$, $A(n) = \Theta(1)$ and $W(n) = \Theta(2^n)$.