COMP 3270

Homework 1

1.) The worst case run time of the algorithm is O(n), meaning this is a linear time algorithm. This would occur when the while loop present in the algorithm has to iterate through each array element.

2.) One possible solution would be, to check each integer from the array to make sure that it is indeed within the range, and whichever array element does not fit within range upon checking, will be the answer.

3.) For a string of length n, we would need to go through each character & convert each one to its integer value. & after we do it to the n^{th} character in the string, convert the entire number into one singular integer

4.) Given that a computer can solve 2×10^7 steps per second

To get time: steps / steps per second

Linear O(n) A1: 2×10^8 → since O(n) 2×10^8 steps

$\frac{2 \times 10^8}{2 \times 10^7} =$ **10 seconds**

O(nlogn) A2: $2 \times 10^8 \cdot \log(2 \times 10^8) \rightarrow 1660205999$ steps

$\approx \frac{1.66 \times 10^9}{2 \times 10^7} =$ **83 seconds**

$O(n^2)$ A3: $(2 \times 10^8)^2 \rightarrow 2 \times 10^{16}$ steps

$= \frac{2 \times 10^{16}}{2 \times 10^7} =$ **10^9 seconds**

5.) Input: Campus Building that a user selects

Data Representations: Both the campus building + parking spots will be represented as sets of (longitude, latitude) coordinates, along with the user's location being represented by a singular (longitude, latitude) coordinate. Also occupied + unoccupied sets of parking spots.

Outputs: The set of (longitude, latitude) coordinates of the five nearest parking spots that are open to the user selected building, along with turn by turn directions. Or it could output none if there are not 5 open spots anywhere

6.) To find the pair that will produce the maximum difference between 2 numbers, we can loop through the array, picking out the minimum and maximum numbers from the array, and outputting the pair. Since the minimum and maximum numbers in any sequence will have the largest difference out of any pairs. This algorithm will run with a Big-O time of O(n) since it only has one loop that will execute one time over the size of the array.

An algorithm containing n numbers will have $\frac{n(n-1)}{2}$ unique pairs + n(n-1) total pairs. Since for every number in the ~~sequence~~, we can eliminate one possible pair: Since we are only making pairs, then we know the first pair can be picked ~~two~~ times, + since we don't want the first picked again, then the 2nd can be picked N-1 times, giving us a total of N(N-1) possible ways.

COMP 3270

Homework 1

7.) a.) The algorithm is designed to go through all possible subarrays of the given array, & identify the largest partial sum from the ~~given~~ subarrays and output this value as "m".

For n=10: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

j=1 1=1
j=2 1+2=3
j=3 1+2+3=6
j=4 1+2+3+4=10
j=5 1+2+3+4+5=15
j=6 1+2+3+4+5+6=21
j=7 1+2+3+4+5+6+7=28
j=8 1+2+3+4+5+6+7+8=36
j=9 1+2+3+4+5+6+7+8+9=45
j=10 1+2+3+4+5+6+7+8+9+10=55

m=55

b.) The time complexity of the algo is $\Theta(n^3)$

c.) The new algorithm goes through the given array & stores the partial sums in a seperate non-nested loop & then goes through another loop where the partial sums are then compared for a Big-O time of $\Theta(n^2)$.

8.)

Step	Big-Oh			
1				
2			$O(n)$	
3		$O(n)$	$O(n^2)$	
4		$O(1)$	$O(n)$	
5	$O(1)$	$O(n)$	$O(n^2)$	$O(n^3)$
6	$O(1)$	$O(n)$	$O(n^2)$	$O(n^3)$
7	$O(1)$	$O(n)$	$O(n^2)$	$O(n^3)$
8	$O(1)$	$O(n)$	$O(n^2)$	$O(n^3)$
9				

[Complexity of Algorithm = $O(n^3)$]

9.)

Steps	Cost	# of times
1	1	1
2	1	$n+1$
3	1	$n(n+1)$
4	1	n^2
5	1	$\sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+2)$
6	6	$\sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+1)$
7	4	n^2
8	2	n^2
9	1	1

$$T(n) = 1\cdot 1 + 1(n+1) + 1(n(n+1)) + 1\cdot n^2 + 1\cdot\sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+2) + 6\cdot\sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+1)$$

$$+ 4n^2 + 2n^2 + 1$$

$$T(n) = 1 + n+1 + n^2 + n + n^2 + \sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+2) + 6\cdot\sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+1)$$

$$+4n^2 + 2n^2 + 1$$

$$\boxed{T(n) = 8n^2 + 2n + 3n + \sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+2) + 6\cdot\sum_{j=1}^{n}\sum_{k=j}^{n}(k-j+1)}$$

10.) The algorithm is incorrect + can be seen when it is presented with 2 consecutive numbers. The example given shows 2 consecutive 6's, which should move the count to 5, however, when 'i' reads the first 6 and 'j' reads the second 6, it will increment count by 1, making count=4. Continuing, the algorithm will then set i=6, and then read in 7 for j. which will not increment count, therefore not including the second 6 and leaving count=4, making the algorithm incorrect by counter-example

11.) 4. For the base cases F_0 and F_1 in Fibonacci numbers, our algorithm defines these base cases as $n=0$ and $n=1$, for which our algorithm will immediately return 1, for both $n=0$ and $n=1$. So in both cases, our algorithm returns the correct answer.

6. When $n=k$ and $k>1$, then the algorithm has a condition where the base cases are satisfied + steps 3-9 will be executed

8.) If $k=3$, the for loop in steps 5-8 will execute twice. The first iteration will return F_2 + update our "last" value to take on F_1 + the "current" value will be F_2. This means $temp = F_0 + F_2$ or $F_3 = F_1 + F_2$, and just like for $k=2$, the values for last + current will update to take on F_2 + F_3.

9.) But if $k=4$, then we will iterate up to $k=4$, where $F_4 = F_3 + F_2$, just like the above iterations.

10.) The above argument can be repeated to show that for any n, where $n>1$, then $F_n = F_{n-1} + F_{n-2}$.

12. a)

```
String-reverse(A[p...q], p, q)
    n = q - p + 1
    if n < 1
        swap(A[p], A[q])
    String-reverse(A[p...q], p+1, q-1)
```

b)

```
String-reverse([i<33270!], 1, 8)
    ↑ !07233<i
String-reverse([i<33270!], 2, 7)
    ↑ i072334!
String-reverse([i<33270!], 3, 6)
    ↑ i<72330!
String-reverse([i<33270!], 4, 5)
    ↑ i<32370!
String-reverse([i<33270!], 5, 4)
    ↑ return
```

13.) Base Case Proof:
If $n \leq 1$ then our function will return the value of "n":
$n=0$: $3^0 - 2^0 = 1 - 1 = 0$ ✓
$n=1$: $3^1 - 2^1 = 1$ ✓

Inductive Hypothesis:
Our algorithm, for any $k \geq 0$ (non-negative integer) will compute + return the answer of $3^n - 2^n$, so, $g(k) = 3^k - 2^k$

Inductive Step:
$g(k+1) = 5g(k+1-1) - 6g(k+1-2)$
$g(k+1) = 5g(k) - 6g(k-1)$
$g(k+1) = 5(3^k - 2^k) - 6(3^{k-1} - 2^{k-1})$
$g(k+1) = 15^k - 10^k - 18^{k-1} - 12^{k-1}$
$g(k+1) = 15 \cdot 3^{k-1} - 10 \cdot 2^{k-1} - 6 \cdot 3^{k-1} - 6 \cdot 2^{k-1}$
$g(k+1) = 9 \cdot 3^{k-1} - 4 \cdot 2^{k-1}$
$g(k+1) = 3^2 \cdot 3^{k-1} - 2^2 \cdot 2^{k-1} \rightarrow \boxed{g(k+1) = 3^{k+1} - 2^{k+1}}$

14.) Loop Invariant: Before any execution of the for loop of line 5 in which the loop variable $i = k$, $2 \leq k \leq n$, the variable last will contain "1" + the variable current will contain "1".

Initialization: When the loop is being initialized the values of current and last take on values of 1. So here, $F_1 = 1$ and $F_0 = 1$, which following our formula, $F_k = F_{k-1} + F_{k-2}$, $F_2 = F_1 + F_0$, which our loop invariant is true before any iteration

Maintenance: Say we are on the $i = k^{th}$ iteration, this means that $F_k = F_{k-2} + F_{k-3}$, or, for the purpose of our algorithm, temp = last + current, where temp = F_k, last = F_{k-2}, + current = F_{k-1}. Moving through this iteration, our algo will set last = current $\rightarrow F_{k-2} = F_{k-1}$ and current = temp $\rightarrow F_{k-1} = F_k$, + this occurs before the k+1 case, meaning our loop invariant holds true.

$k-j+2$ for $i=j$ to k $\Sigma\ k-j+2$

14.) Termination: The loop will continue to perform as stated, until it reaches the $n+1$ iteration, where it will then exit the loop since i is limited by n. The algorithm will return the value of "temp" stored by the n^{th} iteration.