Problem 2

a)
```
Stack s; //All operations on a stack take constant time. ∴ Θ(1)
for (int i=0; i<n; i++){
    for (int j=i; j<n; j++){
        s.push(j);
    }
    for (int k=n; k>i; k--){
        s.pop();
    }
}
while (!s.empty()) {
    s.pop();
}
```

$\Theta(1)$] $\sum_{j=i}^{n-1}\Theta(1)$

$\Theta(1)$] $\sum_{k=i+1}^{n}\Theta(1)$

] $\sum_{i=0}^{n-1}\left(\sum_{j=i}^{n-1}\Theta(1)+\sum_{k=i+1}^{n}\Theta(1)\right)$

] s.empty() == true, ∴ while loop will not activate

Worst-case Upper-bound Run-time = Worst-case Lower-bound Run-time

∴ We found the tightest bound.

$$\sum_{i=0}^{n-1}\left(\sum_{j=i}^{n-1}\Theta(1)+\sum_{k=i+1}^{n}\Theta(1)\right)=\underline{\sum_{i=0}^{n-1}\big(\Theta(n-1-i)+\Theta(n-i-1)\big)}$$

↓

$$\Theta(n-1)+\Theta(n-2)+\Theta(n-3)+\dots+\Theta(1)+\Theta(0)$$

∴ Arithmetic sequence in reverse

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

$$=\underline{n^2+n} \quad \therefore \underline{\Theta(n^2)}$$

all inputs run at least that time

Runtime for any n as there is no control flow and upper bound is $O(n^2)$ and lower bound is $\Omega(n^2)$ as

∴ $T(n)$ is $\underline{\Theta(n^2)}$ as $O(n^2)$ and $\Omega(n^2)$ have the same growth rate.

☆ while loop is never executed as the number of times an integer is pushed onto stack s, it is popped of by the same number in every iteration of i. ∴ s is always empty right before the while loop starts.

```
b) func (0,n);
void func(int curr, int n) {
    if (n <= 0) return;
    if (curr<=0) func (n-1, n-1);
    else func(curr-1, n);
}
```

Convert from head recursion to iterative version

```
void func (int curr, int n) {
    for(curr=n; curr>0; curr--){
        for (int i = curr-1; i>0; i--){
        }
    }
}
```

$\sum_{i=1}^{curr-1} \Theta(1)$ – $\sum_{curr=1}^{n} \sum_{i=1}^{curr-1} \Theta(1)$

Worst-case Upper-bound Run-time = Worst-case Lower-bound Run-time

∵ We found the tighest bound

$$\sum_{curr=1}^{n} \sum_{i=1}^{curr-1} \Theta(1) = \sum_{curr=1}^{n} \Theta(curr-1)$$

$$\Theta(0) + \Theta(1) + ... + \Theta(n-1) + \Theta(n)$$

∴ Arithmetic Sequence

$$= \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \quad \therefore \underline{\Theta(n^2)}$$

∴ T(n) is $\underline{\Theta(n^2)}$ as $O(n^2)$ and $\Omega(n^2)$ have the same growth rate.

Runtime for any n as the upper-bound is $O(n^2)$ and lower-bound is $\Omega(n^2)$ as all inputs take at least that time to run.

c) Queue q; //all operations on a Queue take constant time.

```
for(int i=1; i<=n; i++){
    q.enqueue(i);
}
bool swap = false;
while(!q.empty()){
    if(swap){
        if(q.front()==1){
            for(int i=n+1; i<=2n; i++){
                q.enqueue(i);
            }
        }
        q.dequeue();
    }
    else {
        q.enqueue(q.front());
        q.dequeue();
    }
    swap = !swap;
}
```

$\Theta(1)$] $\sum_{i=1}^{n}\Theta(1) = \Theta(n)$

$\Theta(1)$

$\Theta(1)$] $\sum_{i=n+1}^{2n}\Theta(1) \approx \Theta(n)$ removed only $\frac{1}{2}$ the time!

$\Theta(1)$; Removes one integer from queue

For every m # of integers in the queue, it will take 2m times to remove it from the queue as one integer is

$\Theta(1)$; Size of queue remains the same

Worst-case Upper-bound Run-time = Worst-case Lower-bound Run-time

∴ We found the tightest bound.

1. It takes $\Theta(n)$ at the start to enqueue n integers into queue.
2. It takes $\Theta(2n)$ to dequeue the first 1 to n integers in the queue as a single integer is dequeued half the time the while loop runs as explained above.
3. It takes $\Theta(n)$ to enqueue the next n+1 to 2n integers in the queue after q.front()==1
4. It takes $\Theta(2n)$ to dequeue the n+1 to 2n integers in the queue as a single integer is dequeued half the time the while loop runs as explained above.
5. while loop ends when q.empty() == true.

∴ T(n) is $\Theta(n) + \Theta(2n) + \Theta(n) + \Theta(2n) \approx \underline{\Theta(n)}$ as $O(n)$ and $\Omega(n)$ have the same growth rate.

Runtime for any n as the upper-bound is $O(n)$ and lower-bound is $\Omega(n)$ as all inputs take at least that time to run.

d)
```
struct Node {
    int value;
    Node* next;
    Node(int i): value(i){};
};

Node* head = NULL;
for( int i=0; i<n; i++){
    Node* curr = new Node(i);
    curr->next = head;
    head = curr;
}
for(int i=1; i<n; i++){
    Node* curr = head;
    while(curr) {
        if( (curr->value % i==0)&&(arr[i]==0)){
            for (int j= arr[i]; j<n; j++){
                arr[j] *=2;
            }
        }
        curr = curr->next;
    }
}
```

Worst-case arr: arr filled with only 0s as this will allow the execution of the if statement to only depend on value of curr.

$\Theta(1)$ — $\sum_{i=0}^{n-1}\Theta(1) \approx \underline{\Theta(n)}$

Only executes when (curr→value) is a multiple of i, ∴ Since there are 'n' (curr→values)/iterations of while loop

$\Theta(1)$ — $\sum_{j=0}^{n-1}\Theta(1)$ $\approx \underline{\Theta(n)}$

For a given i, there will be n/i multiples.

$\therefore \sum_{i=1}^{n-1}\left(\frac{n}{i}\right)\Theta(n)$

$= n^2\sum_{i=1}^{n-1}\frac{1}{i}$ ☆ Harmonic Series

$\approx \underline{\Theta(n^2\log n)}$

Worst-case Upper-bound Run-time = Worst-case Lower-bound Run-time

∵ We found the tightest bound.

∴ T(n) is $\underline{\Theta(n^2\log n)}$ as $O(n^2\log n)$ and $\Omega(n^2\log n)$ have the same growthrate

Runtime for any n as the upper-bound is $O(n^2\log n)$ and lower-bound is $\Omega(n^2)$ as all inputs take at least that time to run.

Scanned by CamScanner

For my own reference

$$\sum_{k=0}^{\lg n}(4^k) = \sum_{k=0}^{\lg n} 2^{2k} \rightarrow O(2^{2(\lg n)}) = O(n^2)$$

Dominates

$$\left(\frac{1}{n}\right)\left(O(n^2) + O(n\lg n) - \frac{\lg n(\lg n+1)}{2}\right) = \underline{O(n)}$$

Problem 3

```
void someclass::somefunc(){
    if(this->n == this->max){
        bar();
        this->max *= 2;
    }else{ foo();}
    (this->n)++;
}
```

Assume that when someclass is created, n=0 and max=1.

a) Worst-case runtime for somefunc $T(n)$: $\underline{\Theta(n^2)}$

(When <u>if</u> statement executes and bar() which takes $\Theta(n^2)$ time is called instead of <u>else</u> which calls foo() which takes $\Theta(\log n)$ time.)

b)

	foo()	bar()	bar()	foo()	bar()	foo()	foo()	foo()	bar()	foo()	foo()	foo()	foo()	foo()	foo()	foo()
n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
max	1	1	2	4	4	8	8	8	8	16	16	16	16	16	16	16

Total runtime of bar(): $1^2 + 2^2 + 4^2 + 8^2 + 16^2 + 32^2 + \ldots$

$$= (2^0)^2 + (2^1)^2 + (2^2)^2 + (2^3)^2 + (2^4)^2 + (2^5)^2 + (2^6)^2 + \ldots + (2^k)^2, \text{ where } k = \lg n$$

How many powers of 2 are there in n operations: $\lg n$

$\therefore$ Amortized runtime of some func: $\left(\frac{1}{n}\right)\left(\sum_{k=0}^{\lg n}(2^k)^2 + \underbrace{\left(\sum_{i=0}^{n}(\lg n) - \sum_{i=0}^{\lg n}(\lg(2^i))\right)}\right)$

Method 1

$$= \left(\frac{1}{n}\right)\left(\underbrace{\sum_{k=0}^{\lg n}(4)^k}_{\text{Geometric}} + \sum_{i=0}^{n}(\lg n) - \cancel{\lg(2)}\underbrace{\sum_{i=0}^{\lg n} i}_{\text{Arithmetic}}\right)$$

Total Runtime of foo(), sum of $\lg n$ from $n=0$ to $n=n$, subtracted by sum of $\lg(2^i)$ from $i=0$ to $i=\lg n$ to account for times when if statement is executed instead of else.

☆ In a cycle of n operations from when n==max to the next n==max, bar() would occur once, while foo() would occur n-1 times.

Method 2

bar() foo()

$\therefore$ Amortized Runtime of somefunc: $\left(\frac{1}{n}\right)\left(\Theta(n^2) + \sum_{m=n}^{2n-1}\Theta(\lg n)\right)$

$$= \underbrace{\Theta(n^2) + [\Theta(\lg n) + \Theta(\lg(n+1)) + \ldots + \Theta(\lg(2n-1))]}_{n}$$

Dominates

$\approx \underline{\Theta(n)}$

Dominates

$\approx \underline{\Theta(n)}$

c) Amortized Runtime of somefunc if foo() is $\Theta(n\log n)$

$$= \underbrace{\Theta(n^2) + [\Theta(n\lg n) + \Theta((n+1)\lg(n+1)) + \ldots + \Theta((2n-1)\lg(2n-1))]}_{n}$$

```
d) void someclass::anotherfunc(){
       if(this->n > 0){
           (this->n)--;
       }
       if(this->n < (this->max)/2){
           bar(); - θ(n^2)
           this->max /=2;
       }else{ foo();} - θ(logn)
   }
```

Worst-case sequence:

$n=0$, $max=1$

somefunc() → foo();

$n=1$, $max=1$

somefunc() → bar();

$n=2$, $max=2$

somefunc() → bar();

$n=3$, $max=4$

anotherfunc() → foo();

$n=2$, $max=4$

anotherfunc() → bar();

$n=2$, $max=2$

Cycle

The worst-case sequence happens when somefunc(), followed by anotherfunc(), followed by anotherfunc is called when n==max as bar() is executed twice in each cycle and once for foo().

∴ Amortized runtime / function call: $\underbrace{\frac{\theta(n^2)+\theta(\lg(n+1))+\theta(n^2)}{3}}_{\text{Dominates}}$

$\approx \underline{\theta(n^2)}$