CSC1-163 HW#1 Jesse Mayer

1.1-6: a. gcd(31415, 14142) = gcd(14142, 31415 mod 14142)
31415 mod 14142 = 31415 - (2)(14142) = 3131
gcd(14142, 3131) = gcd(3131, 14142 mod 3131)
14142 mod 3131 = 14142 - 12524 = 1618
gcd(3131, 1618) = gcd(1618, 3131 mod 1618)
3131 mod 1618 = 3131 - 1618 = 1513
gcd(1618, 1513) = gcd(1513, 1618 mod 1513)
1618 mod 1513 = 1618 - 1513 = 105
gcd(1513, 105) = gcd(105, 1513 mod 105)
1513 mod 105 = 1513 - 1470 = 43
gcd(105, 43) = gcd(43, 105 mod 43)
105 mod 43 = 105 - 86 = 19
gcd(43, 19) = gcd(19, 43 mod 19)
43 mod 19 = 43 - 38 = 5
gcd(19, 5) = gcd(5, 19 mod 5)
19 mod 5 = 19 - 15 = 4
gcd(5, 4) = gcd(4, 5 mod 4)
5 mod 4 = 5 - 4 = 1
gcd(4, 1) = gcd(1, 4 mod 1
4 mod 1 = 0
gcd(1, 0) = 1

b. Euclid's algorithm used 10 iterations to find gcd(3145,1414 while the algorithm based on checking consecutive integers from min{m, n} down to gcd(m, n) would take 14141 iterations. Therefore Euclid's algorithm is roughly 1414 times faster than the consecutive integer algorithm.

| 1.1-12: locker # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1st Pass | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2nd Pass | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 3rd Pass | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 4th Pass | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 5th Pass | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

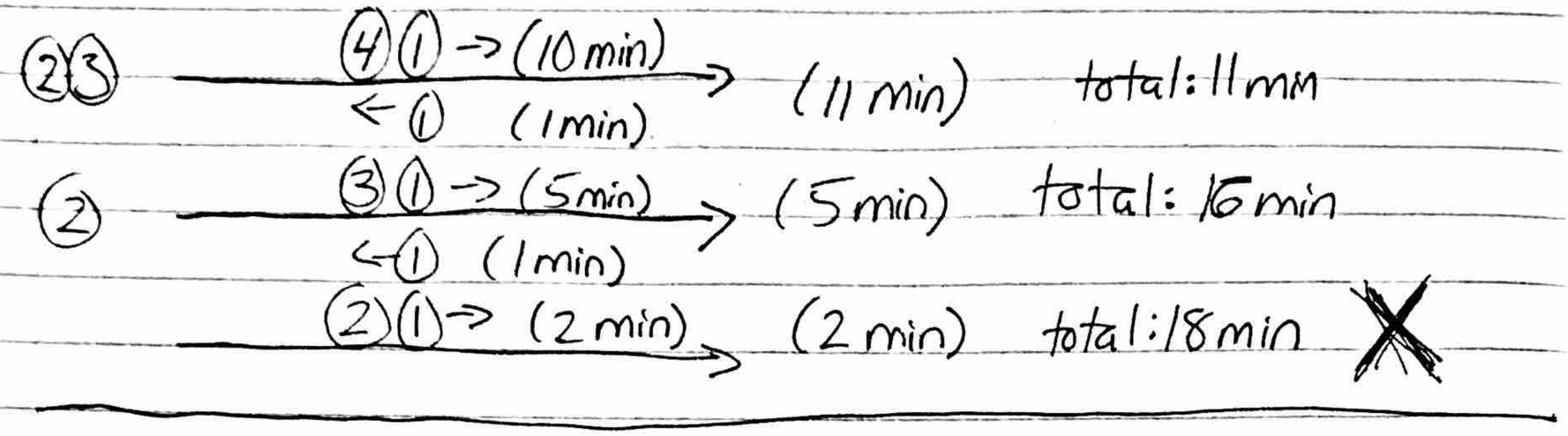
$n=1 \Rightarrow 1$ open
$n=4 \Rightarrow 2$ open
$n=9 \Rightarrow 3$ open

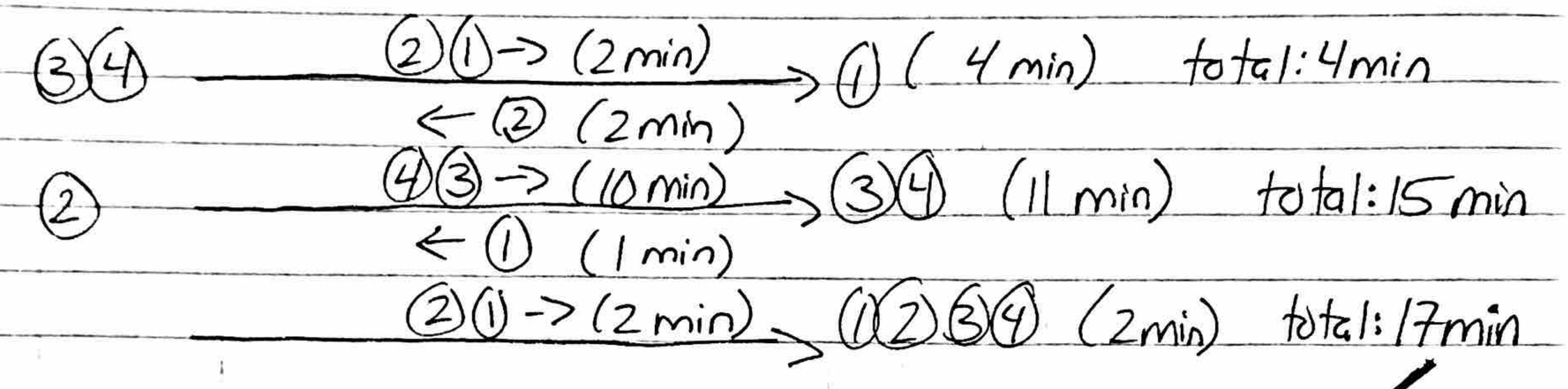
| locker # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 6th Pass | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 7th Pass | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 8th Pass | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 9th Pass | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 10th Pass | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

For $n>1$ lockers at the end of the last pass, locker #1 is open followed by [illegible] the locker 3 counts after the previously open locker and then 5 counts after the next previously open locker, thus increasing the count between the last open locker and the next by +2 starting at 3. Therefore at the end there will be a number of doors open equal to the greatest number of odd integers starting at 1 and incremented by +2 [illegible] whose sum is less than or equal to $n$.

1.2-2: person#1 - 1 minute, person#2 - 2 minutes, person#3 - 5 minutes, person #4 - 10 minutes (17 minute time limit)

③④ ①② → (2 min) → ② (3 min) total: 3 min
← ① (1 min)
③① → (5 min) → ②③ (6 min) total: 9 min
← ① (1 min)
④① → (10 min) → ①②③④ (10 min) total: 19 min ✗




1.2-4: FindRoots(int a, int b, int c)

~~d ←~~ d ← b^2 - 4*a*c

d ← sqrt(d)

r_1 ← (-b + d) / (2*a)

r_2 ← (-b - d) / (2*a)

return r_1, r_2

1.2-5: a. For a given positive decimal integer i find the greatest ~~[illegible]~~ value of n for which $2^n$ is less than or equal to i and place a 1 in the $n^{th}$ decimal place. ~~and fill the rest with 0. Then subtract from i & then check if~~ Then subtract $2^n$ from i and check if $2^{n-1}$ is less than or equal to i, if so put 1 in the $(n-1)^{th}$ decimal place. ~~and subtract~~ Otherwise fill the decimal place with 0 and continue to decrement the exponent of 2 following

the previous steps.

b. IntToBin(int i)  n←0
while (!(i < 2^n ∧ (n+1)) && i > 2^n ∧ n)) do
    n←n+1
    t←n
    bin←"1"
    for t>0
        bin←bin+"0"
        t←t-1
return bin + IntToBin(i-2^n)

1.3-1: a. A | 60 | 35 | 81 | 98 | 14 | 47 |

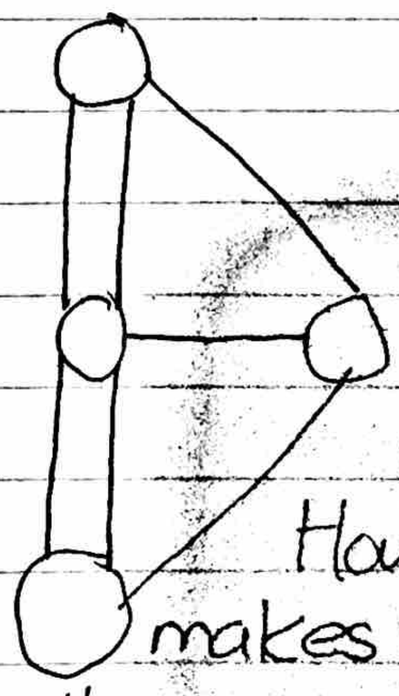
Count | 0 | 0 | 0 | 0 | 0 | 0 | → Count | 1 | 0 | 0 | 0 | 0 | 0 |
→ Count | 1 | 0 | 1 | 0 | 0 | 0 | → Count | 1 | 0 | 1 | 1 | 0 | 0 |
→ Count | 1 | 0 | 1 | 2 | 0 | 0 | → Count | 1 | 0 | 1 | 2 | 0 | 1 |

S | 60 | → S | 35 | 60 | → S | 35 | 81 |
→ S | 35 | 81 | 98 | → S | 14 | 81 | 98 | → S | 35 | 47 | 98 |

b. This algorithm is not stable since it would in most cases switch the order of two equal elements.

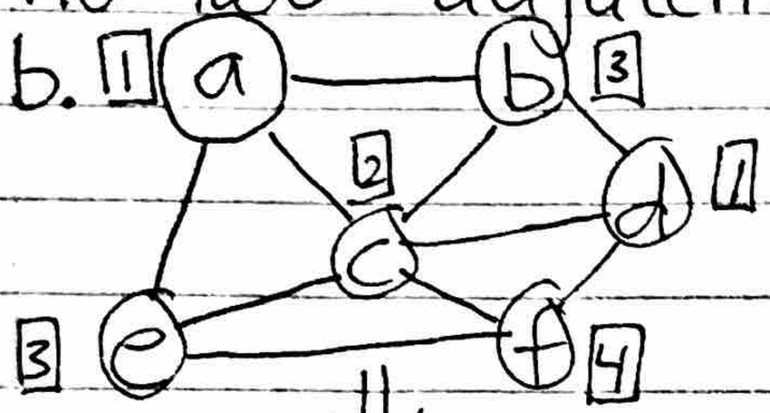
c. This algorithm is not in place since it uses additional memory through The Count and S arrays.

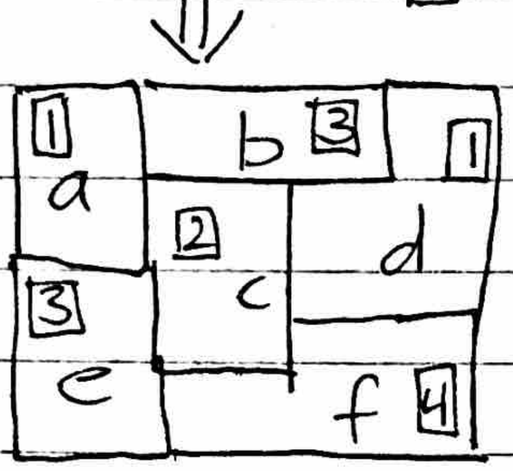
1.3-4: a.



b. There is no solution to this problem, since there are an odd number of bridges it is impossible to return to the same starting location without crossing a bridge twice. However adding just one bridge makes the number of bridges even and allows you to return to the starting point with only using each bridge once.

1.3-8: a. We can use the graph coloring problem to represent each region as a vertices so that no two adjacent vertices/regions have the same color.

b. 


1 = blue
2 = green
3 = red
4 = orange

2.1-1: a. (i) n (ii) addition (iii) no
b. (i) $n!$ (ii) multiplication (iii) no
c. (i) n (ii) comparing elements (iii) Yes
d. (i) m, n (ii) recursive loops (iii) No
e. (i) n (ii) removing multiples of i for $i: 0 \to n$ (iii) No
f. (i) $m \times n$ (ii) multiplying single digits (iii) No

2.1-7: a. $\frac{1}{3}(1000)^3 = 3.33 \times 10^8$ $\frac{1}{3}(500)^3 = 4.16 \times 10^7$
$3.33 \times 10^8 / 4.16 \times 10^7 = 8$ times longer
b. $1000 = \frac{1}{3}n^3 \Rightarrow n^3 = 3000 \Rightarrow n = \sqrt[3]{3000} = 14.4$ times bigger

2.1-9: a. same b. same c. lower

2.2-3: b. $\sqrt{10n^2+7n+3} = \sqrt{10}\,n + \cdots \Rightarrow \sqrt{10n^2+7n+3} \in \theta(n)$
c. $2n \lg(n+2)^2 + (n+2)^2 \lg\frac{n}{2} = (n^2+4n+4)\lg\frac{n}{2} + \cdots \Rightarrow \in n^2 \lg\frac{n}{2}$

2.2-6: a. Every polynomial of degree k, $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$ with $a_k > 0$ belongs to $\theta(n^k)$ because $a_k n^k$ is a constant multiple of $n^k$ and the remaining

terms, $a_{k-1}n^{k-1}+\cdots+a_0$, are all of lower order than $n^k$ and can therefore be disregarded.

b. Exponential functions $a^n$ have different orders of growth for different values of base $a>0$, for example for $n=2$ $5^2=25$ and $3^2=9$, while for $n=3$ $5^3=125$ and $3^3=27$. As you can see the rate of growth when $a=5$ is significantly greater than when $a=3$.

2.3-1: a. $\sum_{i=1}^{1000} i \approx \frac{1}{2}n^2 = 5\cdot 10^5 \Rightarrow 1+3+5+7+\cdots+999 \approx \frac{1}{2\cdot 2}n^2 = 2.5\cdot 10^5$

d. $\sum_{i=3}^{n+1} i = n-1+1-2 = n-2$

e. $\sum_{i=0}^{n-1} i(i+1) = \sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i \approx \frac{1}{3}n^3 - n^2 + \frac{1}{2}n^2 - i \approx \frac{1}{3}n^3 - \frac{1}{2}n^2 - i$

g. $\sum_{i=1}^{n}\sum_{j=1}^{n} ij = \sum_{i=1}^{n} i \cdot \sum_{j=1}^{n} j \approx \frac{1}{2}n^2 \cdot \frac{1}{2}n^2 = \frac{n^4}{4}$

2.3-2: a. $\sum_{i=0}^{n-1}(i^2+1)^2 = \sum_{i=0}^{n-1} i^4+2i+1 \approx \sum_{i=0}^{n-1} i^4 \approx \frac{n^5}{5} \in \theta(n^5)$

b. $\sum_{i=2}^{n-1} \lg i^2 = \sum_{i=2}^{n-1} 2\cdot \lg i = 2\sum_{i=2}^{n-1} \lg i \approx 2(n\lg n - \lg n) \in \theta(n\lg n)$

2.3-4: a. $\sum_{i=1}^{n} i^2$ b. $i*i$ c. n

d. linear e. cannot be improved the basic operation of the algorithm is i*i, which is essential in finding the sum of i*i from 1 to n.

2.3-10: 1000.

2.4-1: b. $x(n-1)=3(n-2) \Rightarrow x(n)=3[3x(n-1)]$

$\Rightarrow x(n)=3^i x(n-i) = 3^{n-1}x(n-(n-1)) = 3^{n-1}\cdot(4)$

d. $x(2^k)=x(2^{k-1})+2^k \Rightarrow x(2^{k-1})=x(2^{k-2})+2^{k+1}$

$x(2^k)=x(2^{k-i})+2^{k+i} \Rightarrow x(2^k)=x(2^{k-k})+2^{k+k}$

$\Rightarrow x(2^k)=1+2^{2k}$

S(n):

2.4-3: a. $S(n-1) + n*n*n \Rightarrow S(n-1) = S(n-2) + n*n*n$

$S(n) = S(n-2) + n*n*n + n*n*n \Rightarrow S(n) = s(n-i) + i \cdot n^3$

$S(n) = S(n-(n-1)) + (n-1)n^3 = (n-1)n^3 + 1$

b. for $i \leftarrow 1$ to $n$ do  $S \leftarrow 0$ (previously at top)

$S \leftarrow S + i*i*i$

return S

// nonrecursive algorithm has efficiency of $i^3 \cdot i$

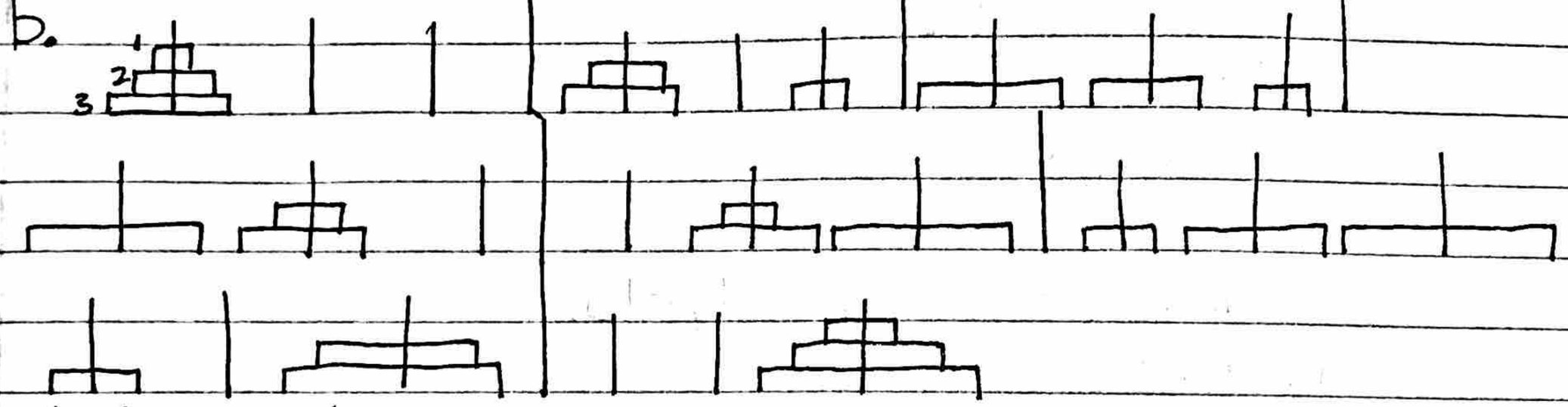
// While the recursive algorithm has efficiency of $(n-1)^3 n+1$

// so they are essentially the same.

2.4-5: a. $t(n) = 2^{64} - 1 \approx 2^{64} = 1.84 \times 10^{19}$ min / 527,040 min/year

$= 3.5 \times 10^{13}$ years

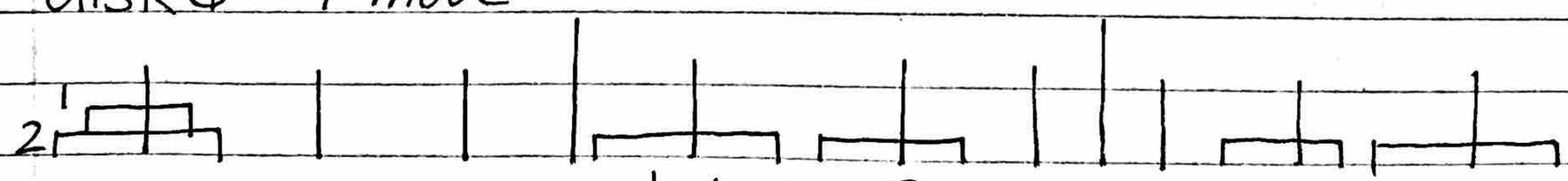
b.




disk 1 → 4 moves

disk 2 → 2 moves

disk 3 → 1 move




disk 1 → 2 moves

disk 2 → 1 move

disk i has n+1-i moves generally

c. #define MAXSTACK 100

struct details {

int num;

char start;

char dest;

char aux;

};

```
struct stack {
int top;
struct details item [MAXSTACK];
};
void push(struct stack *s, struct details *current){
if(s->top == MAXSTACK-1){
    cout<< "Overflow";
    exit(1); }
else
    s-> item [++(s->top)] = *current;
    return;}
void pop{test(struct *s, struct details *current, int *flag){
  if( s->top==-1){
    *flag=1;
    return;}
  *flg=0;
  *current = s->item[s->top-1];
  return;}
void towersofhanoi(int n, char startpeg, char destpeg, char
                                                    auxpeg){
struct stacks;
struct details current;
int flag;
char temp;
s.top=-1;
current.numb=n;
current.dest=destpeg;
current.start=startpeg;
current.aux=auxpeg;
while(1){
   while(current.numb!=1){
      push(&s, &current);
      --current.numb;
```

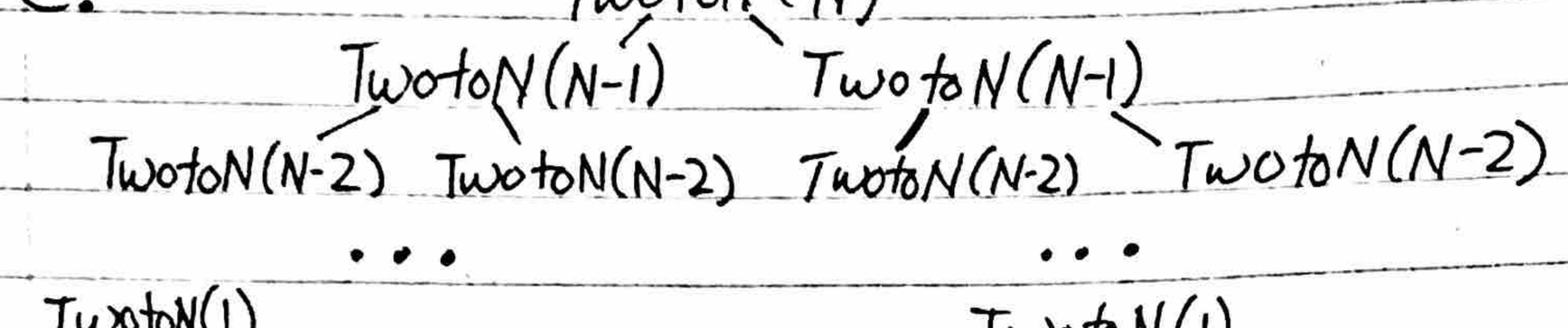
```
temp=current.dest;
current.dest = current.aux;
current.aux= temp; }
cout<< " Move Disc 1 from << current.start<<" to "
    <<current.dest <<endl;
pop&test(&s,&current,&flag);
if(flag==1)
   return;
cout<<"Move Disc" << current.numb <<" from" <<current.start
    <<" to " << current.dest <<endl;
--current.numb;
temp= current.start;
current.start = current.aux;
current.aux = temp; } }
int main(){
int N;
cout<<"Enter # of disks: ";
cin >> N;
towers of hanoi (N,'S','D','A');
return 0; }
```

2.4-8: a. int TwotoN(int N){
if(N==0){
return 1; }
else
return TwotoN(N-1)+TwotoN(N-1);
}

b. ~~A(n)=A(n-1)+1 ⇒ A(n-1)=A(n-2)+2 A(0)=0~~
~~A(n)=2A(n-1)+2 ⇒ A(n)=A(n-1)+n ⇒ A(n)=n~~
c. A(n)=2*A(n-1) A(n)=4A(n-2) $A(n)=2^iA(n-i) \Rightarrow A(n)=2^nA(0)$

Sum of $A=\sum_{i=1}^{n} A(i)=2^n+2^{n-1}+\cdots+2^1$

c.




The number of calls is $2^n + 2^{n-1} + \dots + 2^0$

d. This is not a good algorithm since it makes more recursions than necessary, instead of adding the values of two recursions you can simply multiply the recursive value by 2 to get the same result and make far less recursive calls.

2.6-1: For an array of size 2:
$i=1 \rightarrow j=0 \rightarrow$ if $A[j] <= v \rightarrow count=0$ ✗
Correction:

```
while j ≥ 0 do
    count ← count + 1
    if(A[j] > v)
        A[j+1] ← A[j]
    j ← j-1
```

2.6-2: a.

| array size | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|
| count | 499500 | 1999000 | 4498500 | 7998000 | 12497500 | 17997000 | 24496500 | 31996000 | 40495500 | 49995000 |

| arraysize | 11000 | 12000 | 13000 | 14000 | 15000 | 16000 | 17000 | 18000 | 19000 | 20000 |
|---|---|---|---|---|---|---|---|---|---|---|
| count | 60494500 | 71994000 | 84493500 | 97993000 | 112492500 | 127992000 | 144491500 | 161991000 | 180490500 | 199990000 |

b. $C(n) = \frac{n^2}{2} - \frac{n}{2}$

c. $C(25{,}000) = \frac{25000^2}{2} - \frac{25000}{2} = 312487500$

| 2.6-3: array size | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|
| run time(ms) | 1.203 | 4.703 | 10.515 | 18.546 | 29.078 | 41.812 | 57.031 | 73.640 | 93.530 | 115.046 |

| array size | 11000 | 12000 | 13000 | 14000 | 15000 | 16000 | 17000 | 18000 | 19000 | 20000 |
|---|---|---|---|---|---|---|---|---|---|---|
| run time(ms) | 141.40 | 166.40 | 195.31 | 227.03 | 260.46 | 303.1 | 342.1 | 376.5 | 429.6 | 478.1 |

$\frac{4.703}{1.203} = 3.91$ $\frac{10.515}{4.703} = 2.24$ $\frac{18.546}{10.515} = 1.76$ $\frac{29.078}{18.546} = 1.57$

As the size of the array increases the growth rate of run time decreases, so this algorithm appears to have a logarithmic efficiency class.

2.6-4: $\frac{24303}{11966} = 2.03$ $\frac{39992}{24303} = 1.65$ $\frac{53010}{39992} = 1.33$

Since the growth rate of this algorithm decreases with increasing size the efficiency class of this algorithm is likely logarithmic.