# CS 251 Written-HW1

Kalpkumar Shah

TOTAL POINTS

## 153 / 170

QUESTION 1

### 1 Problem 1 13 / 20

- **0 pts** Correct
- **2 pts** Doesn't handle n==0
- **3 pts** Proof by example
- **3 pts** Proved wrong function
- **3 pts** Error in program (will not terminate)

✓ **- 7 pts** **No proof**

- **5 pts** Proof unclear
- **20 pts** no submission/not honestly attempted
- **3 pts** Slight errors
- **8 pts** Attempted proof and method, but some errors
- **5 pts** Proof describes function but does not prove correctness
- **3 pts** Incorrect method
- **3 pts** Incorrect proof
- **3 pts** Used division
- **20 pts** Click here to replace this description.
- **13.4 pts** assignment wide deduction

QUESTION 2

### Problem 2 30 pts

#### 2.1 2.a 10 / 10

✓ **- 0 pts** **Correct**

- **2 pts** 1 mistake
- **6 pts** 3 mistakes

#### 2.2 2.b 10 / 10

✓ **- 0 pts** **Correct**

- **5 pts** Partial
- **10 pts** incorrect

#### 2.3 2.c 10 / 10

✓ **- 0 pts** **All Correct**

- **5 pts** Tried or Partially correct
- **10 pts** incorrect

QUESTION 3

### Problem 3 30 pts

#### 3.1 3.a 15 / 15

✓ **- 0 pts** **Correct**

- **3 pts** Returns true if values > n are included
- **3 pts** Doesn't return false if duplicates found
- **6 pts** Major issues with function
- **15 pts** No submission.
- **3 pts** Didn't dynamically size array
- **3 pts** Summed to wrong value
- **3 pts** a[i] == a[j] returns false, and i can equal j

#### 3.2 3.b 15 / 15

✓ **- 0 pts** **Correct**

- **3 pts** Does not check whether the given array is a valid permutation
- **3 pts** Does not return integer pointer
- **7.5 pts** Significant error, but honest effort
- **15 pts** Incorrect

QUESTION 4

### 4 Problem 4 60 / 60

✓ **+ 60 pts** **Correct**

Function - A

- **+ 7 pts** All Correct
- **+ 3.5 pts** Runtime correct
- **+ 3.5 pts** Explanation Correct

Function - B

- **+ 7 pts** All Correct
- **+ 3.5 pts** Runtime correct
- **+ 3.5 pts** Explanation Correct

Function - C

+ **8 pts** All correct
+ **4 pts** Runtime correct
+ **4 pts** Explanation Correct

Function - D

+ **8 pts** All Correct
+ **4 pts** Runtime correct
+ **4 pts** Explanation Correct

Function - E

+ **10 pts** All Correct
+ **5 pts** Runtime correct
+ **5 pts** Explanation Correct

Function - F

+ **10 pts** All Correct
+ **5 pts** Runtime correct
+ **5 pts** Explanation Correct

Function - G

+ **10 pts** All Correct
+ **5 pts** Runtime correct
+ **5 pts** Explanation Correct

+ **0 pts** All incorrect
+ **0 pts** Click here to replace this description.

QUESTION 5

## 5 Problem 5 20 / 30

- **0 pts** Correct
- **10 pts** Both answers incorrect, but work shown
✓ - **5 pts** **2nd answer incorrect**
- **10 pts** Second question not completed
- **15 pts** Missing work and one question incorrect
- **15 pts** Second question not attempted and first question incorrect
- **5 pts** First question incorrect
- **30 pts** Click here to replace this description.
- **25 pts** did not directly answer question
- **5 pts** serious logical issues
✓ - **5 pts** **explanation lacks details**
- **30 pts** question not attempted
- **20 pts** Submission unclear
- **15 pts** No work shown, but appeared to get right answer
- **25 pts** No correct answers and no work shown, but problem appears to be attempted

**CS-251, Summer 2021**
**Written Homework 1**
**Due: Monday, July 12 by 11:59PM**

**NO LATE SUBMISSIONS FOR THIS ASSIGNMENT!**

**Submission will be done using gradescope (you will submit your homework as a pdf, either scanned or exported from a word processor).**

- **Your writeup must be neat and clear**
- **There are 5 problems, some with multiple parts; clearly label your answers.**

**Total possible points: 170 (each problem labeled with possible points).**

---

**PROBLEM 1 - Thinking Recursively (20 points):** Consider a finite set $S$ with $n$ elements; the number of distinct subsets of $S$ with exactly two is called "n choose 2" and typically written as $\frac{n}{2}$. You may Recall that $\frac{n}{2} = \frac{n(n-1)}{2}$.

Below is a (trivial) C++ function which takes a non-negative integer $n$ and returns $\frac{n}{2}$ (also a non-negative integer):

```
unsigned int n_choose_2(unsigned int n) {

    if(n==0)
       return 0;
    else
       return n*(n-1)/2;
}
```

Your job: write a function which also returns $\frac{n}{2}$ but with the following constraints:

- You cannot use the multiplication operator '*'
- You cannot use the division operator '/'
- You cannot have any loops
- You cannot add any additional parameters to the function
- Your function must be self-contained: no helper functions!
- You cannot use any globals
- You cannot use any static variables
- You cannot use any "bit twiddling" operations -- no shifts, etc.

However, …

- You *can* use recursion
- You *can* use the '+' and '-' operators.

You are free to try out your solution in a real program, but just submit a printout of your function.

In addition: argument of correctness required!

You must explain the logic of your solution! In other words, explain *why* it works. Think in terms of an inductive argument.

Just giving a correct C++ function is not sufficient and will not receive many points (possibly zero!)

Another note: an example is not an argument of correctness!

**Answer:**

```
unsigned int n_choose_2(unsigned int n) {
        if(n==0)
                return 0;
        return (n-1) + n_choose_2(n-1);
}
```

**PROBLEM 2 (30 points):** Consider the recursive C++ function below:

```
void foo(unsigned int n) {

    if(n==0)
        cout << "tick" << endl;
    else {
        foo(n-1);
        foo(n-1);
        foo(n-1);
    }
}
```

**2.A:** Complete the following table indicating how many "ticks" are printed for various parameters n.

Unenforceable rule: derive your answers "by hand" -- not simply by writing a program calling the function.

| n | number of ticks printed when foo(n) is called |
|---|---|
| 0 | 1 |
| 1 | 3 |
| 2 | 9 |
| 3 | 27 |
| 4 | 81 |

## 1 Problem 1 13 / 20

- **0 pts** Correct
- **2 pts** Doesn't handle n==0
- **3 pts** Proof by example
- **3 pts** Proved wrong function
- **3 pts** Error in program (will not terminate)

✓ **- 7 pts** **No proof**

- **5 pts** Proof unclear
- **20 pts** no submission/not honestly attempted
- **3 pts** Slight errors
- **8 pts** Attempted proof and method, but some errors
- **5 pts** Proof describes function but does not prove correctness
- **3 pts** Incorrect method
- **3 pts** Incorrect proof
- **3 pts** Used division
- **20 pts** Click here to replace this description.
- **13.4 pts** assignment wide deduction

**Answer:**

```
unsigned int n_choose_2(unsigned int n) {
        if(n==0)
                return 0;
        return (n-1) + n_choose_2(n-1);
}
```

**PROBLEM 2 (30 points):** Consider the recursive C++ function below:

```
void foo(unsigned int n) {

    if(n==0)
        cout << "tick" << endl;
    else {
        foo(n-1);
        foo(n-1);
        foo(n-1);
    }
}
```

**2.A:** Complete the following table indicating how many "ticks" are printed for various parameters n.

Unenforceable rule: derive your answers "by hand" -- not simply by writing a program calling the function.

| n | number of ticks printed when foo(n) is called |
|---|---|
| 0 | 1 |
| 1 | 3 |
| 2 | 9 |
| 3 | 27 |
| 4 | 81 |

2.1 2.a **10 / 10**

✓ **- 0 pts** **Correct**

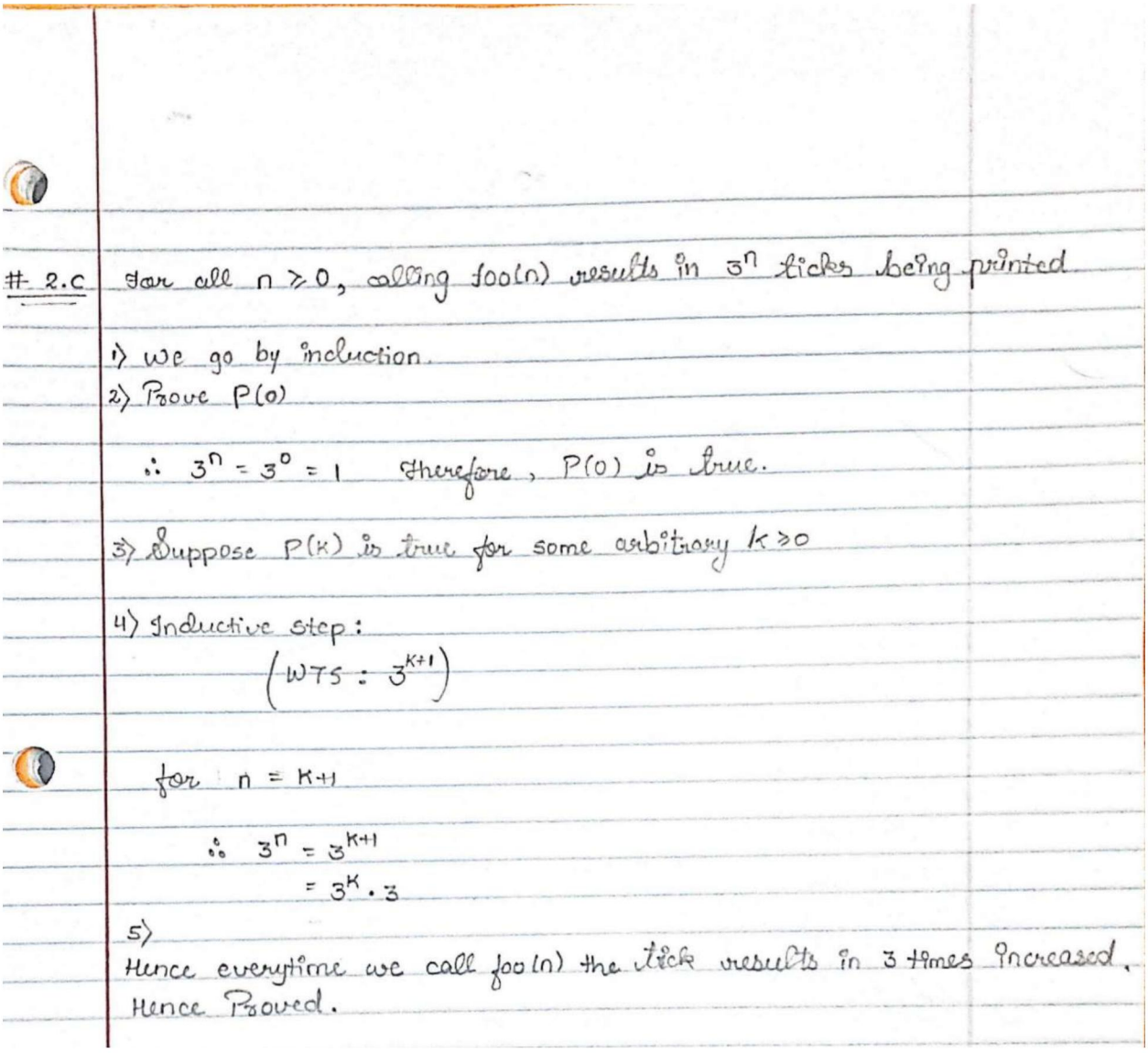
**- 2 pts** 1 mistake

**- 6 pts** 3 mistakes

**2.B:** Derive a conjecture expressing the number of ticks as a function of n -- i.e., complete the following:

> *"Conjecture: for all $n \geq 0$, calling foo(n) results in $3^n$ ticks being printed"*

**2.C:** Prove your conjecture from part B (hint: Induction!)



#2.C For all $n \geq 0$, calling foo(n) results in $3^n$ ticks being printed

1) We go by induction.

2) Prove P(0)

$\therefore 3^n = 3^0 = 1$ Therefore, P(0) is true.

3) Suppose P(K) is true for some arbitrary $K \geq 0$

4) Inductive step:

(WTS : $3^{K+1}$)

for $n = K+1$

$\therefore 3^n = 3^{K+1}$

$= 3^K \cdot 3$

5)

Hence everytime we call foo(n) the tick results in 3 times Increased.

Hence Proved.

### 2.2 2.b **10 / 10**

✓ **- 0 pts** **Correct**

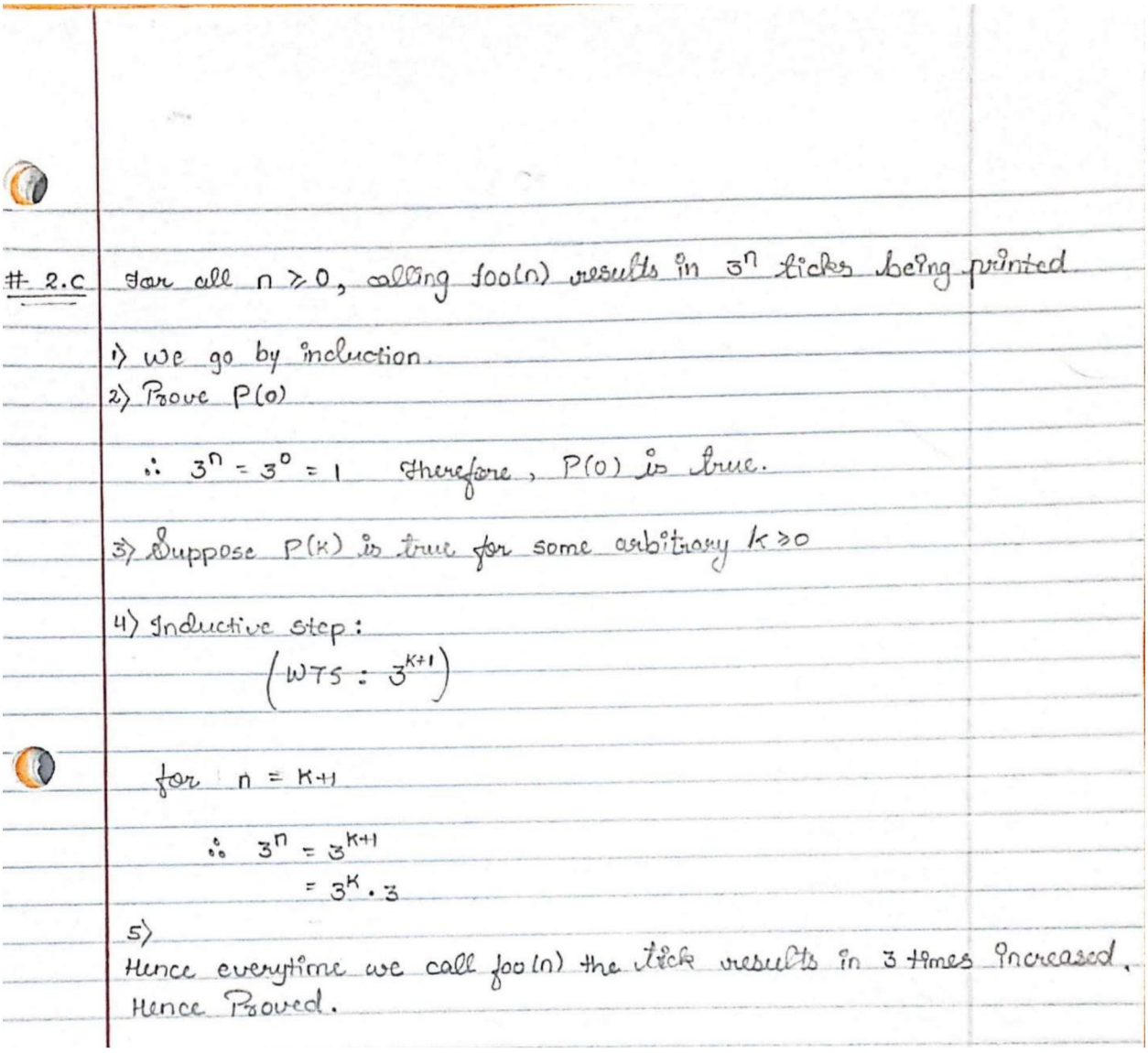
**- 5 pts** Partial

**- 10 pts** incorrect

**2.B:** Derive a conjecture expressing the number of ticks as a function of n -- i.e., complete the following:

> *"Conjecture: for all $n \geq 0$, calling foo(n) results in $3^n$ ticks being printed"*

**2.C:** Prove your conjecture from part B (hint: Induction!)



#2.C For all $n \geq 0$, calling foo(n) results in $3^n$ ticks being printed

1) We go by induction.

2) Prove P(0)

$\therefore 3^n = 3^0 = 1$ Therefore, P(0) is true.

3) Suppose P(K) is true for some arbitrary $K \geq 0$

4) Inductive step:

(WTS : $3^{K+1}$)

for $n = K+1$

$\therefore 3^n = 3^{K+1}$

$= 3^K \cdot 3$

5)

Hence everytime we call foo(n) the tick results in 3 times Increased.

Hence Proved.

2.3 2.c **10 / 10**

✓ **- 0 pts All Correct**

**- 5 pts** Tried or Partially correct

**- 10 pts** incorrect

**PROBLEM 3 (30 points)**: Short description:

A. Write a function which determines if a given array of n integers is a *permutation* of {0..n-1}.
B. Write a function which takes a permutation of n integers and constructs its *inverse*.

Background: recall that a permutation of a set of integers is just an ordered listing of the elements. For example, consider the set $S = \{0,1,2\}$.

| example permutations of $S$ | These are *not* permutations of $S$ |
|---|---|
| [0, 1, 2] | [5, 0, 1] |
| [1, 2, 0] | [0, 1, 0] |
| [2, 0, 1] | [2, 1, 1] |

Suppose an array of $n$ integers is indeed a permutation of $\{0..n-1\}$. We can view such an array as a *bijective function* $f$ from **POSITION-to-ELEMENT-AT-INDEX**:

$$f: \{0..n-1\} \to \{0..n-1\}.$$

Since $f$ is a bijection, it also has an *inverse* which is a function from **ELEMENT-to-POSITION-OF-ELEMENT**:

$$f^{-1}: \{0..n-1\} \to \{0..n-1\}$$

Examples of permutations of {0,1,2} and their inverses as array diagrams:

| **Permutation** | **Inverse** |
|---|---|
| permutation: <1, 2, 0><br>array representation:<br>val: 1, 2, 0<br>idx: 0, 1, 2 | val: 2, 0, 1<br>idx: 0, 1, 2<br>example: 2 appears at position 1 in permutation. relevant entries highlighted |
| permutation: <2, 1, 0><br>array representation:<br>val: 2, 1, 0<br>idx: 0, 1, 2 | val: 2, 1, 0<br>idx: 0, 1, 2 |

permutation: <2, 0, 1>
array representation:

| val | 2 | 0 | 1 |
|---|---|---|---|
| idx | 0 | 1 | 2 |

| val | 1 | 2 | 0 |
|---|---|---|---|
| idx | 0 | 1 | 2 |

Finally, your tasks:

**A.** Write a C++ function **is_perm** which determines if a given array of n integers is a *permutation* of {0..n-1}. **COMMENT**: There is no runtime requirement for this function, however FYI it is possible to solve the problem in linear time!

**B.** Write a function which takes a permutation as an array of n integers and constructs its *inverse* (also as an array). Your function will allocate storage for the inverse array and returns it as an integer pointer. If the array it receives is not a permutation of {0..n-1}, it returns nullptr instead.

Templates given below.

```
bool is_perm(int a[], int n) {

      int find[n];
      for(int i = 0; i < n; i++)

      {

           if(a[i]<0 || a[i]>n-1)

               return false;

           if(find[a[i]] == 1)

               return false;

           else

               find[a[i]] = 1;

      }

      return true;

}
```

### 3.1 3.a **15 / 15**

✓ **- 0 pts** **Correct**

**- 3 pts** Returns true if values > n are included

**- 3 pts** Doesn't return false if duplicates found

**- 6 pts** Major issues with function

**- 15 pts** No submission.

**- 3 pts** Didn't dynamically size array

**- 3 pts** Summed to wrong value

**- 3 pts** a[i] == a[j] returns false, and i can equal j

```
int * perm2inverse(int a[], int n) {

    if(!is_perm(a, n))  // to get you started...
           return nullptr;

    // otherwise, a[] is a permutation of 0..n-1; allocate and populate its
    //    inverse array (and return it)

    int* arrInverse = new int[n];
    for(int i = 0; i < n; i++)
    {
         arrInverse[a[i]] = i;
    }
    return arrInverse;
}
```

**PROBLEM 4 (60 pts.):** For each of the functions below, give as tight a ***worst-case*** runtime bound as you can. Express your answers with Big-O / Big-Theta notation.
Since you are seeking "tight" bounds, your answers should be expressed using Big-*Θ* -- unless you are especially candid and aren't sure if your upper-bound is indeed tight, you might say something like "I know the worst case is O(<something>), but I have not been able to show that this is a tight bound..."

- Show your reasoning and
- express your answers in the ***SIMPLEST TERMS POSSIBLE!!***

```
// 7 points
int A(int a[], int n) {
int x=0, i, j;
  for(i=0; i<n; i++) {
     x += a[i]%2;
  }
  for(i=0; i<n; i++){
     for(j=0; j<i; j++)
       x += a[j];
  }
  return x;
}
```

Here, the first loop is of $O(n)$ and for the second loop the runtime is of $O(n^2)$. Hence the final runtime is $O(n) + O(n^2) = O(n^2)$.

### 3.2 3.b **15 / 15**

✓ **- 0 pts** **Correct**

**- 3 pts** Does not check whether the given array is a valid permutation

**- 3 pts** Does not return integer pointer

**- 7.5 pts** Significant error, but honest effort

**- 15 pts** Incorrect

```
int * perm2inverse(int a[], int n) {

    if(!is_perm(a, n))  // to get you started...
           return nullptr;

    // otherwise, a[] is a permutation of 0..n-1; allocate and populate its
    //    inverse array (and return it)

    int* arrInverse = new int[n];
    for(int i = 0; i < n; i++)
    {
         arrInverse[a[i]] = i;
    }
    return arrInverse;
}
```

**PROBLEM 4 (60 pts.):** For each of the functions below, give as tight a ***worst-case*** runtime bound as you can. Express your answers with Big-O / Big-Theta notation.
Since you are seeking "tight" bounds, your answers should be expressed using Big-*Θ* -- unless you are especially candid and aren't sure if your upper-bound is indeed tight, you might say something like "I know the worst case is O(<something>), but I have not been able to show that this is a tight bound..."

- Show your reasoning and
- express your answers in the ***SIMPLEST TERMS POSSIBLE!!***

```
// 7 points
int A(int a[], int n) {
int x=0, i, j;
  for(i=0; i<n; i++) {
     x += a[i]%2;
  }
  for(i=0; i<n; i++){
     for(j=0; j<i; j++)
       x += a[j];
  }
  return x;
}
```

Here, the first loop is of $O(n)$ and for the second loop the runtime is of $O(n^2)$. Hence the final runtime is $O(n) + O(n^2) = O(n^2)$.

```
// 7 points
int B(int a[], int n) {
int x=0, k;

  for(k=1; k<n; k = k*2)
     x += a[k];
  return x;
}
```

For this, the k must loop till n and then it is doubled than the original one. Hence it runs on the runtime of $O(\log_2 n)$.

```
// 8 points
int C(int a[], int n) {
int x=0, i, j, k;

  for(i=1; i<n; i = i*2){
     for(j=1; j<n; j=j*2){
        x += (a[j]-a[i]);
     }
  }
  k = n;
  while(k>0) {
        x += a[k];
        k = k/2;
  }
  return x;
}
```

Here the first loop is nested and runs doubled so the runtime is $O((\log_2 n)^2)$. While the second is just one loop but the k is divided by 2 so it runs by $O(\log n)$. Hence the final function runs on $O((\log_2 n)^2) + O(\log_2 n) = O((\log_2 n)^2)$.

```
// 8 points
int D(int a[], int n) {
int x=0, i, j;

  for(i=0; i<n; i++){
     if(i%2 == 0){
```

```
        for(j=0; j<n; j++)
           x += a[j];
     }
     else
        x--;
  }
  return x;
}
```

Here the first loop runs n times so the runtime complexity is O(n). Then for the second loop the condition matters. If the number is even, then the second loop is executed so the runtime becomes $O(n^2)$. If the number is odd, then the runtime complexity will remain the same which is O(n). But at the end the runtime complexity will be $O(n^2)$.

```
// 10 points

int E(int a[], int n) { // tricky!
int x=0, i, j;

  for(i=0; i<n*n; i++){
      if(i%n == 0)
           for(j=0; j<n; j++)
               x += a[j];
      else
          x--;
  }
  return x;
}
```

Here the first loop executes for the $n^2$ times. Then it is again divided by the n, so it becomes n times and then the second loop is run for n times. Hence the overall runtime complexity becomes n*n = $O(n^2)$.

```
// 10 points

int F(int a[], int n) {
int x=0, i, j;

  for(i=1; i<n; i=i*2){
```

```
    for(j=0; j<n; j++)
       x += a[j];
  }
  return x;
}
```

Here the first loop runs for the runtime complexity of O(logn) and then the inside loop runs for the runtime complexity of O(n) times. Hence the final runtime time is O(logn)*O(n) = O(nlogn).

```
// 10 points

int G(int a[], int n) {
int x=0, i, j;

  for(i=1; i<n; i=i*2){
     for(j=0; j<i; j++)
        x += a[j];
  }
  return x;
}
```

Here the first loop doubles the value each time we call the loop, so the runtime complexity becomes $O(2^{\log n})$ and then the second loop is based on the first condition. Hence the final runtime complexity is O(n).

**PROBLEM 5 (30 pts.):**

Analyze the runtime of C functions below and give a tight runtime bound for each.

- Both functions have the same best-case and worst-case runtime (so this is not an issue).
- Since we want a "tight" runtime bound, your final answer should be in big-$\theta$ form.
- Show your work! "The runtime of foo() is $\theta(< something >)$" is not sufficient even if <something> happens to be correct. In other words, convince the reader of the correctness of your answer.

## 4 Problem 4 60 / 60

✓ **+ 60 pts Correct**

Function - A

+ **7 pts** All Correct
+ **3.5 pts** Runtime correct
+ **3.5 pts** Explanation Correct

Function - B

+ **7 pts** All Correct
+ **3.5 pts** Runtime correct
+ **3.5 pts** Explanation Correct

Function - C

+ **8 pts** All correct
+ **4 pts** Runtime correct
+ **4 pts** Explanation Correct

Function - D

+ **8 pts** All Correct
+ **4 pts** Runtime correct
+ **4 pts** Explanation Correct

Function - E

+ **10 pts** All Correct
+ **5 pts** Runtime correct
+ **5 pts** Explanation Correct

Function - F

+ **10 pts** All Correct
+ **5 pts** Runtime correct
+ **5 pts** Explanation Correct

Function - G

+ **10 pts** All Correct
+ **5 pts** Runtime correct
+ **5 pts** Explanation Correct

+ **0 pts** All incorrect
+ **0 pts** Click here to replace this description.

```
    for(j=0; j<n; j++)
       x += a[j];
  }
  return x;
}
```

Here the first loop runs for the runtime complexity of O(logn) and then the inside loop runs for the runtime complexity of O(n) times. Hence the final runtime time is O(logn)*O(n) = O(nlogn).

```
// 10 points

int G(int a[], int n) {
int x=0, i, j;

  for(i=1; i<n; i=i*2){
     for(j=0; j<i; j++)
        x += a[j];
  }
  return x;
}
```

Here the first loop doubles the value each time we call the loop, so the runtime complexity becomes $O(2^{\log n})$ and then the second loop is based on the first condition. Hence the final runtime complexity is O(n).

**PROBLEM 5 (30 pts.):**

Analyze the runtime of C functions below and give a tight runtime bound for each.

- Both functions have the same best-case and worst-case runtime (so this is not an issue).
- Since we want a "tight" runtime bound, your final answer should be in big-$\theta$ form.
- Show your work! "The runtime of foo() is $\theta(< something >)$" is not sufficient even if <something> happens to be correct. In other words, convince the reader of the correctness of your answer.

```
int foo(int n) {
int i, j, limit, x;

  limit = 16;
  x = 0;

  for(i=0; i<n; i++) {
    if(i==limit) {
        for(j=0; j<limit; j++) {
             x++;
        }
        limit = limit * 2;
    }
  }
  return x;
}
```

Here the first loop has a runtime complexity of n times. When the limit is reached from the first loop the second loop is executed hence it runs for n-1 times. Hence the final runtime is $\Theta(n)$.

```
int bar(int n) {
int i, j, limit, x;

  limit = 16;
  x = 0;

  for(i=0; i<n; i++) {
    if(i==limit) {
        for(j=0; j<limit; j++) {
             x++;
        }
        limit = limit + 8;
    }
  }
  return x;
}
```

Here the first loop has a runtime complexity of n times. When the limit is reached from the first loop the second loop is executed with the increasing of the size of limit by 8 times hence it runs for n-1 times. Hence the final runtime is $\Theta(n)$.

## 5 Problem 5 20 / 30

- **0 pts** Correct
- **10 pts** Both answers incorrect, but work shown
✓ **- 5 pts** **2nd answer incorrect**
- **10 pts** Second question not completed
- **15 pts** Missing work and one question incorrect
- **15 pts** Second question not attempted and first question incorrect
- **5 pts** First question incorrect
- **30 pts** Click here to replace this description.
- **25 pts** did not directly answer question
- **5 pts** serious logical issues
✓ **- 5 pts** **explanation lacks details**
- **30 pts** question not attempted
- **20 pts** Submission unclear
- **15 pts** No work shown, but appeared to get right answer
- **25 pts** No correct answers and no work shown, but problem appears to be attempted