CS 251 Written-HW2

Kalpkumar Shah

TOTAL POINTS

118 / 160

QUESTION 1

1 Problem 1 10 / 20

- **0 pts** Correct: $$\Theta(n)$$ and correct explanations
- **3 pts** minor mistake
- **7 pts** $$\Theta(n)$$ incorrect explanation
- ✓ **- 10 pts** **$$\Theta(n^2)$$ because of last for loop**
- **20 pts** Incorrect or no submission
- **20 pts** Click here to replace this description.

QUESTION 2

2 Problem 2 40 / 50

- **0 pts** Correct
- **10 pts** Incorrect answer, didn't sort start and end times independently
- **10 pts** Didn't explicitly mention sorting, but algorithm correct otherwise.
- **15 pts** Solution O(N^2)
- **10 pts** Incorrect answer, increments count for all overlaps, not just simultaneous overlaps
- ✓ **- 10 pts** **Missing details**
- **25 pts** Algorithm does not explicitly solve the problem.
- **50 pts** No submission
- **5 pts** Reset counter instead of decrementing it when reaching end of range
- **10 pts** Answer will always be 0
- **15 pts** Code is O(nlogn) but would not produce a correct output
- **5 pts** Times are floating point numbers
- **10 pts** Misunderstanding of the definition of overlap
- **17.4 pts** Assignment wide deduction

1 What is done within the loop? This is the essential part of the algorithm. Minor deduction...

QUESTION 3

3 Problem 3 15 / 20

- **0 pts** Correct (d = 2.5 approximately)
- ✓ **- 5 pts** **Almost correct with honest effort or significant mistake with significant honest effort**
- **10 pts** Significant mistake, but some honest effort
- **20 pts** Incorrect / no effort
- **20 pts** Click here to replace this description.

QUESTION 4

Problem 4 40 pts

4.1 4.A 17 / 20

- **0 pts** Correct
- **15 pts** Incorrect or No explanation
- ✓ **- 3 pts** **Minor mistake**
- **20 pts** Click here to replace this description.

4.2 4.B 17 / 20

- **0 pts** Correct
- ✓ **- 3 pts** **Minor mistake**
- **10 pts** Incorrect Logic
- **20 pts** Click here to replace this description.
- **15 pts** Click here to replace this description.

QUESTION 5

Problem 5 30 pts

5.1 Part-I 9 / 10

- **0 pts** Correct
- ✓ **- 1 pts** **doesn't handle "high < low" case correctly**
- **1 pts** Minor bug in program
- **2 pts** 2 minor bugs
- **2 pts** Solution is not recursive
- **3 pts** Algorithm is not complete
- **10 pts** problem not submitted

- **1 pts** Incorrect value for "mid" (should be (low+high)/2 or equivalent)
- **1 pts** Copies entire arrays which increases runtime complexity.
- **10 pts** Click here to replace this description.

5.2 Part-II 0 / 10

- **0 pts** Correct
- **5 pts** Justification missing or some mistake

✓ **- 10 pts Incorrect or no answer**

- **10 pts** Click here to replace this description.

5.3 Part-III 10 / 10

✓ **- 0 pts Correct**

- **10 pts** Did not attempt
- **2 pts** Correct tight bound, but no explanation
- **2 pts** Attempted to analyze non-recursive function
- **5 pts** does not display honest effort
- **1 pts** Analysis good but no tight runtime given.
- **1 pts** Recursion tree given, but it does not demonstrate runtime clearly
- **2 pts** error in analysis leads to wrong runtime
- **1 pts** Proof by example
- **2 pts** Answer is right for algorithm given (which is incorrect), but wrong for recurrence relation given (which is correct).
- **1 pts** incorrect use of master theorem, but reached correct answer
- **1 pts** Correct answer for wrong algorithm and wrong recurrence relation, but not correct answer for correct algorithm
- **1 pts** Correct answer, but analysis is incorrect
- **10 pts** Click here to replace this description.

CS251 - Written HW2
Due: TBD

NO LATE SUBMISSIONS ACCEPTED FOR THIS ASSIGNMENT!

A SAMPLE SOLUTION WILL BE POSTED AFTER THE SUBMISSION DEADLINE

Submission: will be through gradescope
TOTAL POINTS: 160

NOTE: This problem set includes "algorithm design" problems. An appendix has been included at the end of this handout to help you in writing clear algorithm descriptions (which are not the same as "programs").

Correction (7/14): path to executable in Problem 3 has been corrected.

PROBLEM 1 (20 pts.):
The function below distributes jelly beans to n children.

Analyze the function and a tight worst-case runtime bound for it (you are looking for a big-*O* bound).

Show your work / provide a rationale

```
void jelly(int a[], int n) {
int i, j;
int beans=10*n;

  // all children start with
  //   zero beans.
  for(i=0; i<n; i++)
    a[i]=0;

  // hand out beans one by one
  //   to a random child
  while(beans > 0){
    i = rand() % n;
    a[i]++;
    beans--;
  }
  // print one line of beans
  //   for each child
  for(i=0; i<n; i++) {
    cout << "CHILD " << i << ": ";
    for(j=0; j<a[i]; j++)
      cout << "bean! ";
    cout << "\n";
  }
}
```

Here, the first loop runs for n times so the runtime is O(n). Now, the second loop runs for the 10*n as the beans are 10*n, so the runtime is O(10n). The final loop runs for n times and then the inner loop runs for n-1 time so in total the loop runs for O(n^2). Hence the final runtime is O(n^2).

1 Problem 1 10 / 20

- **0 pts** Correct: $$\Theta(n)$$ and correct explanations
- **3 pts** minor mistake
- **7 pts** $$\Theta(n)$$ incorrect explanation

✓ **- 10 pts** **$$\Theta(n^2)$$ because of last for loop**

- **20 pts** Incorrect or no submission
- **20 pts** Click here to replace this description.

PROBLEM 2: algorithm design (50 points):
You are given as set of n time intervals $\{(s_1, e_1), (s_2, e_2), \ldots, (s_n, e_n\})$. The interval (s_i, e_i) has a start time of s_i and an ending time of e_i.

You can think of each interval indicating that a "process" is active during that time. You may assume that $e_i > s_i$. Or you can just think of them as horizontal line segments.

You want to find the maximum overlap among all of the given time intervals. Your algorithm simply determines this maximum value and reports it.

The diagram below illustrates an instance of the problem with 6 time intervals, each represented by horizontal line segments.

For this instance, the algorithm would report 4 as the maximum overlap.

Devise an algorithm solving this problem in $O(n \log n)$ time.

Discussion/details:

- Time intervals are *inclusive* of their end points. As a result, if one interval ends at exactly the same time another begins, the two intervals are overlapping.
- The interval start and end times are given as floating point numbers.
- The intervals are given in an arbitrary order
- If your approach includes some kind of sorting operation, you can assume the existence of, for example, MergeSort (and its runtime). Exactly ***what*** you choose to sort and ***how*** you interpret the results are things you must explain and justify.

Answer: So we go by declaring two arrays where the arrTimes[] is an array containing the arrival times and endTimes[] is an array containing the ending times of all the processes.
We declare a function which finds the maximum, so we start sorting the arrays using mergeSort. Then we find the time for finding the max in the variable time. We start with a loop which loops for n times so the runtime for it is O(n).
Now, the mergeSort takes O(nlogn) time and the loop we declared in the function runs for O(n) times, so the final runtime is O(nlogn) + O(n) = O(nlogn).

PROBLEM 3 (20 points): (estimation of asymptotic runtime from experiments)

Log into ~~bert or~~ bertvm (or ernievm, systems[1-4]).

In the directory ~~~jlillis/CS251-public~~ **~lillis/CS251-public** you will find an executable called mystery (but no source code!).

The program takes a single command line argument N ("problem size").

When you run the program it does some mysterious computations and

2 Problem 2 40 / 50

- **0 pts** Correct
- **10 pts** Incorrect answer, didn't sort start and end times independently
- **10 pts** Didn't explicitly mention sorting, but algorithm correct otherwise.
- **15 pts** Solution O(N^2)
- **10 pts** Incorrect answer, increments count for all overlaps, not just simultaneous overlaps

✓ - **10 pts** **Missing details**

- **25 pts** Algorithm does not explicitly solve the problem.
- **50 pts** No submission
- **5 pts** Reset counter instead of decrementing it when reaching end of range
- **10 pts** Answer will always be 0
- **15 pts** Code is O(nlogn) but would not produce a correct output
- **5 pts** Times are floating point numbers
- **10 pts** Misunderstanding of the definition of overlap
- **17.4 pts** Assignment wide deduction

1 What is done within the loop? This is the essential part of the algorithm. Minor deduction...

PROBLEM 2: algorithm design (50 points):
You are given as set of n time intervals $\{(s_1, e_1), (s_2, e_2), \ldots, (s_n, e_n\})$. The interval (s_i, e_i) has a start time of s_i and an ending time of e_i.

You can think of each interval indicating that a "process" is active during that time. You may assume that $e_i > s_i$. Or you can just think of them as horizontal line segments.

You want to find the maximum overlap among all of the given time intervals. Your algorithm simply determines this maximum value and reports it.

The diagram below illustrates an instance of the problem with 6 time intervals, each represented by horizontal line segments.

For this instance, the algorithm would report 4 as the maximum overlap.

Devise an algorithm solving this problem in $O(n \log n)$ time.

Discussion/details:

- Time intervals are *inclusive* of their end points. As a result, if one interval ends at exactly the same time another begins, the two intervals are overlapping.
- The interval start and end times are given as floating point numbers.
- The intervals are given in an arbitrary order
- If your approach includes some kind of sorting operation, you can assume the existence of, for example, MergeSort (and its runtime). Exactly ***what*** you choose to sort and ***how*** you interpret the results are things you must explain and justify.

Answer: So we go by declaring two arrays where the arrTimes[] is an array containing the arrival times and endTimes[] is an array containing the ending times of all the processes.
We declare a function which finds the maximum, so we start sorting the arrays using mergeSort. Then we find the time for finding the max in the variable time. We start with a loop which loops for n times so the runtime for it is O(n).
Now, the mergeSort takes O(nlogn) time and the loop we declared in the function runs for O(n) times, so the final runtime is O(nlogn) + O(n) = O(nlogn).

PROBLEM 3 (20 points): (estimation of asymptotic runtime from experiments)

Log into ~~bert or~~ bertvm (or ernievm, systems[1-4]).

In the directory ~~~jlillis/CS251-public~~ **~lillis/CS251-public** you will find an executable called mystery (but no source code!).

The program takes a single command line argument N ("problem size").

When you run the program it does some mysterious computations and

eventually terminates. When it terminates it prints out the (approximate) CPU time taken by the run in seconds.

For example if you do this:

~lillis/CS251-public/mystery 2000

it will run with N=2000 and then report the elapsed CPU time in seconds.

Your job: do some experiments by running the mystery program for a range of values of N and try to come up with a conjecture on the asymptotic runtime of the program as a function of N by doing some analysis.

Assumption/hint: the actual runtime has the form $\Theta(N^d)$ for some d. You are trying to figure out what d is.

SHOW YOUR WORK: describe the experiments you did and how you arrived at your final conclusion.

```
kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 100
n: 100 ; cpu elapsed time (sec) : 0.005

kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 200
n: 200 ; cpu elapsed time (sec) : 0.029

kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 300
n: 300 ; cpu elapsed time (sec) : 0.079

kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 400
n: 400 ; cpu elapsed time (sec) : 0.146

kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 500
n: 500 ; cpu elapsed time (sec) : 0.243

kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 600
n: 600 ; cpu elapsed time (sec) : 0.383

kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 700
n: 700 ; cpu elapsed time (sec) : 0.572

kshah223@cs-bertvm:~$ ~lillis/CS251-public/mystery 800
n: 800 ; cpu elapsed time (sec) : 0.801
```

Here, when the size increases the runtime increases by the factor of 25 which is 5^2 so on this basis I came to the conclusion that the runtime is of O(n^2). Hence the d in the question is 2.

PROBLEM 4: algorithm design (40 points):
You are given two arrays each containing n floating point numbers with the following properties:

f[]: values are in non-decreasing order (smallest to largest)
g[]: values are in non-increasing order (largest to smallest).

We want to find an index i such that MAX(f[i], g[i]) is minimized (over all indices i).

In other words, we have the following objective (sometimes called a "minimax" objective):

$$\min_{i:\ 0 \le i < n} (max(f[i], g[i]))$$

(A): Warmup - Given an example instance of this problem with n=10.

1. Specify the contents of both arrays f[] and g[].
2. Make things "non-trivial" -- e.g., although two arrays populated with identical values is a valid input, it is not very interesting.
3. For each index show the maximum of the two corresponding array entries.
4. Identify an index which yields the minimum value from (3). (In general, there may be more than one such index).

(template given below...)

f[]	5	7	9	6	1	54	10	15	21	37
g[]	17	4	9	10	19	4	10	7	28	40
max	17	7	9	10	19	54	10	15	28	40

(B): Now the fun part: devise an $O(\log n)$ time algorithm for this problem. Give a full implementation and explain the key concepts in your approach.

```
// returns index i minimizing max(f[i], g[i]) [assume n>0]
int minimax(double f[], double g[], int n) {
      double min = INFINITY;
      int currIndex = n/2;
      int lastIndex = currIndex;
      int minimumIndex = 0;
      int maximumIndex = n;
```

3 Problem 3 15 / 20

- **0 pts** Correct (d = 2.5 approximately)

✓ **- 5 pts** **Almost correct with honest effort or significant mistake with significant honest effort**

- **10 pts** Significant mistake, but some honest effort
- **20 pts** Incorrect / no effort
- **20 pts** Click here to replace this description.

Here, when the size increases the runtime increases by the factor of 25 which is 5^2 so on this basis I came to the conclusion that the runtime is of O(n^2). Hence the d in the question is 2.

PROBLEM 4: algorithm design (40 points):
You are given two arrays each containing n floating point numbers with the following properties:

f[]: values are in non-decreasing order (smallest to largest)
g[]: values are in non-increasing order (largest to smallest).

We want to find an index i such that MAX(f[i], g[i]) is minimized (over all indices i).

In other words, we have the following objective (sometimes called a "minimax" objective):

$$\min_{i:\ 0 \le i < n} (max(f[i], g[i]))$$

(A): Warmup - Given an example instance of this problem with n=10.

1. Specify the contents of both arrays f[] and g[].
2. Make things "non-trivial" -- e.g., although two arrays populated with identical values is a valid input, it is not very interesting.
3. For each index show the maximum of the two corresponding array entries.
4. Identify an index which yields the minimum value from (3). (In general, there may be more than one such index).

(template given below...)

f[]	5	7	9	6	1	54	10	15	21	37
g[]	17	4	9	10	19	4	10	7	28	40
max	17	7	9	10	19	54	10	15	28	40

(B): Now the fun part: devise an $O(\log n)$ time algorithm for this problem. Give a full implementation and explain the key concepts in your approach.

```
// returns index i minimizing max(f[i], g[i]) [assume n>0]
int minimax(double f[], double g[], int n) {
      double min = INFINITY;
      int currIndex = n/2;
      int lastIndex = currIndex;
      int minimumIndex = 0;
      int maximumIndex = n;
```

4.1 4.A **17 / 20**

- **0 pts** Correct
- **15 pts** Incorrect or No explanation

✓ **- 3 pts** **Minor mistake**

- **20 pts** Click here to replace this description.

Here, when the size increases the runtime increases by the factor of 25 which is 5^2 so on this basis I came to the conclusion that the runtime is of O(n^2). Hence the d in the question is 2.

PROBLEM 4: algorithm design (40 points):
You are given two arrays each containing n floating point numbers with the following properties:

f[]: values are in non-decreasing order (smallest to largest)
g[]: values are in non-increasing order (largest to smallest).

We want to find an index i such that MAX(f[i], g[i]) is minimized (over all indices i).

In other words, we have the following objective (sometimes called a "minimax" objective):

$$\min_{i:\ 0 \le i < n} (max(f[i], g[i]))$$

(A): Warmup - Given an example instance of this problem with n=10.

1. Specify the contents of both arrays f[] and g[].
2. Make things "non-trivial" -- e.g., although two arrays populated with identical values is a valid input, it is not very interesting.
3. For each index show the maximum of the two corresponding array entries.
4. Identify an index which yields the minimum value from (3). (In general, there may be more than one such index).

(template given below...)

f[]	5	7	9	6	1	54	10	15	21	37
g[]	17	4	9	10	19	4	10	7	28	40
max	17	7	9	10	19	54	10	15	28	40

(B): Now the fun part: devise an $O(\log n)$ time algorithm for this problem. Give a full implementation and explain the key concepts in your approach.

```
// returns index i minimizing max(f[i], g[i]) [assume n>0]
int minimax(double f[], double g[], int n) {
      double min = INFINITY;
      int currIndex = n/2;
      int lastIndex = currIndex;
      int minimumIndex = 0;
      int maximumIndex = n;
```

```
      while (currIndex >= 0 && currIndex < n)  {
            if (f[currIndex] > g[currIndex])  {
                  if (f[currIndex] < min)  {
                        min = f[currIndex];
                        minimumIndex = currIndex;
                        maximumIndex = currIndex;
                  }
                  currIndex = currIndex / 2;
            }
            else  {
                  if (g[currIndex] < min)  {
                        min = g[currIndex];
                        minimumIndex = currIndex;
                  }
                  currIndex = (currIndex + maximumIndex) / 2;
            }
            if (lastIndex == currIndex)
                  break;
            lastIndex = curIndex;
      }
      return minIndex;
}
```

PROBLEM 5 (30 points): Suppose you are given an array of numbers (in arbitrary order) and the boundaries of a sub-array of interest; you are supposed to determine the largest element in the sub-array.

The following approach should not surprise you:

4.2 4.B 17 / 20

- **0 pts** Correct

✓ **- 3 pts** **Minor mistake**

- **10 pts** Incorrect Logic
- **20 pts** Click here to replace this description.
- **15 pts** Click here to replace this description.

```
      while (currIndex >= 0 && currIndex < n)  {
            if (f[currIndex] > g[currIndex])  {
                  if (f[currIndex] < min)  {
                        min = f[currIndex];
                        minimumIndex = currIndex;
                        maximumIndex = currIndex;
                  }
                  currIndex = currIndex / 2;
            }
            else  {
                  if (g[currIndex] < min)  {
                        min = g[currIndex];
                        minimumIndex = currIndex;
                  }
                  currIndex = (currIndex + maximumIndex) / 2;
            }
            if (lastIndex == currIndex)
                  break;
            lastIndex = curIndex;
      }
      return minIndex;
}
```

PROBLEM 5 (30 points): Suppose you are given an array of numbers (in arbitrary order) and the boundaries of a sub-array of interest; you are supposed to determine the largest element in the sub-array.

The following approach should not surprise you:

```
// returns largest element in a[lo]...a[hi]
double largest(double a[], unsigned int lo, unsigned int hi) {
int i;
double answer;

  if(hi < lo)           // just for completeness... return smallest
    return -DBL_MAX;    //    double (negation of largest double) if
                        //    sub-array is "empty:

  answer = a[lo];
  for(i=lo+1; i<=hi; i++) {
    if(a[i] > answer)
       answer = a[i];
  }
  return answer;
}
```

It should also come as no surprise that the runtime of this approach is $\theta(n)$ where n is the size of the given sub-arrary (hi-lo+1).

But this isn't the *only* way to solve the problem. Consider this divide-and-conquer approach:

```
     BASE-CASES:
          if subarray is empty, return -DBL_MAX
          if subarray is size-one, return that element

     RECURSIVE CASE:

       A. Otherwise, split the subarray into two equal-sized halves (as
          close to equal as possible).
       B. Recursively compute the largest element in the left half.
       C. Recursively compute the largest element in the right half.
       D. return the maximum value produced by the two recursive calls
          (steps B, C).
```

Your Tasks:

Part-I: Write a C++ function implementing the divide-and-conquer approach described above. Function stub given below.

```
double largest2(double a[], unsigned int lo, unsigned int hi) {

      if (lo == hi)
           return a[lo];

      int mid = (lo+hi)/2;

      double x = largest2(a,lo,mid);
      double y = largest2(a,mid+1,hi);


      if(x < y)
          return y;
      else
          return x;

}
```

Part-II: Now give and justify a recurrence-relation describing the runtime of this algorithm.

T(n) = ___O(n)_____

Part-III: Now analyze your recurrence relation and derive a tight runtime bound for the algorithm (i.e., a Θ bound). Justify your conclusions.

Answer: T(n) = (1 + 2T(n/2)
= (2^(k+1) - 1) + (2^k)T(n/2^k)
= n + (2n - 1)
= 3n - 1
Hence, the final runtime is O(n).

5.1 Part-I 9 / 10

- **0 pts** Correct

✓ **- 1 pts doesn't handle "high < low" case correctly**

- **1 pts** Minor bug in program
- **2 pts** 2 minor bugs
- **2 pts** Solution is not recursive
- **3 pts** Algorithm is not complete
- **10 pts** problem not submitted
- **1 pts** Incorrect value for "mid" (should be (low+high)/2 or equivalent)
- **1 pts** Copies entire arrays which increases runtime complexity.
- **10 pts** Click here to replace this description.

Your Tasks:

Part-I: Write a C++ function implementing the divide-and-conquer approach described above. Function stub given below.

```
double largest2(double a[], unsigned int lo, unsigned int hi) {

      if (lo == hi)
           return a[lo];

      int mid = (lo+hi)/2;

      double x = largest2(a,lo,mid);
      double y = largest2(a,mid+1,hi);


      if(x < y)
          return y;
      else
          return x;

}
```

Part-II: Now give and justify a recurrence-relation describing the runtime of this algorithm.

T(n) = ___O(n)_____

Part-III: Now analyze your recurrence relation and derive a tight runtime bound for the algorithm (i.e., a Θ bound). Justify your conclusions.

Answer: T(n) = (1 + 2T(n/2)
= (2^(k+1) - 1) + (2^k)T(n/2^k)
= n + (2n - 1)
= 3n - 1
Hence, the final runtime is O(n).

5.2 Part-II 0 / 10

- **0 pts** Correct
- **5 pts** Justification missing or some mistake

✓ **- 10 pts** **Incorrect or no answer**

- **10 pts** Click here to replace this description.

Your Tasks:

Part-I: Write a C++ function implementing the divide-and-conquer approach described above. Function stub given below.

```
double largest2(double a[], unsigned int lo, unsigned int hi) {

      if (lo == hi)
           return a[lo];

      int mid = (lo+hi)/2;

      double x = largest2(a,lo,mid);
      double y = largest2(a,mid+1,hi);


      if(x < y)
          return y;
      else
          return x;

}
```

Part-II: Now give and justify a recurrence-relation describing the runtime of this algorithm.

T(n) = ___O(n)_____

Part-III: Now analyze your recurrence relation and derive a tight runtime bound for the algorithm (i.e., a Θ bound). Justify your conclusions.

Answer: T(n) = (1 + 2T(n/2)
= (2^(k+1) - 1) + (2^k)T(n/2^k)
= n + (2n - 1)
= 3n - 1
Hence, the final runtime is O(n).

Appendix: what makes a good "Algorithm Description"??

First, an algorithm description is not the same as a program. What makes a good/correct algorithm description? Like a lot of things, we know it when we see it, but a formal set of rules is not always so easy. Nevertheless, I'll try below:

1. **It should be translatable to an actual implementation by any competent computer scientist.** After you have a draft, re-read your description as if you are seeing it for the first time. Does it pass this test? Are there ambiguities that you should resolve?
2. **Just the right amount of detail.** It does not get bogged down with language specific issues and implementation details. For example, for Problem-2, we would probably represent an interval as a structure in C; including the code for an INTERVAL typedef would not be expected or desired! The person described in (1) knows how to do this already!
3. **Step-by-step presentation.** It avoids a "narrative" (paragraph form) presentation in favor of a step-by-step description. Maybe expressed as bullet points.
4. **You do not need to re-invent known algorithms.** If, for example, you want to perform a sort of some kind, you don't have to re-invent merge-sort. You can just say "use merge-sort to…". The "..." part is probably important though! What exactly are you sorting and the sorting criteria are important in an algorithm description.
5. **Drawings!** A diagram or two is often useful. Pay careful attention to how you label your diagrams.
6. **The Big Picture.** Often a "bird's-eye view" description of the key concepts in the algorithm before a more detailed description can make a huge difference!

A good algorithm description also usually needs two things along with it:

A. **Correctness.** Some kind of argument of correctness -- why it works. It is up to you to show us that it works; it is not our job to determine if/why it does/does not work. This may be folded into the algorithm description in some cases, but it must be in there somehow.
B. **Runtime analysis.** Sometimes this will be trivial, but it must still be in there.

5.3 Part-III 10 / 10

✓ **- 0 pts Correct**

- 10 pts Did not attempt

- 2 pts Correct tight bound, but no explanation

- 2 pts Attempted to analyze non-recursive function

- 5 pts does not display honest effort

- 1 pts Analysis good but no tight runtime given.

- 1 pts Recursion tree given, but it does not demonstrate runtime clearly

- 2 pts error in analysis leads to wrong runtime

- 1 pts Proof by example

- 2 pts Answer is right for algorithm given (which is incorrect), but wrong for recurrence relation given (which is correct).

- 1 pts incorrect use of master theorem, but reached correct answer

- 1 pts Correct answer for wrong algorithm and wrong recurrence relation, but not correct answer for correct algorithm

- 1 pts Correct answer, but analysis is incorrect

- 10 pts Click here to replace this description.