CS251 - Written HW2
Spring 2018
Due Monday, FEB 26 by 8:59AM
Submission: will be through gradescope
TOTAL POINTS: 250 (7 problems)

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").

PROBLEM 1 (60 pts.): For each of the C functions below, give as tight a worst-case runtime bound as you can. Express your answers with Big-O / Big-Theta notation **in the simplest terms possible.** Show your reasoning.

```
// 7 points
int A(int a[], int n) {
int x=0, i, j;

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

// 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;
}
```

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

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

// 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;
}
```

```
// 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;
}

// 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;
}

// 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;
}
```

PROBLEM 2 - algorithm design (40 points):
Find the Missing Integer.
You are given a sorted array of N ***distinct*** integers drawn from $\{0..N\}$ (you know this ahead of time).

Since there are only N array entries and there are N+1 distinct values in $\{0..N\}$, exactly one of them must be missing from the array.

Your job: devise an $O(\log N)$ time algorithm to find the missing integer.

HINT: Think of a "binary-search-like" approach.

COMMENT: You can present your algorithm as a C function but, you must also include an explanation of why it works -- i.e., the underlying logic of your solution.

PROBLEM 3 - analysis (30 points):
QuickSort on n identical elements.
Suppose the array-based quicksort implementation given in class is applied to an array in which all n elements are identical.

Analyze the runtime of quicksort in this scenario. Give your answer using big-Theta notation and ***explain clearly*** how you arrived at your answer (i.e., *convince* us!)

(The C++ implementation given in class also appears in Appendix B for reference).

PROBLEM 4 - analysis (30 points):
Solvable problem size with given time budget: Suppose we have an algorithm with best and worst-case runtime of $\Theta(n^3)$.

On our current system, it takes about 60 seconds for the algorithm to solve an instance of size n=1000.

Your Job: Determine approximately how large of a problem can be solved with double the time budget (120 seconds).

Show your work/reasoning! COMMENT: this is really just an algebra problem!

PROBLEM 5 - analysis (30 points):
Solving the duplicates problem via divide and conquer.
Recall the has_dups problem: given an array, determine if there are any duplicates in the array and return true if there are and false otherwise (in which case all elements are distinct).

On the next page is a divide-and-conquer approach to the problem.

The idea is simple:

- Recursively determine if the left half has any duplicates
- Recursively determine if the right half has any duplicates
- If neither half has duplicates, there may still be a duplicate -- one copy on the left and one on the right.
    - determine if this is the case by looping through the candidate pairs (one from left; one from right).

Your Job:

(**A**) Write a recurrence relation for the runtime of this algorithm. Explain each component of the recurrence relation.

(**B**) Use the recursion tree method (as we did with MergeSort) to derive a tight (big-Theta) bound on the worst-case runtime of the algorithm.

(**C**) Discuss: how does this approach compare with the simple approach you know from Homework 1?

```
// subarray a[lo...hi].  N
bool _has_dups(int a[], int lo, int hi) {
int i, j, m;
bool ldups, rdups;

  if(hi <= lo) return false;  // zero or one element: no dups!

  m = (lo + hi)/2;

  ldups = _has_dups(a, lo, m);
  rdups = _has_dups(a, m+1, hi);

  if(ldups || rdups)
     return true;

  // could still be dups:  one on left & one on right

  for(i=lo; i<=m; i++) {
     for(j=m+1; j<=hi; j++) {
        if(a[i]==a[j]) return true;
     }
  }
  return false;
}
bool has_dups(int a[], int n){
  return _has_dups(a, 0, n-1);
}
```

PROBLEM 6 (10 points):
I claim that merge-sort is actually a linear time algorithm -- i.e., *faster* than Θ(NlogN).

Below is a sketch of my "proof" that the runtime of merge-sort is actually $O(n)$.

For simplicity, I am only proving my claim for powers of 2 (so I don't have to worry about floors and ceilings).

We already know that the runtime of merge-sort is described by the recurrence relation

$$T(1) = 1$$
$$T(n) = 2T(n/2) + n \quad \text{for } n > 1$$

I want to show that $T(n) = O(n)$. To make this Big-Oh claim, I will show that that $T(n) \leq cn$ for all $n \geq 1$ and some constant c.

BASE-CASE: $T(1) = 1 \leq c \times 1$ as long as $c \geq 1$ clearly such a constant exists!

INDUCTIVE HYPOTHESIS: Assume the claim holds for *powers of 2 less than* k where k is itself a power of 2:

$k = 2^i$ where $i > 0$ (and so $k \geq 2$).

Notice that this covers $T(k/2)$

Now I have to prove the claim holds at k.

PROOF:

$$T(k) = 2T(k/2) + k \qquad \text{from recurrence relation and } k > 1$$
$$\leq 2c(k/2) + k \qquad \text{by I.H. applied to } T(k/2)$$
$$= ck + k \qquad \text{algebra}$$
$$= (c+1)k \qquad \text{algebra}$$
$$= O(k) \qquad \text{because } (c+1) \text{ is a constant and a constant times } k \text{ is still } O(k), \text{ since we know that "constants don't matter"!}$$

Q.E.D.

Of course, I am wrong! The runtime of MergeSort is really $\Theta(n \log n)$.

Your Job: Answer this question:

What is fundamentally wrong with my "proof"?

NOTES:

The problem is NOT that the proof only holds for powers of two.

(Indeed, if the above proof ***were*** correct, it would not be too hard to use it to show that the claim also holds when *n* is not a power of 2).

The "real" problem is much more fundamental than that!

PROBLEM 7 (30 points):

Exercise 7.53 (parts (a) and (b) only) from Weiss ("Data Structures and Algorithm Analysis in C++", 4th edition).

Note: the exercise includes the following hint:

> *"Hint: Sort the items first. After that is done, you can solve the problem in linear time."*

While it is true that you can solve the problem in linear time after sorting, it is not *necessary* to achieve the runtime target.

Appendix A: 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.

Appendix B: QuickSort implementation

```
void qs(int a[], int l, int r){
        int i, j, pivot;

        if(r <= l)
                return;

        if(r==l+1) {
                if(a[l] > a[r])
                    swap(a, l, r);
                return;
        }
        // median of three
        int m = (l+r)/2;
        if(a[m] < a[l])
                swap(a, l, m);
        if(a[r] < a[l])
                swap(a, r, l);
        if(a[r] < a[m])
                swap (a, r, m);
        swap(a, m, r-1);

        pivot = a[r-1];
        i = l;
        j = r-1;
        for(;;){
                while(a[++i] < pivot);
                while(a[--j] > pivot);
                if(i >= j) break;
                swap(a, i, j);
        }
        swap(a, i, r-1); // replace pivot

        qs(a, l, i-1);
        qs(a, i+1, r);
}

void swap(int a[], int i, int j) {
        int tmp;

        tmp =  a[i];
        a[i] = a[j];
        a[j] = tmp;
}
```