CS570
Analysis of Algorithms
Fall 2010
Exam I

Name: _____

Student ID: _____

DEN Student YES / NO

	Maximum	Received
Problem 1	20	
Problem 2	20	
Problem 3	15	
Problem 4	15	
Problem 5	15	
Problem 6	15	
Total	100	

2 hr exam
Close book and notes

If a description of an algorithm is required, please limit your description within 200 words, anything beyond 200 words will not be considered. Proof/justification or complexity analysis will only be considered if the algorithm is either correct or provided by the problem.

1) 20 pts
Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

T
For a connected graph with *n* vertices and *m* edges, $\Theta(m+n) = \Theta(m)$ is always true..

F
$f(n) = \Theta(f(n/2))$

T
Stable matching: Given any non-empty and valid pair of preference lists, there should be a stable matching.

F
Union/merge operation can be done on a binary heap of size *n* in O(log *n*) time.

F (Finding would take O(n))
Given a Fibonacci Heap storing *n* integers, finding a given integer and deleting it can be done in O(log *n*) amortized time.

T
An undirected graph $G(V, E)$ must contain at least 1 cycle if | *E* | = | *V* |.

F
An undirected graph $G(V, E)$ must be connected if | *E* | > | *V* | - 1.

F or T (Ambiguous, everybody get points for this problem)
Prim's algorithm is more suited than Kruskal's algorithm for sparse graph.

T
For any real constants $a > 0, b > 0$,and *c*,following is always true: $\left(\frac{n}{a} + c\right)^b = O(n^b)$.

F
The advantage of Fibonacci heap over binary heap is that it has better amortized cost for the EXTRACT-MIN operation.

2) 20 pts
a) Arrange these functions under O notation using only = (equivalent set) or $\subset$ (strict subset of):
E.g. for functions $n,\ n+1,\ n^2$, the answer should be O(n+1) = O(n) $\subset$ O(n^2).

$n^2-5,\ \log_3(n+4),\ \sin(n^3),\ \log_2(n^{10n}),\ 2^{2^{n-3}},\ \log_2(n-1),\ 3,\ \sqrt{6n},\ n^n,\ \log_2(n!),\ n\log_4 n^2,\ n!$

Solution:

$$O(\sin(n^3)) \subset O(3) \subset O(\log_3(n+4)) = O(\log_2(n-1)) \subset O(\sqrt{6n}) \subset O(\log_2(n^{10n}))$$
$$= O(\log_2(n!)) = O(n\log_4 n^2) \subset O(n^2-5) \subset O(n!) \subset O(n^n) \subset O(2^{2^{n-3}})$$

b) Find the complexity of the following nested loop (using Θ notation)

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++)) {
    C[i][j] = 0.0;
    for (k=0; k<p; k++) {
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

$\Theta(mnp)$

c) Find the complexity of the following code section (using Θ notation)

```
for (i=0; i<n; i++) {
  for (j=0; j<i; j++) {
    // by calculating median, we first do a sort, and
assume
    // we use the fastest sorting algorithm
    // assume that A is never changed.
    B[j] = A[j]/median(A, i);
  }
}
```

$\Theta(n^2)$, while we also give points to those provided $\Theta(n^2 logn)$ as answers.
Assume that we are using a second array to store the results of sorting first *i* elements of A, each increment of *i* would make this array updated, by doing an insertion for the newly arrived element, this takes $\Theta(i)$ time, and this time doesn't change for the later median calculation for B, so for the whole B calculation under a single *i*, it takes $\Theta(i)$ time. To sum them up, it takes $\Theta(n^2)$ time.

d) Find the complexity of the following function (using Θ notation)

```
int countPrime(int n) {
  counter = 0;
  for(i=101; i<n; i+=2) {
    isPrime = true;
    for(j=3; j<i/2; j+=2)) {
      if(i%j == 0) {
        isPrime = false;
        break;
```

```
        }
      }
      if(isPrime) counter ++;
    }
    return counter;
```

$\boldsymbol{\Theta(n^2)}$

3) 15 pts

You are given a complete binary tree of n nodes storing n integers, which satisfies a constraint that for any node, its integer value is not smaller than the max integer in its left subtree (if exists) and not larger than the min integer in its right subtree (if exists). This complete binary tree is stored as an array. Give an algorithm with O(n) complexity to sort this array in non-decreasing order.
E.g. the original array is [7, 4, 9, 2, 5, 8], and the sorted array should be [2, 4, 5, 7, 8, 9].

Solution:
Given the properties the binary tree has, there's a standard method called inorder traversal to do this. The idea is that for any node, we need to first finish place all the integers in its left subtree, then its own integer, then integers in its right subtree. Therefore we can do this neatly using recursion. In the following code, a[] is the original array, and b[] is the sorted array, global integer variable ind is initialized as 0, and we call visit(0) to start sorting.

```
void visit(int k)
{
    if (k>=n) return;
    visit(k*2+1);
    b[ind++] = a[k];
    visit(k*2+2);
}
```

Since every integer is visited exactly once and the number of out-of-boundary visits (for index larger than n-1) is O(n), we sort the original array in O(n) time.

Comments:
Some people give an algorithm to solve this in left-mid-right order without using recursion. This will lead to either incorrect answer or higher complexity. You need to state clear about the details on how to do this, since it is critical part of the solution to this problem.

4) 15 pts

Suppose there are n stairs, and one can only climb 1 or 2 steps, how many different ways of climbing n stairs? E.g. there are 5 stairs, so (1,1,1,1,1) and (1,2,1,1) are two different ways. Describe a divide-and-conquer algorithm and its time complexity.

Let $T(n)$ denote the number of different ways for climbing up n stairs.
There are two choice for each first step: either 1 or 2, so we divide the problem into two sub problems: $T(n-1)$, $T(n-2)$. And then let us check the boundary conditions: when there is only one step, $T(1) = 1$. If only two steps, $T(2) = 2$.

Here is the pseudo code for the process:

ClimbSteps (n):

```
if n is 1
  return 1
else if n is 2
  return 2
else
  F[1] = 1
  F[2] = 2
  for i = 3 to n do
    F[i] = F[i-1]+F[i-2]
  return F[n]
```

It takes one for loop for each calculation, therefore its time complexity is $\Theta(n)$.

5) 15 pts

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Solution:

We first sort all the coordinates of the points in ascending order within O(n log n) time and the result is array x[]. Then we place a unit-length closed interval starting at x[0] (thus ending at x[0]+1). We then look at the elements in x[] one by one until the current one is not covered by the interval we just placed. Then we place a new interval starting at this point and do the same thing. When we have no more point to cover, we get a covering solution. The overall complexity is O(n log n).

Now we can argue that our solution is optimal, i.e. the number of intervals is minimized. Obviously we covered all the given points by running our algorithm. Suppose that there's an optimal solution B with k intervals and different from our solution (called A). Then we sort the intervals in B according to their starting point in ascending order. Since B is different from A, we can find the left-most different interval used in B, say interval B[i]. We know that B[i] cannot start from any point right to corresponding A[i] since otherwise the point in x[] which A[i] starts at cannot be covered by B. So B[i] starts left to the start point of A[i]. Then we can move B[i] to the position of A[i] without covering any less point so the result would be the same or better. By doing this for all the B[i] different from A[i], we can claim that A does not use more intervals than B does, therefore A is optimal and our solution is correct.

Comments:

There are some common mistakes that many of you have made:

1. Pick x1 as the left-most one without sorting. A set contains no order.
2. Instead specifying the position of intervals, some used words like "use an interval to cover x1" and this is usually not considered correct.
3. Traverse over the real line to find points. This is not doable. You need to check the coordinates of given points instead.
4. Proof not strict. This is the most common mistake, especially for those using induction proofs. If you use induction, when goes from n=k to n=k+1, only arguing that in every previous step your solution covers at least the same number of points is not enough to get to the conclusion. In this case, you also need to prove that in every previous step, your solution will have a last interval reaches at least as far as the optimal solution. Although this seems trivial, it is logically necessary. This and some other detail issues in proof lead to serious deduction of points.

6) 15 pts

There are m universities, each with a certain number of available research faculty positions for hiring fresh graduated PhDs. There are n PhDs students to be graduated at a given year, each interested in joining one of the universities. Each university has a ranking of the students in order of preference and each student has a ranking of the universities in order of preference. In general, there could be many more students graduating than there are slots in the m universities.

The interest, naturally, was in finding a way of assigning each student to at most one university, in such a way that all available positions in all universities were filled. We say that an assignment of students to universities is stable if neither of the following situations arises:

- First type of instability: There are students s and s', and a university u, so that
  - s is assigned to u, and
  - s' is assigned to no university, and
  - u prefers s' to s
- Second type of instability: There are students s and s', and university u and u', so that
  - s is assigned to u, and
  - s' is assigned to u', and
  - u prefers s' to s, and
  - s' prefers u to u'.

So we have the Stable Matching Problem, except that (i) universities generally want more than one student, and (ii) there is a surplus of students.

Show that there is always a stable assignment of students to universities, and give an algorithm to find one.

Solution:

We have n students and u universities. Each university u_i has k_i open positions to fill. The total # of positions is $p = \sum_{i=1}^{m} k_i$, $p \leq n$. We propose the following algorithm to fill each p_j position:

Algorithm

While there exists an unfilled position p_j:

The university u_i that contains p_j, select its highest ranked student s on its list, whom u_i has not yet proposed to.

if s is free

add student s to position p_j

else // s is already engaged

if student s prefers p_j to its engaged position p_j'

add student s to p_j

release p_j'

Complexity

Since there are m universities and each university has a list of n students, at the worst case, each university will go through all of them, the complexity is $O(mn)$.

Proof of Stability

Instability 1 cannot occur, because if u prefers s' to s, u would have proposed to s' first. Since once a student has a position, he/she would never be free again, u must have never proposed to s'. This is a contradiction.

Instability 2 cannot occur, because if u prefers s' to s, u would have proposed to s' first, s' then would have withdrawn from u to go to u'. So s' would prefer u' to u. This draws another contradiction.

Therefore, the instability cannot happen.