EXAM I - PRACTICE I
15-211
SUMMER SESSION 2, 2010

MATTHEW MIRMAN (MMIRMAN@ANDREW.CMU.EDU)

Name : _____

Andrew ID (email) : _____

Date : _____

Start Time : _____

Instructions:

- Write your full name and **Andrew ID** as neatly as humanly possible.
- You have 1 hour and 30 minutes to finish.
- There are 6 pages, and 11 problems.
- No Notes or books are permitted.
- Collaboration is not permitted.
- For full credit, show work, and write neatly. Illegible answers may not receive full credit.

Part 1. **Recurrences & Asymptotics**

Problem 1. Find a closed form for the following recurrences

(1) $T(n) = \frac{n}{2}T(n-1)$ where $T(1) = 1$

Notice that nothing is added in the recursion that that without being divided by two, the equation would be:

$$T(n) = nT(n-1) = \prod_{i=1}^{n} i = n!$$

Also, notice that

$$T(n) = \frac{T(n-1)}{2} = \prod_{i=1}^{n} \frac{1}{2} = \frac{1}{2^n}$$

Thus, $T(n) = \frac{n!}{2^n}$

(2) $T(n) = n + T(n-1)$ where $T(0) = 0$

This turns out to be $T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Problem 2. Use the master method to find a close bound for $T(n) = 3T(\frac{n}{2}) + n\log_4 n$

Let $f(n) = n\log_4 n$ and $a = 3$ and $b = 2$. First we can notice that $f(n)$ is probably in $O(n^{\log_b a - \epsilon})$, but we need to show this formally.
First, notice that $2 > \log_b a - \epsilon > 1$. we can thus write this as $1 + v = \log_b a - \epsilon$ where $0 < v < 1$. thus we want to show $f(n) \in O(n^{1+v}) = O(n * n^v)$ we already know that $n \in O(n)$ so we need to show $\log_4 n \in O(n^v)$. We know $\log_4 n \in O(\ln n)$ given a previous proof.
Let $n_0 = (\frac{1}{v})^{\frac{1}{v}}$. So AFSOC that there exists an $n > n_0$ such that $\ln n > n^v$. Then $\frac{1}{n} > v * n^{v-1} \implies 1 > v * n^v \implies (\frac{1}{v})^{\frac{1}{v}} > n$. This is a contradiction.
Thus, $\ln n \in O(n^v)$. Thus $\log_4 n \in O(n^v)$, and $n \in O(n)$
Finally, multiply these together to find that $n\log_4 n \in O(n^{\log_b a - \epsilon})$.
We can thus apply master method and find that $T(n) \in \Theta(n^{\log_b a})$.

Problem 3. Use the tree method to find a solution (not necessarily closed form) to $T(n) = T(\frac{n}{2}) + n$ given base case $T(1) = 1$

This method actually doesn't require a tree because there is no integer multiplying $T(\frac{n}{2})$. You should notice however that the "Line" drawn would be depth about $\log_2 n$, and that at depth i, it does $n(\frac{1}{2})^i$ work. thus you can set up the equation $T(n) = n\sum_{i=1}^{\log_2 n} \frac{1}{2^i}$.

Problem 4. Prove or disprove that $(\forall s, t > 1)\ n^{\log_s n} \in \Theta(n^{\log_t n})$

This is not true. If we write $n^{\log_s n} = n^{\frac{\log n}{\log s}} = (n^{\ln n})^{\frac{1}{\ln s}}$, we can say that the statement is equivalent to $(\forall s, t > 1)\ (n^{\ln n})^{\frac{1}{\ln s}} \in \Theta((n^{\log n})^{\frac{1}{\ln t}})$. But what if we pick $s = 10$ and $t = 100$. Then this says $n^{\log n} \in O(n^{\frac{1}{2}\log n})$. Suppose that $\exists n_0, c > 0$ such that $\forall n > n_0$, $n^{\ln n} < cn^{\frac{1}{2}\ln n}$. Thus $n^{\ln n} < c$. This is a contradiction, because $\ln n > 1$ for values of $n > 3$ and thus also for $n > c$, $n^{\ln n} > n > c$.
Therefore, the statement is incorrect. (Note, this is pretty informal. I'd like you to try doing this as formally as possible-more so than this- but on a test, I'll give you some slack)

Problem 5. Let $f(n) = 4n + n\log_2(n^2)$ Circle all true statements

$f(n) \in O(2^n)$ $f(n) \in \Omega(2^n)$ * $f(n) \in O(n^2)$ * $f(n) \in \Omega(n^2)$

$f(n) \in O(n\log_2 n)$ *$f(n) \in \Omega(n\log_2 n)$* $f(n) \in O(n)$ *$f(n) \in \Omega(n)$*

Part 2. **Hashtables**

Problem 6. (circle all that apply) A working hashtable has a load factor of 3: **(Solution: 2, 7)**

(1) The hash table uses open addressing:

Open addressing requires load factors less than 1.

(2) The hash table uses separate chaining: **YES**

(3) The hash table uses linear probing:

Linear probing is open addressing

(4) The hash table's size is 9: **NO**

(5) The hash table's size is 3: **NO**

(6) The table is less space efficient than a table who's load factor is .5:

A table who's load factor is .5 is wasting half the space it uses. a table with a load factor of 3 is likely wasting no space.

(7) The table is less time efficient than a table who's load factor is .5:

This table has to go through on average 2 checks before if finds the element it wants. a table with a load factor of .5 would have to go through 1 check. This is true (although slightly).

(8) The table does not exist:
Load factors can be greater than 1 on tables with separate chaining.

Problem 7. Does the following class have a good hash function? Explain thoroughly. How could it be improved?

```
public class MagicData{
    short month;
    int min;
    String trunk;
    boolean cable;
    public boolean equals(Object other){
        MagicData b=(MagicData) other;
        return trunk.charAt(0)==b.trunk.charAt(0)
               && month==b.month
               && min== b.min
               && cable==b.cable;
    }

    public int hashCode(){
        return trunk.hashCode()*31
               + (cable? 91 : 7 )*37
               + min * 137;
    }
}
```

This is a horrible hashcode.
There are a number of poor coding choices here:

- The equals method in this only takes into account the first character of trunk, while the hashcode uses the hashcode from String which uses every character of trunk. This is wrong not only because its inefficient, but objects with different trunks but same first characters in trunk which might otherwise have the same data would get hashed to different locations in the hashtable and thus not be found on a hash table contains().
- 91 isn't prime. 7*91 isn't prime. 91*37 isn't prime. While not being prime isn't a problem, we don't know how well these numbers work.
- Not including the month parameter in the hashCode make it inefficient by making it possible that objects with different months (months matter according to the equals method) get hashed to the same location. Whether you use separate chaining or open addressing, you could have to go through over 11 different non equal objects to get to the one you want.
- The multiplied primes chosen are irregular. multiplied numbers should be chosen such that their location in the integer is well distributed.

Improvement:

```
public int hashCode(){
    return (trunk.charAt(0)*31*31*31
            + min*31*31
            +month*31
            +(cable?1:0);
}
```

Reasoning: Java's ''String''.hashCode() method simply multiplies returns $\sum_{i=0}^{n} 31^i c[i]$ with $c[i]$ being the i'th character (a byte) with values ranging from $0 - 255$. Why not just use this since we know it works well, and that all our data ranges from 0 to at most 255?

Problem 8. Fred Hacker has proposed a hashing scheme for strings of up to 10 bytes that works as follows. He initializes a table T = new int[10][256] with random 32-bit numbers. Here's his hash function:

```
static int hash(byte[] s) {
    int a=0;
    for (int i=0; i<s.length; i++)
        a += T[i][((int)s[i]) & 255];
    return a;
}
```

(1) What is the purpose of writing T[i][((int)s[i]) & 255] instead of the more obvious T[i][s[i]]?

This was silly. Sorry. Bytes in java happen to be signed. If we cast a byte to an int, and use &255, we ensure values in the byte range, but using an unsigned int so that you won't be attempting to access a negative element of the array.

(2) Assuming that the table has been initialized as described, what is the probability that "dog" and "god" have the same hash value?

Consider a position, i, where there is a difference (name the words 1 and 2). Then, let the sum so far of each be x_1, x_2. Furthermore, let the value at position i be $v[i][w_{1,i}]$ and $v[i][w_{2,i}]$ for each letter. Then, for the hash values to be the same,

$$x_1 + v[i][w_{1,i}] = x_2 + v[i][w_{2,i}]$$

and there is a $\frac{1}{2^{32}}$ probability that this holds. A simple way to think about this is in terms of degrees of freedom. In this case there is 1. You can choose any value for $v[i][w_{1i}]$, and then the probability that $v[i][w_{2i}]$ is the correct value is (trivially) $\frac{1}{2^{32}}$.

(3) Annoyed by the large size of the table T, Hacker changes it to a one-dimensional array with 256 random 32-bit integers. So line 4 of the computation becomes: a += T[((int)s[i]) & 255]; In this new scheme (assuming the table is initialized as described) what is the probability that the strings "abc" and "abd" have the same hash value?

Since there is no order, these strings hash to the same value exactly when $T['c'] = T['d']$. The probability of this is $\frac{1}{2^{32}}$.

(4) For the one-dimensional table from part (3), what is the probability that the strings "dog" and "god" have the same hash value?

In this case, the probability is 1 since position no longer matters, since always holds.

Part 3. Dynamic Programming

Problem 9. DP & Fib

(1) Write a bottom-up solution to the Fibonacci sequence: $F(n) = F(n-1) + F(n-2)$ with $F(0) = 0$ and $F(1) = 1$.

```
int fib(int n){
    if (n<2) return n;
    int table[]=new int[n+1];
    table[0]=0;
    table[1]=1;
    for (int i=2; i<=n ;i++){
        table[i]=table[i-1]+table[i-2];
    }
    return table[n];
}
```

(2) Find and prove a theta bound for your algorithm.
$T(n) \in \Theta(n)$ because for each value from $i = 1$ to $i = n$, all we do is make a call to two elements in the table. Not much proving to do here.

Problem 10. Write recurrences for the following problems:

(1) You have infinite amounts of 2, 3, 5, 6, and 7 cent coins. You want to make change for x cents, and you want to do it with as many coins as possible. (we don't care how its done, the number of coins total). Let $M(i)$ be the max number of coins usable to make i cents (with $i > 5$).
Here, I do not ask for an efficient algorithm, or an efficient equation, just a recursion relation, which comes naturally from the problem statement:
$M(i) = 1 + \max\{M(i-2),\ M(i-3),\ M(i-5),\ M(i-6),\ M(i-7)\}$
with $M(0) = 0,\ M(1 \text{ or } i<0) = -infinity$.

(2) Suppose a duck is sitting on one side of a river, and wants to get to the other side of the river without getting wet. Fortunately, the river is filled with floating rocket pads (numbered $1, \cdots, n$). If the duck is on rocket pad i, then the duck can jump to any rocket pad j with $i - k < j < i - 2$ or $i + 2 < j < j + k$ provided the rocket pad j exists. Unfortunately, some rocket pads are very hot from just being used to launch rockets, and the duck doesn't want to spend too much time on those. Let h_i be the heat gained by standing on rocket pad i. Find a recurrence for the minimum heat required for the duck to get from rocket pad 1 to rocket pad t (given that $n \geq t$).

Yet again, this was an exercise not in coming up with an efficient algorithm, but in writing an equation. Here, the equation is simply
$M(i) = h_i + \min\{\min_{i-k<j<i-2}\{M(j)\}, \min_{i+2<j<i+k}\{M(j)\}\}$.
Note that this is equivalent to trying to find minimum weight path from vertex a to vertex b on a graph, under certain connections.

Problem 11. You have a set of n integers each in the range $0...K$. You want to partition these integers into two subsets such that you minimize $|S_1 - S_2|$,where S_1 and S_2 denote the sums of the elements in each of the two subsets. Write some code (pseudo-code or java is fine) to solve this problem, and analyze your solution.

```
public LinkedList<Integer> partition(int [] vals){
   int prev_incl[]=new int[vals.length];
   int s1_sums[]=new int[vals.length];
   int halfway=0;
   for (int i=0;i<vals.length;i++){
       s1_sums[i]=vals[i];
       prev_incl[i]=-1;
       halfway+=vals[i];
   }
   halfway/=2;
   int best=0;
   for (int i=0; i<vals.length ; i++){
      for (int j=0; j<i; j++){
         if (Math.abs(vals[i]+s1_sums[j]-halfway)<Math.abs(prev_incl[i]-halfway)){
             prev_incl[i]=j;
             s1_sums[i]=vals[i]+s1_sums[j];
         }
      }
      if (Math.abs(s1_sums[i]-halfway)<Math.abs(s1_sums[best]-halfway)){
          best=i;
      }
   }
   LinkedList<Integer> reconstruct=new LinkedList<Integer>();
   while(best>=0){
       reconstruct.add(vals[best]);
       best=s1_sums[best];
   }
   return reconstruct;
}
```

This algorithm runs in $\Theta(n^2)$ time because we can construct the time equation $T(n) = n + F(n) + n$ with $F(n) = F(n-1) + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

so $T(n) = 2n + \frac{n^2+n}{2} \in \Theta(n^2)$.