Introduction to Algorithms – Assignment 2

1. Purpose: Apply recursion to solve a problem, practice formulating and analyzing algorithms. In the US, coins are minted with denominations of 50, 25, 10, 5, and 1 cent. A greedy algorithm for making change uses the smallest possible number of coins repeatedly and returns the biggest coin smaller than the amount to be changed until it is zero. For example, 17 cents will result in the series 10 cents, 5 cents, 1 cent, and 1 cent.

   a) Write a recursive algorithm for changing n cents as described above.

   ⇨ The algorithm is described as follows:

```
initialize arr1 = [50,25,10,5,1]
initialize arr2 = [0,0,0,0,0]
initialize i = 0;
Changify(i, n)
{
    if(n > 0)
    {
            if(arr1[i] <= n)
            {
                    x = n/arr1[i]
                    n = n % arr1[i]
                    arr2[i] = arr2[i] + x
            }
            increment i
            Changify(i, n)
    }
}
```

   b) Analyze its time complexity.

   Considering that each arithmetic operation takes some constant amount of time. We can assume that the time for the 4 arithmetic operations **x = n/arr1[i], n = n % arr1[i], arr2[i] = arr2[i] + x** and **increment i**, equals c. We know that the recursion calls will occur for a maximum of 6 times i.e one more than the size of the subproblem array. So, we can say that the time taken in total can be <= 6c.

   Hence, we can say that the running time complexity of the algorithm is O(6c) or O(1).

c) Write an O(1) (non-recursive!) algorithm to compute the number of returned coins.

⇨ The algorithm is as follows:

```
initialize arr1 = [50,25,10,5,1]
initialize arr2 = [0,0,0,0,0]
for(i = 1; i < arr1.length; i++)
{
    if(arr1[i] < n)
    {
            x = n/arr1[i]
            n = n % arr1[i]
            arr2[i] = arr2[i] + x;
    }
}
```

d) Show that the above greedy algorithm does not always give the minimum number of coins in a country whose denominations are 1, 6, and 10 cents.

⇨ By taking an example, we can successfully show that the above algorithm doesn't give the right results for different denominations.

For example: In case of n = 25.

The above algorithm will output: 2 coins of 10 and 5 coins of 1 cents. Hence a total of 7 coins.

But the optimal solution would be 4 coins of 6 and 1 coin of 1 cent. Hence a total of 5 coins which is less than 7 coins.

e) Given a set of arbitrary denominations c=(c1,...,cd), describe an algorithm that can compute the minimum number of coins required for making change. You may assume that c contains 1 cent.

⇨ The algorithm for the above problem is as follows:

1. The algorithm sorts the denomination array in descending order and then iterates through it starting from the largest denomination.
2. It produces a result giving the number of coins returned after the end of the while loop.
3. After getting the result each time, it proceeds to the next denomination in the array leaving the previous greater denomination out of the calculation and
4. After each iteration of the for loop it stores the minimum number of coins in a a variable which is returned when the for loop exists.

```
changeFor = input value for the money.
c=[c1,...,cd];
initialize minChange = 0
sort(c) //Sort in descending order.
GETCHANGECOUNT(x,changeFor)
    for(i = 1 <- c.length)
            initialize x = 0, j = i;
            while(changeFor != 0)
                    if(changeFor > c[j])
                            x += changeFor/c[j] //Increase the changeCount value by the
appropriate change of that denomination.
                            changeFor = changeFor - xC[j]; // Decrement number by the
denomination
                    increment j by 1
            if(x equals 0)
                    minChange = x
            else
                    minChange > x ? x:minChange //Set minChange to minumum of x or
minChange
    return minChange
```

2. For each of the following recurrences, use the Master Theorem to derive asymptotic bounds for T(n), or indicate why the Master Theorem does not apply. If not explicitly stated, please assume that small instances need constant time c. Justify your answers, in particular, for case 3 of the Master Theorem show that the regularity condition is satisfied.

(a) $T(n)=\frac{1}{2}T(n/3) + \sqrt{n}$

⇨ $T(n) = \frac{1}{2}\, T(n/3) + \sqrt{n}$
Here, $f(n) = \sqrt{n}$.
$a = \frac{1}{2}$
$b = 3$
In this case, master's theorem cannot be applied as value of $a < 1$. For Masters Theorem, the value of 'a' needs to be >= 1.

(b) $T(n)=7T(n/3) + n^2$

⇨ T(n) = 7T(n/3) + n^2
Here, f(n) = n^2
a = 7
b = 3

$$n^{\log_b a} = n^{\log_3 7}$$
$$n^{\log_3 7} = n^{\log 7 - \log 3} = n^{0.37}$$

Comparing f(n) and $n^{\log_b a}$:

$$n^2 \geq n^{0.37+0.1} \text{ for } \varepsilon = 0.1.$$

This leads us to case 3, where:

$$f(n) \in \Omega(n^{\log_b a + \varepsilon})$$

Therefore,

$$T(n) \in \Theta(f(n))$$

$$T(n) \in \Theta(n^2)$$

Checking regularity condition:

Which states that there is a c<1 and $n_0 \geq 0$ such that af(n/b) <= cf(n) for all n >= n_0.

af(n/b) = 7f(n/3) = 7/9(n^2) < 8/9(n^2). Therefore, value of c = 8/9 = 0.89.

(c) T(n)=4T(n/2) + n^2 sqrt(n)

⇨ T(n)=4T(n/2) + n^2 sqrt(n)

Therefore, a = 4, b = 2 and f(n) = n^2 sqrt(n) = $n^{5/2}$.

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

Comparing f(n) and $n^{\log_b a}$:

$$n^{5/2} > n^2$$
$$n^{5/2} > n^{2+0.4}$$

For $\varepsilon = 0.4$.

Using case 3, we can say that

$$f(n) \in \Omega(n^{\log_b a + \varepsilon})$$

Therefore,

$$T(n) \in \Theta(f(n))$$

$$T(n) \in \Theta(n^2 sqrt(n))$$

Checking regularity condition:

Which states that there is a c<1 and $n_0 \geq 0$ such that af(n/b) <= cf(n) for all n >= n_0.

af(n/b) = 4f(n/2) = 4(n^2/4)(sqrt(n/2)) = $n^{5/2}$/sqrt(2) < 0.8$\left(n^{5/2}\right)$. Therefore, value of c = 0.8.

(d) T(n)=2T(n/2) + n-n lg(n)

⇨ Here, a = 2, b = b and f(n) = n-n.lg(n) which is a negative function. We know that Masters theorem applies only to non-negative functions, hence in this case, Masters Theorem is not applicable.

(e) T(n)=sqrt(2)T(n/2) + lg(n)

⇨ Here, a = sqrt(2)
b = 2
f(n) = lg(n)

$$n^{\log_b a} = n^{\log_2 2^{1/2}} = n^{1/2}$$

Comparing f(n) and $n^{\log_b a}$:

$$\lg|n| \in o(n^{1/t})$$

Where t > 0.

Therefore, we can say that:

$$\lg|n| \leq (n^{\frac{1}{2}-\frac{1}{4}})$$

For $\varepsilon = 0.25$.

Using case 1:

$$f(n) \in O(n^{\log_b a-\varepsilon})$$

Therefore,

$$T(n) \in \Theta\left(n^{\log_b a}\right)$$

$$T(n) \in \Theta\left(n^{1/2}\right)$$

3. Please solve the following recurrences via iteration. (2 points each.) (a1 & a2) The two recurrences in lecture 4, page 7. Use T(1)=1. (b) The recurrence in lecture 6, page 7. Use T(1)=0.

a1):

$T(n) = 2T(n/2) + nlgn$

Assume that $n = 2^k$.
We can say that $lg|n| = k$

$T(2^0) = 1$

$T(2^1) = 2T(2/2) + 2lg2 = 2(1) + 2.1$

$T(2^2) = 2T(4/2) + 4lg4 = 2^2 + 2^2 + 2^2.2$

$T(2^3) = 2^4 + 2^4 + 2^4.2 + 2^4.3 + 2^4.4$

$T(2^k) = 2^k + 2^k + 2^k.2 + 2^k.3 + 2^k.4 + \ldots\ldots + 2^k.k = 2^k[1 + 1 + 2 + 3 + 4 + \ldots\ldots + k]$

Substituting n back in the equation.

$T(2^k) = n(1) + n(1+2+3+4+5+\ldots..+k)$

$T(2^K) = n(1 + k(k+1)/2)$ ……Summation of first n natural numbers

$T(2^K) = n(1 + (lg|n|(lg|n| + 1))/2)$

Therefore, $T(n) = n/2 + n/2(lg|n|(1+lg|n|))$

a2):

$T(n) = 2T(n/4) + sqrt(n)/\log_4 n$

Assuming $n = 4^k$
Therefore $\log_4 n = k$

$T(4^0) = 1$

$T(4^1) = 2.1 + 2^1/1$

$T(4^2) = 2(2.1 + 2^1/1) + 2^2/2 = 2^2 + 2^2 + 2^2/2$

$T(4^3) = 2^2(2+2) + 2^3/2 + 2^3/3 = 2^3 + 2^3 + 2^3/2 + 2^3/3$

$T(4^k) = 2^k + 2^K(1+ ½ + 1/3 + ¼ + \ldots\ldots\ldots + 2^k/k)$

$T(4^k) = 2^k + 2^k(\sum_{i=1}^{k} 1/i))$

Using integration, we can find the sum of a harmonic progression.

$T(4^k) = 2^k + 2^k(\int_1^k \frac{1}{i} di)$

$T(4^k) = 2^k + 2^k(\ln k - \ln 1)$

$T(4^k) = 2^k + 2^k(\ln k)$

Substituting for n:

$T(n) = \sqrt{n} + \sqrt{n}\ln(\log_4 n)$

(b) The recurrence in lecture 6, page 7. Use T(1)=0.

- ⇨ Using forward iteration:

$T(1) = 0$

$T(2) = 3/2(0) + 3/2(c)$

$T(3) = 4/3.(3/2c) + 5/3(c)$

$T(4) = 5/4.(4/3.(3/2.(c)) + 5/3.(c)) + 7/4(c)$

$T(5) = 6/5.5/4.4/3.3/2(c) + 6/5.5/4.5/3.c + 6/5.7/4.c + 9/5c$

Dividing and multiplying the last term with 6.

$T(5) = 6/5.5/4.4/3.3/2(c) + 6/5.5/4.5/3.c + 6/5.7/4.c + 9/5.6/6.c$

$T(5) = 6c\ (\ 9/(5.6) + 7/(5.4) + 5/(4.3) + 3/(3.2)\ )$

Observing the pattern, we can analyze that the numerator is of the form 2n-1 and the denominator is a product of n and (n+1). The series is a summation of all number starting from 2 and ending at k.

$T(k) = (n+1)c \sum_{i=2}^{k} \frac{2i-1}{i(i+1)}$

4. Purpose: Often, recursive function calls use up precious stack space and might lead to stack overflow errors. Tail call optimization is one method to avoid this problem by is replacing certain recursive calls with an iterative control structure. Learn how this technique can be applied to QUICKSORT. (2 points each) Solve Problem 7-4, a-c on page 188 of our textbook.

4.a: Argue that TAIL-RECURSIVE-QUICKSORT(A,1,A.length) correctly sorts array A.

- Looking at the quicksort algorithm

```
QUICKSORT(A, p, r)
if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)
```

In this algorithm, the initial function call is done with QUICKSORT(A, 1, A.length). And within each call, QUICKSORT is called recursively twice. Once from p to q-1(which is the left subarray) and then from q+1 to r(which is from the right subarray), hence covering the entire array.

Now, if we look at the TAIL-RECURSIVE-QUICKSORT:

```
TAIL-RECURSIVE-QUICKSORT(A,p,r)
while p < r
        //Partition and sort left subarray
        q = PARTITION(A,p,r)
        TAIL-RECURSIVE-QUICKSORT(A,p,q-1)
        p = q+1
```

From the algorithm, it can be observed that its covering the entire array and recursively calling TAIL-RECURSIVE-QUICKSORT on the array first from p to q-1 and then assigning the value of q+1 to p. This way, it also recursively sorts the right subarray, from q+1 to r. In the process, the algorithm removes the extra recursion. To make sure that it doesn't go into an infinite loop of recursions, the while loop terminates when p reaches the end of the array.

4.b: Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

- Stacks are used for recursive-procedures to contain pertinent information, including the parameter values. Assuming that for each function call the parameter values are pushed to a stack and when the procedure ends, the value is popped out of the stack.

For the above algorithm, the stack depth will equal $\Theta(n)$ when the entire array is sorted, at this point the subproblems will be divided into a 1 single element subproblem and another subproblem with all remaining elements. This will occur when p >= r and the while loop terminates. At this point the stack depth will equal $\Theta(n)$.

4.c: Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$ Maintain the is $O(n.\lg n)$ expected running time of the algorithm.

⇨ The code can be modified in the following manner:

```
TAIL-RECURSIVE-QUICKSORT(A,p,r)
while p < r
        //Partition and sort left subarray
        q = PARTITION(A,p,r)
if(q – p +1 >= r –q)
        TAIL-RECURSIVE-QUICKSORT(A,p,q-1)
Else
        TAIL- RECURSIVE-QUICKSORT(A,q+1,r)
p = q+1
```

To avoid iterating through the entire array in case of redundant scenarios, we can reduce the stack depth to $\Theta(\lg n)$ by calling the function recursively on the larger subproblem. This ways we avoid more recursive calls.

5. Purpose: Practice algorithm design and the use of data structures. This problem was an interview question! Consider a situation where your data is almost sorted—for example you are receiving time-stamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time-stamps. To simplify this problem, assume that each time-stamp is an integer, all time-stamps are different, and for any two timestamps, the earlier time-stamp corresponds to a smaller integer than the later timestamp. The time-stamps arrive in a stream that is too large to be kept in memory completely. The time-stamps in the stream are not in their correct order, but you know that every time-stamp (integer) in the stream is at most thousand positions away from its correctly sorted position. Design an algorithm that outputs the timestamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time-stamps processed. Tip: map the problem to a data structure covered in class.

⇨ The pseudo code for the algorithm is as follows:

```
inputStream = [t1,t2,t3,.....t4]
Initialize PriorityQueue pq
for(i <- 1 to 1000)
        pq.add(inputStream[i]);

DISPLAY-TIMESTAMP(pq, i)
        if i < inputStream.length
                HEAP-SORT(pq)
                //Print and remove the smallest element in the queue.
                print pq.remove();
                //Increment i by 1.
                i = i+1;
                pq.add(inputStream[i]);
                //Add another element from the timestamp stream and recursively display the next
                timeStamp in ascending order
                DISPLAY-TIMESTAMP(pq, i);

//This function sorts the Heap pq.
HEAPSORT(A)
        BUILD-MAX_HEAP(A)
        while heap not empty do
                exchange A[1] with A[n]
                A.heap-size = A.heap-size -1
                MAX-HEAPIFY(A,1)

//This function builds a Max-Heap by calling Heapify iteratively and sorting each subtree.
BUILD-MAX-HEAP(A)
        A.heap-size = A.length
        for i = Math.FLoor(A.length/2) downto 1
                MAX-HEAPIFY(A,i)

//This function Heapifies a specific subtree
MAX-HEAPIFY(A,i)
        l = LEFT(i)
        r = RIGHT(i)
        if l <= A.heap-size and A[l] > A[i]
                largest = l
        else
                largest = i
        if r <= A.heap-size and A[r] > A[largest]
                largest = r
        if largest != i
                exchange A[i] with A[largest]
                MAX_HEAPIFY(A,largest)

LEFT(i)
        return 2*i

RIGHT(i)
        return 2*i+1
```

RUNNING TIME:

The running time complexity of the above algorithm is as follows:

We know that the running time of a heap-sort is n.lg|n|. In our case we know the size of the heap to be always less than equal to 1000.
Therefore:

$$n.\lg|n| = 1000.\lg|1000|$$

$$\lg|1000| \cong 10$$

$$1000 * 10 = 10{,}000$$

So, each time the sorting is done for 1000 elements, the running time comes out to be 10,000. Therefore, for 'n' timeStamps, the running time complexity comes out to be:

$$T(n) \leq 10{,}000 * n$$

$T(n) \in O(n)$ for c = 10,000.

Hence, the algorithm specified above sorts an input stream of timeStamps and displays them in a sorted manner. It takes a maximum memory space of ~ 1000(space complexity ~ 1000), which is a constant amount of storage.