Name: Saidheeraj Poosarla ASU ID: 1219683413

FOUNDATIONS OF ALGORITHMS – CSE 551 – ASSIGNMENT 2

1. The Fibonacci series can be computed as follows,

F(n) =F(n−1) +F(n−2)

In class, we showed how this can be done in O(log n) computation time. Now suppose that the definition is changed in the following way,

F′(n) = F′(n−1) +F′(n−2) +F′(n−3) +F′(n−4)

Can F′(n) be computed in O(log n)? If yes, please show how it can be done. If no, show a counter example where this fails. Please provide your rationale for both. Assume that F′(0) = 0, F′(1) = 1, F′(2) = 1, F′(3) = 1.

Solution:

Yes, F'(n) can be computed in O(log n).

The Fibonacci series F(n) can be computed in O(log n) time. But now, the definition of the Fibonacci series has been changed to

F′(n) = F′(n−1) +F′(n−2) +F′(n−3) +F′(n−4)

Finding out the computation time of F’(n) by solving in an identical way as discussed in the class,

$$[F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n)]$$
$$= [F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n-1)+F'(n-2)+F'(n-3)+F'(n-4)]$$
$$= [F'(n-4)\ \ F'(n-3)\ \ F'(n-2)\ \ F'(n-1)] * \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Assume A = $\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$

Hence,

$$[F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n)] = [F'(n-4)\ \ F'(n-3)\ \ F'(n-2)\ \ F'(n-1)] * A$$

Now, this can be further simplified till we reach F′(0), F′(1), F′(2), F′(3).

$$
\begin{aligned}
&[F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n)] \\
&= [F'(n-4)\ \ F'(n-3)\ \ F'(n-2)\ \ F'(n-1)] * A \\
&= [F'(n-5)\ \ F'(n-4)\ \ F'(n-3)\ \ F'(n-2)] * A^2 \\
&= [F'(n-6)\ \ F'(n-5)\ \ F'(n-4)\ \ F'(n-3)] * A^3 \\
&= [F'(n-7)\ \ F'(n-6)\ \ F'(n-5)\ \ F'(n-4)] * A^4
\end{aligned}
$$

And finally, we arrive at

$$[F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n)] = [F'(0)\ \ F'(1)\ \ F'(2)\ \ F'(3)] * A^{n-3}$$

$$[F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n)] = [0\ \ 1\ \ 1\ \ 1] * A^n * A^{-3}$$

A is a 4 x 4 matrix and since A^n can be calculated in O(log n) time as discussed in the class, $F'(n)$ can also be computed in O(log n) time.

2. Consider the following problem. You're given an array A consisting of n integers A[1], A[2], ..., A[n]. You'd like to output a two-dimensional n×n array B in which B[i, j] (for i < j) contains the sum of array entries A[i] through A[j], i.e., the sum A[i] +A[i+ 1] +...+A[j]. (The value of array entry B[i, j] is left unspecified whenever i≥j, so it doesn't matter what is output for these values.)A simple algorithm to solve this problem illustrated in Algorithm 1:

```
For i=1, 2, ..., n
    For j = i+1, i+2, ..., n
        Add up array entries A[i] through A[j]
        Store the result in B[i, j]
    End for
End for
```

I) For some function f that you should choose, give a bound of the form O(f(n)) on the running time of this algorithm on an input of size n(i.e., a bound on the number of operations performed by the algorithm).

Solution:

The outer for loop goes for n times and for each outer loop iteration, the inner loop iterates up to n times with a combined computation time of O(n^2). Moreover, summing up the array elements A[i] through A[j] takes O(n) time as well because the algorithm has to iterate through all the array elements from index i to index j and then add them up. ***Hence, the total computation time required for the algorithm is O(n^3).***

II) For this same function f, show that the running time of the algorithm on an input of size n is also Ω(f(n)). (This shows an asymptotically tight bound of Θ(f(n)) on the running time.)

Solution:

Outer loop iterates from 1 to n (n times).

Total number of iterations of the inner loop: (n-1) + (n-2) + ... + 1 + 0 = (1/2) (n-1) n

Number of operations required for adding the array elements for n iterations of inner loop:

In case of i = 1:

Number of addition operations = 1 + 2 + ... + n-1 = (1/2)(n-1)(n)

In case of i = 2:

Number of addition operations = 1 + 2 + ... + n - 2 = (1/2)(n-2)(n-1)

In case of i = 3:

Number of addition operations = 1 + 2 + ... + n - 3 = (1/2)(n-3)(n-2)

In case of i = k:

Number of addition operations = 1 + 2 + ... + n - k = (1/2)(n - k)(n -k + 1)

And in case of i = n-2:

Number of addition operations = 1 + (n - (n - 2)) = 1 + 2

In case of i = n/2:

Number of addition operations = 1 + 2 + ... + (n - n/2) = (1/2)(n/2)((n/2)+1) = $n^2/8 + n/4 >= n^2/8$

For all values of i <= n/2, at least $n^2/8$ addition operations take place and there will be n/2 such iterations with i <= n/2.

Therefore,

T(n) >= (n/2 iterations satisfying i <= n/2) (at least n^2 / 8 addition operations in each corresponding iteration)

T(n) >= n^3 / 16

T(n) >= (1/16) * n^3 for all n_o > 1 and e = 1/16

Therefore, the algorithm is lower bounded by n^3 and hence the running time of the algorithm is also Ω(f(n)).

III) Although the algorithm you analysed in parts (a) and (b) is most natural way to solve problem-after all, it just iterates through the relevant entries of the array B, filling in a value for each - it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time O(g(n)), where $\lim_{n\to\infty} g(n)/f(n) = 0$.

Solution:

Algorithm:

Set a new variable addvalue to A[i] in each iteration of i. Next, in each iteration of j, store the sum of A[i] to A[j] in addvalue, so for each incremented value of j, you just have to add A[j] to the current value of addvalue, since the current value in addvalue is A[i] + A[i+1] + ... + A[j-2] + A[j-1], so adding A[j] to the current addvalue will give the required sum of A[i] + A[i+1] + ... + A[j]. Adding of A[j] to the current addvalue can be done in constant time, and hence now the inner loop takes constant time. ***Therefore, the computation time for the algorithm is due to the i and j loops only and hence the complexity gets reduced to O(n^2) which is much better than O(n^3).***

Pseudo-code of algorithm:

```
addvalue = 0

For i=1, 2, ..., n

    addvalue = A[i]

     For j = i+1, i+2, ..., n

        addvalue = addvalue + A[j]
        Store the current value of addvalue in B[i, j]
    End for
End for
```

3. Design an algorithm to compute the 2nd smallest number in an unordered (unsorted) sequence of numbers {a1, a2, ..., an}in $n + [\log_2(n)] - 2$ comparisons in the worst case. If you think such an algorithm can be designed, then show how it can be done. If your answer is no, then explain why it cannot be done.

Solution:

Consider the list of n numbers as a tournament with n participants. Now, we can have a tournament rule such that the smallest of the two numbers wins. Now, we can use the FindMin algorithm below to find out the smallest of all the numbers.

FindMin Algorithm:

1. Split the data into two sets of equal sizes S1 and S2.
2. FindMin(S1): Suppose min1 is the minimum value of the set S1.
3. FindMin(S2): Suppose min2 is the minimum value of the set S1.
4. Return min (min1, min2).

Now, the number of comparisons for n numbers is $C(n) = 2C(n/2) + 1$. As discussed in the class, this comes out to be n-1 comparisons.

Hence, it takes (n-1) comparisons to find out the smallest number from the list of n unordered numbers.

Now, in this type of a tournament, the best player must face the second-best player which implies that the smallest number is guaranteed to face the second smallest number. As we know that the second smallest number has played against the smallest number, we can consider only the subtree which contains the smallest number and neglect the remaining subtree. Now that we have established the smallest number from the above FindMin algorithm, we have to analyse the subtree from which the smallest number has originated because the best player could have played the second-best player anywhere in his/her path to the tournament win and not necessarily in the finals. This computation to find the second smallest number after knowing the smallest number takes $[\log_2(n)] - 1$ comparisons.

Finally, the total number of comparisons needed to find out the second smallest number is the sum of the number of comparisons required to find out the smallest number and the number of comparisons required to find out the second smallest number using the smallest.

Therefore, the total number of comparisons required to find out the second smallest number from a set of n unordered numbers = $n - 1 + [\log_2(n)] - 1 = n + [\log_2(n)] - 2$

Name: Saidheeraj Poosarla ASU ID: 1219683413

4. Let n= 2p, V= (v_1, ..., v_n), W= (w_1, ..., w_n). Then we can compute the vector product VW by the formula:

$$\sum_{1\le i\le P}[(v_{2i-1}+\omega_{2i})\times(v_{2i}+w_{2i-1})]-\sum_{1\le i\le p}(v_{2i-1}\times v_{2i})\ -\sum_{1\le i\le p}(w_{2i-1}\times w_{2i})$$

which requires 3n/2 multiplications. Show how to use this formula for the multiplication of two n×n matrices giving a method which requires $n^3/2 + n^2$ multiplications rather than the usual n^3 multiplications.

Solution:

Given: n = 2p,

V = (v_1, v_2, …., v_n) , W = (w_1, w_2, ….., w_n)

vector product VW =

$$\sum_{1\le i\le P}[(v_{2i-1}+\omega_{2i})\times(v_{2i}+w_{2i-1})]-\sum_{1\le i\le p}(v_{2i-1}\times v_{2i})\ -\sum_{1\le i\le p}(w_{2i-1}\times w_{2i})$$

Let us consider two matrices A_1 and A_2 of the order n x n.

Suppose $A_1 = \begin{pmatrix} R_1 \\ R_2 \\ . \\ R_n \end{pmatrix}$

where R_1, R_2, ……., R_n are representations for Row 1, Row 2, ……., Row n.

Suppose $A_2 = \begin{pmatrix} C_1 & C_2 & .... & C_n \end{pmatrix}$

where C_1, C_2, ………, C_n are the representations for Column 1, Column 2, ……, Column n.

Now consider a third matrix B = A_1 x A_2

$$B = \begin{pmatrix} R_1C_1 & R_1C_2 \,..... & R_1C_n \\ R_2C_1 & R_2C_2 \,...... & R_2C_n \\ . & . & . \\ R_nC_1 & R_nC_2 \,...... & R_nC_n \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \,............ & B_{1n} \\ B_{21} & B_{22} \,........... & B_{2n} \\ & . & \\ B_{n1} & B_{n2} \,............ & B_{nn} \end{pmatrix}$$

The first element in B is R_1C_1 (B_{11}).

Since its given in the question that it takes 3n/2 computations for the multiplication of two vectors V and W, it takes 3n/2 multiplications to compute the first element of the first row in B.

Now, for the calculation of the other values in the first row i.e, B_{12}, B_{13}, ………, B_{1n}, it requires only 2n/2 multiplications as the second term in the formula is already calculated while computing B_{11}.

Therefore, the total number of multiplications for the first row becomes,

$$\frac{3n}{2} + (n-1)\frac{2n}{2}$$

Similarly, for the calculation of B_{21}, the third term in the formula is already computed during the calculation of B_{11} and hence it needs only 2n/2 multiplications.

And for the remaining values of the second row, the second and the third terms in the formula are already computed and hence will require only n/2 multiplications.

Therefore, total number of multiplications for the second row becomes,

$$\frac{2n}{2} + (n-1)\frac{n}{2}$$

Similar is the case for all the rows except for the first one and hence the total number of multiplications required for all the rows from 2 to n is

$$\left[\frac{2n}{2} + (n-1)\frac{n}{2}\right](n-1)$$

Total number of multiplications for all the n rows = $\frac{3n}{2} + (n-1)\frac{2n}{2} + \left[\frac{2n}{2} + (n-1)\frac{n}{2}\right](n-1)$

= $\frac{3n}{2} + n^2 - n + n^2 + \frac{n^3}{2} - \frac{n^2}{2} - n - \frac{n^2}{2} + \frac{n}{2}$

= $\frac{n^3}{2} + n^2$

Hence, the total number of multiplications needed to multiply two n x n matrices using the mentioned formula is

$$\boldsymbol{n^2 + \frac{n^3}{2}}$$

5. Find the solution to the following recurrence relation:

$T(n) = 16T(n/4) + n^2$,

For n a power of 4 (or $n = 4^k$) and $n > 1$.

Solution:

The recurrence relation is given as $T(n) = 16T(n/4) + n^2$ and $T(1) = 1$ where $n = 4^k$, $n > 1$.

$T(n) = 16T(n/4) + n^2$

$T(n) = 16 [16T(n/16) + (n/4)^2] + n^2$

$T(n) = 16^2T(n/16) + 2n^2$

$T(n) = 16^2 [16T(n/64) + (n/16)^2] + 2n^2$

$T(n) = 16^3T(n/64) + 3n^2$

When continued till T(1), we get

$T(n) = 16^kT(1) + kn^2$

Since T(1)=1,

$T(n) = 16^k + kn^2$

$T(n) = n^2 + n^2\log_4 n$ (since $n = 4^k$)

Therefore, $T(n) = n^2 [1 + \log_4 n]$

6. Find the solution to the following recurrence relation:

$T(n) = 3^nT(n/2)$,

$T(1) = 1$

For n a power of 2 (or $n = 2^k$) and $n > 1$.

Solution:

The recurrence relation is given as $T(n) = 3^nT(n/2)$ and $T(1) = 1$ where $n = 2^k$, $n > 1$.

$$T(n) = 3^{n}T(n/2)$$

$$T(n) = 3^{n} [\, 3^{n/2}T(n/4) \,]$$

$$T(n) = 3^{n}3^{n/2}3^{n/4}T(n/8)$$

When continued till T(1), we get

$$T(n) = 3^{n}3^{n/2}3^{n/4}\ldots\ldots..3^{n/(2\wedge(k-1))}T(1)$$

Since T(1) = 1,

$$T(n) = 3^{n(1+1/2+1/4+\ldots\ldots..1/2\wedge(k-1))}$$

$$T(n) = 3^{n[1-(1/2)\wedge k]/[1-1/2]}$$

$$T(n) = 3^{2n[1-(1/2)\wedge k]}$$

Since $n = 2^{k}$

$$T(n) = 3^{(2\wedge(k+1))-2}$$

$$T(n) = 3^{2n-2}$$

Therefore, T(n) = 9^{n-1}