CS 3340 Assignment 1

Matvey Skripchenko 250899673

February 2nd, 2021

Question 1. A complete binary tree is defined inductively as follows. A complete binary tree of height 0 consists of 1 node which is the root. A complete binary tree of height $h + 1$ consists of two complete binary trees of height h whose roots are connected to a new root. Let T be a complete binary tree of height h. Prove that the number of leaves of the tree is 2^h and the size of the tree (number of nodes T) is $2^{h+1} - 1$.

Proof.

Firstly, we prove that the complete binary tree of height h has 2^h leaves in a binary tree.

Induction base: if $h = 0$, then $2^h = 2^0 = 1$.
Induction hypothesis: assume that a tree of height h has 2^h leaf nodes.

Based on the given definition of the complete binary tree, there should exist 2 complete binary subtrees, each of height h. These 2 subtrees will join the root node what will then form a complete binary tree of height $h + 1$. Then, we have 2 complete binary subtrees of height h, where each will have 2^h leaf nodes.

Consider the height of the complete binary tree to be $h + 1$, then the resulting number of leaf nodes coming from the 2 complete binary subtrees will give:

$$2 * 2^h \ = 2^{h+1}$$

where 2^{h+1} is number of leaf nodes.

Therefore, by induction, we proved that for a complete binary tree of height $h + 1$, the number of leaf nodes is 2^{h+1}.

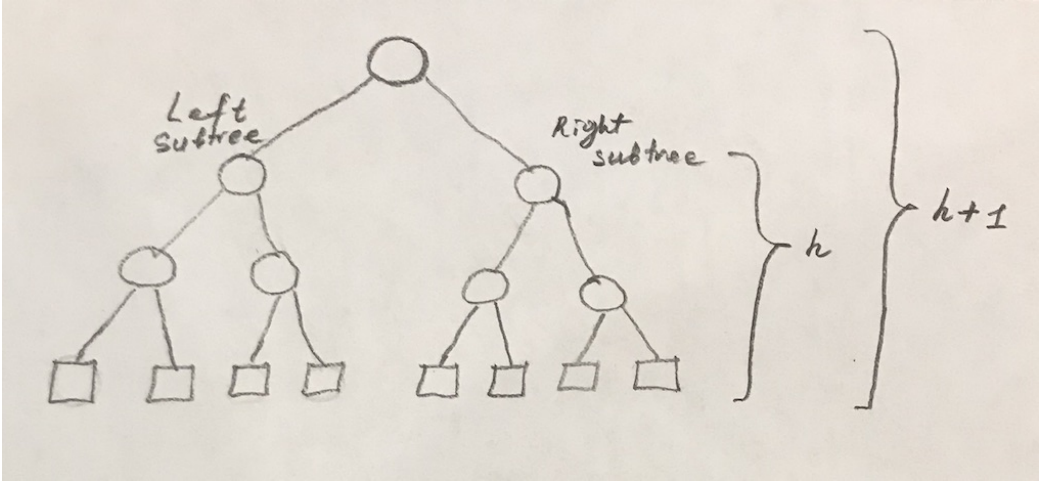
Secondly, we prove that the complete binary tree of height h has $2^{h+1} - 1$ total nodes.

Induction base: if $h = 0$, then the total number of nodes in the complete binary tree will be $2^{h+1} - 1 = 2^{0+1} - 1 = 2^1 - 1 = 1$.
Induction hypothesis: assume that for the height h, the complete binary tree has $2^{h+1} - 1$ total nodes in the binary tree.

So, now we need to prove that a binary tree of height $h+1$ will have the following number of nodes:

$$2^{(h+1)+1} - 1 = 2^{h+2} - 1$$

To do this, let us consider the complete binary tree of $h = 3$, which can be seen below.



Because this is a complete binary tree, we can say more generally that the the height is $h + 1$. If we look at this recursively, then the complete binary tree with $h > 0$ will have 2 complete binary subtrees, each of height $h - 1$. If that is the case, then height of the left/right subtree will be h.

Also, if we consider the left or right subtree separately, then the total number of total nodes for each such subtree must also be $2^{h+1} - 1$, i.e., $2^{2+1} - 1 = 2^3 - 1 = 7$ total nodes, therefore true.

Now, from the induction hypothesis, the condition for total number of nodes $2^{h+1} - 1$ must hold at each and every state of the tree. Thus, the induction base condition must hold. Therefore we will do the following:

$$\textit{root node + total nodes of right subtree + total nodes of left subtree}$$

$$= 1 + (2^{h+1} - 1) + (2^{h+1} - 1)$$

$$= 1 + 2 * (2^{h+1} - 1)$$

$$= 1 + 2 * 2^{h+1} - 2 * 1$$

$$= 1 + 2^{h+2} - 2$$
$$= 2^{h+2} - 1$$

Therefore, we showed that $h + 1$ holds for all.

Thus, we have proved that for all $h \geq 0$, the complete binary tree will have $2^{h+1} - 1$ total nodes.

Question 2. Insertion sort on small arrays in merge sort. Although merge sort runs in $\theta(nlgn)$ worst-case time and insertion sort runs $\theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to **coarsen** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which $n = k$ sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. *Show that insertion sort can sort the n/k sublists, each of length k, in $\theta(nk)$ worst-case time.*

We know that the worst time complexity of insertion sort is $\theta(n^2)$, meaning that for a list with n elements the insertion sort will take max $\theta(n^2)$ time to sort such a list. Thus, the time taken to sort n/k lists with k elements is:

$$= \theta(k^2 * (n/k)$$
$$= \theta(nk)$$

Thus, we have shown that insertion sort can sort n/k sublists, each of length k, in $\theta(nk)$ worst-case time.

b. *Show how to merge the sublists in $\theta(nlg(n/k))$ worst-case time.*

Let us consider the case where 2 lists are used for merging. Then the time it would take is $\theta(n(n/k)) = \theta(n^2/k)$. This is because there are a total of n/k lists to merge and n time would be needed in order to properly copy each element and get the new sorted list.

However, if we are trying to achieve $\theta(nlg(n/k))$ time, then let us consider taking 2 sublists at a time and merge them until a single list is obtained.

So, through divide-and-conquer, the recursion tree can be obtained, where $lg(n/k)$ is the height of the tree, meaning that there are $lg(n/k)$ steps required to complete the process. Every step would require to compare n elements, i.e., $\theta(n)$ time per level. Therefore, the whole process will result in $\theta(nlg(n/k))$.

Thus, we have shown how the sublists can be merged in $\theta(nlg(n/k))$ worst-case time.

c. *Given the modified algorithm runs in $\theta(nk + nlg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as the standard merge sort, in terms of $\theta - notation$.*

We need to show that in order for the modified algorithm to have the same asymptotic run time as the non-modified merge sort, then $\theta(nk+nlg(n/k)) = \theta(nlg(n))$ must hold. Essentially, we do not want k to run faster than lgn asymptotically, because otherwise we would get a worse asymptotic time than $\theta(nlg(n))$ due to the nk term.

Let us assume that $k = \theta(lgn)$ and do the following:

$$\begin{aligned}\theta(nk + nlg(n/k)) &= \theta(nk + nlgn - nlgk)\\ &= \theta(nlgn + nlgn - nlg(lgn))\\ &= \theta(2nlgn - nlg(lgn))\\ &= \theta(nlgn)\end{aligned}$$

The reason why we got $\theta(nlgn)$ is because for larger n, $lg(lgn)$ is significantly smaller compared to lgn.

Question 3. Inversions. Let *A[1..n]* be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. *List the five inversions of the array $\langle 2, 3, 8, 6, 1\rangle$.*

So, for the given array $\langle 2, 3, 8, 6, 1\rangle$ we have the following indecies $\langle 0, 1, 2, 3, 4\rangle$, respectively. Thus, the 5 inversions of A will be the pairs:

$$(0, 4), (1, 4), (2, 4), (2, 3), (3, 4)$$

as they satisfy the given condition.

b. *What array with elements from the set* $\langle 1, 2, ..., n \rangle$ *has the most inversions? How many does it have?*

Such array will have the elements in reverse sorted order and look something like this:

$$\langle n, n-1, ..., 2, 1 \rangle$$

The array above will have the most inversions because it will have n distinct elements sorted in reverse and, then, we will get an inversion for every distinct pair (i, j).

Now, let us find that number of inversions. So, if we look at the $0th$ index or the first element of this array and choose it to be first element of (i, j), then we will get $n - 1$ inversions. Same will hold for the second element and, therefore, $n - 2$ inversions. Same will hold for the third element and, therefore, $n - 3$ inversions. We keep going until reaching the last element n of that array, which will give no inversions since $n - n = 0$. Thus we can find the *total number of inversions*:

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

c. *What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.*

Knowing the nature of the insertion sort algorithm and that it usually takes $O(n^2)$ time to sort n items. Now, let us take into account our findings from **b.** and consider the *inner while loop* part of the insertion sort where most of the array manipulations are made. It can be noticed that if we apply this to the reverse sorted array, then the more inversions we have, the more times the *inner while loop* of the insertion sort algorithm will run. Thus, the number of inversions of the array and the running time of the insertion sort algorithm have direct proportionality, i.e., more inversions give longer run time and less inversions give shorter run time.

d. *Give an algorithm that determines the number of inversions in any permutation on n elements in* $\theta(nlgn)$ *worst-case time. (Hint: Modify merge sort.)*

```
COUNT-INVERSIONS(A, p, r)
   if p < r
      q = floor((p + r) / 2)
      left = COUNT-INVERSIONS(A, p, q)
      right = COUNT-INVERSIONS(A, q + 1, r)
      inversions = MERGE-INVERSIONS(A, p, q, r) + left + right
      return inversions

MERGE-INVERSIONS(A, p, q, r)
   n1 = q - p + 1
   n2 = r - q
   let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
   for i = 1 to n1
      L[i] = A[p + i - 1]
   for j = 1 to n2
      R[j] = A[q + j]
   L[n1 + 1] = ∞
   R[n2 + 1] = ∞
   i = 1
   j = 1
   inversions = 0
   for k = p to r
      if L[i] <= R[j]
         A[k] = L[i]
         i = i + 1
      else
         inversions = inversions + n1 - i + 1
         A[k] = R[j]
         j = j + 1
   return inversions
```

Question 4. Relative asymptotic growths. Indicate, for each pair of expressions (A, B) in the table below, whether A is *O, o,* Ω*,* ω*, or* Θ *of*

B. Assume that $k \geq 1$, $\epsilon > 0$ and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	o	Ω	ω	Θ
$lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{sin(n)}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
n^{lgc}	c^{lgn}	yes	no	yes	no	yes
$lg(n!)$	$lg(n^n)$	yes	no	yes	no	yes

Question 5. *Parameter-passing costs.* Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time = $\Theta(1)$.
2. An array is passed by copying. Time = $\Theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time = $\Theta(q-p+1)$ if the subarray $A[p..q]$ is passed.

a. *Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.*

1. For when array is passed by pointer:

Recurrence is:

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

Master theorem can be used to solve, where $a = 1, b = 2$ and $f(n) = c$. Then:

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

Case 2 of master theorem can be used as $f(n) = \Theta(n^{\log_b a}) = \Theta(1) = c$.

Thus, we have our solution:

$$T(n) = \Theta(\log_2 n)$$

2. For when array is passed by copying:

Recurrence is:

$$\begin{aligned} T(n) &= T(n/2) + cN \\ &= T(n/4) + 2cN \\ &= T(n/8) + 3cN \\ &= \sum_{i=0}^{\log_2 n - 1} (2^i c(N/2^i) \\ &= cN \log_2 n \\ &= \Theta(n \log_2 n) \end{aligned}$$

Thus, we have our solution:

$$T(n) = \Theta(n \log_2 n)$$

3. For when array is passed by copying only the subrange that might be accessed by the called procedure:

Recurrence is:

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

Master theorem can be used, where $a = 1, b = 2$ and $f(n) = cn = n$.

Then:

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

So, we have that $f(n) = \Omega(n^{\log_b a + \epsilon}), \epsilon = 1$ as well as $1(n) \leq cn, c > 1$.

Thus, Case 3 of master theorem can be used and our solution is:

$$T(n) = \Theta(n)$$

b. *Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.*

1. For when array is passed by pointer:

Recurrence is:

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

Master theorem can be used to solve, where $a = 2, b = 2$ and $f(n) = cn = n$. Then:

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

Case 2 of master theorem can be used as $f(n) = \Theta(n^{\log_b a}) = n$.

Thus, we have our solution:

$$T(n) = \Theta n(\log_2 n)$$

2. For when array is passed by copying:

Recurrence is:

$$\begin{aligned}
T(n) &= 2T(n/2) + cn + 2N \\
&= 4T(n/4) + cn + 4N + 2c(n/2) \\
&= 8T(n/8) + 2cn + 8N + 4c(n/4) \\
&= \sum_{i=0}^{\log_2 n - 1} (cn + 2^i N) \\
&= \sum_{i=0}^{\log_2 n - 1} (cn) + \sum_{i=0}^{\log_2 n - 1} (2^i) \\
&= cn \log_2 n + N \frac{2^{\log_2 n} - 1}{2 - 1} \\
&= cn \log_2 n + nN - N \\
&= \Theta(nN) \\
&= \Theta(n^2)
\end{aligned}$$

Thus, we have our solution:

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

3. For when array is passed by copying only the subrange that might be accessed by the called procedure:

Recurrence is:

$$\begin{aligned}
T(n) &= 2T(n/2) + cn + 2(n/2) \\
&= 2T(n/2) + cn + n \\
&= 2(n/2) \\
&= 2T(n/2) + n(c + 1)
\end{aligned}$$

Master theorem can be used to solve, where $a = 2, b = 2$ and $f(n) = n(c + 1) = n$. Then:

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

Case 2 of master theorem can be used as $f(n) = \Theta(n^{\log_b a}) = n$.

Thus, we have our solution:

$$T(n) = \Theta(n \log_2 n)$$

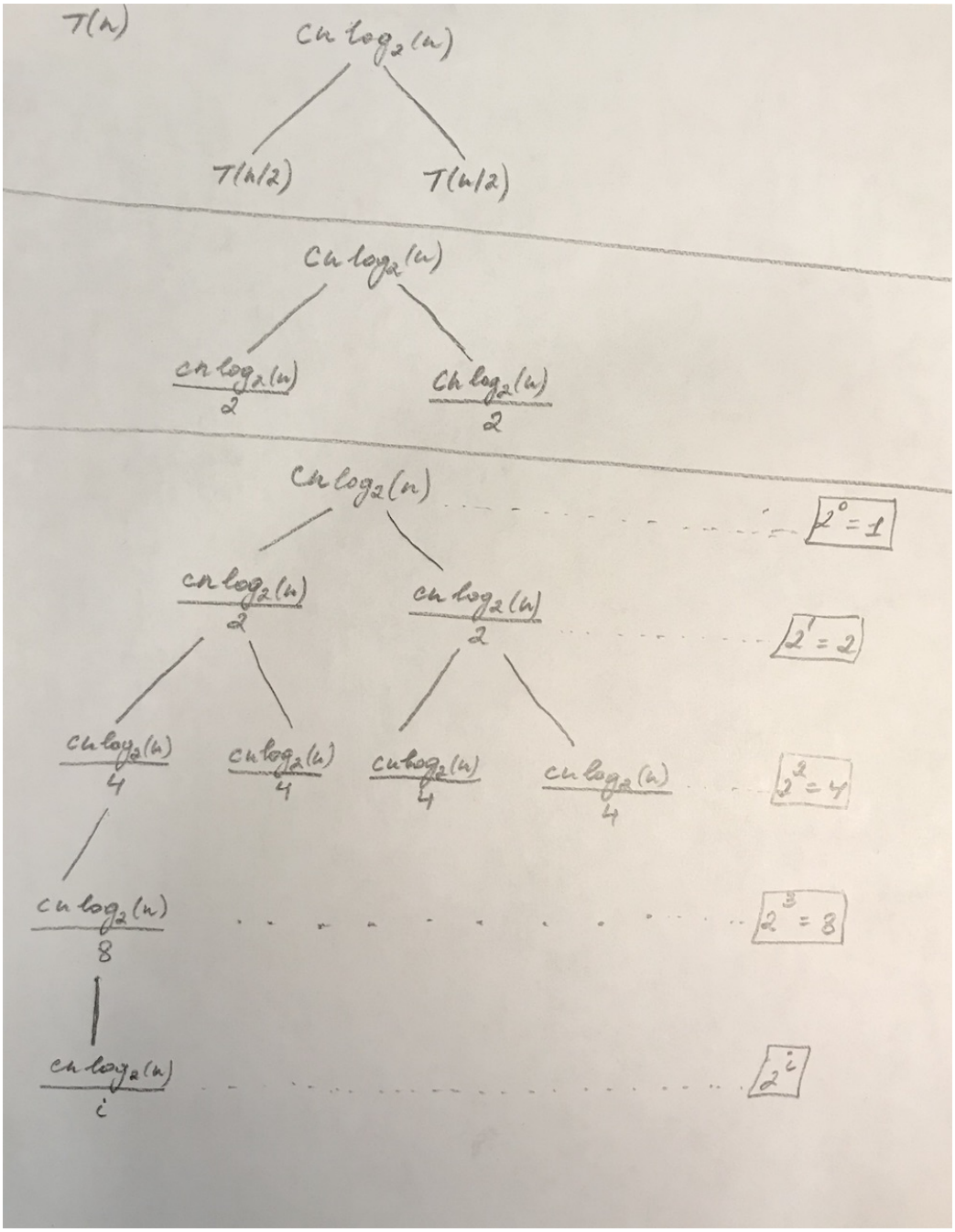
Question 6. Suppose that the running time of a recursive program is represented by the following recurrence relation:

$$T(2) \leq c$$
$$T(n) \leq 2T(n/2) + cn \log_2(n)$$

Determine the time complexity of the program using recurrence tree method and then prove your answer.

Solution.

The obtained recurrence tree can be seen below:


T(n)
$cn\log_2(n)$
T(n/2)
T(n/2)
$cn\log_2(n)$
$\frac{cn\log_2(n)}{2}$
$\frac{cn\log_2(n)}{2}$
$cn\log_2(n)$
$2^0 = 1$
$\frac{cn\log_2(n)}{2}$
$\frac{cn\log_2(n)}{2}$
$2^1 = 2$
$\frac{cn\log_2(n)}{4}$
$\frac{cn\log_2(n)}{4}$
$\frac{cn\log_2(n)}{4}$
$\frac{cn\log_2(n)}{4}$
$2^2 = 4$
$\frac{cn\log_2(n)}{8}$
$2^3 = 8$
$\frac{cn\log_2(n)}{i}$
2^i

Now, we will sum each level:

$$
\begin{aligned}
&= cn\log_2(n) + 2\frac{cn\log_2(n)}{4} + 4\frac{cn\log_2(n)}{8} + 8\frac{cn\log_2(n)}{64} + ... \\
&= cn\log_2(n) + \frac{cn\log_2(n)}{2} + \frac{cn\log_2(n)}{4} + \frac{cn\log_2(n)}{8} + ... \\
&= cn\log_2(n)(1 + 1/2 + 1/4 + 1/8 + ...) \\
&= cn\log_2(n)(\frac{1}{1 - 1/2}) \\
&= 2cn\log_2(n) \\
&= O(cn\log_2(n))
\end{aligned}
$$

Therefore, the time complexity is $O(cn\log_2(n))$.

Question 7.

a.

```
//
// Created by Matvey Skripchenko.
// Student number: 250899673
//

#include <iostream>

using namespace std;

int fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main ()
{
    for (int i=0;i<=10;i++)
        cout<<fibonacci(5*i)<<endl;
    return 0;
}
```

b.

```cpp
// Created by Matvey Skripchenko
// Student number: 250899673
//

#include <iostream>
#include <cstdlib>
#include <cstring>

// defining the constant
#define var unsigned long long

using namespace std;


// This function will multiply 2 matrices F and L
// and stores result in F
void matrix_mult(var F[2][2], var L[2][2]){
    var a =  F[0][0] * L[0][0] + F[0][1] * L[1][0];
    var b =  F[0][0] * L[0][1] + F[0][1] * L[1][1];
    var c =  F[1][0] * L[0][0] + F[1][1] * L[1][0];
    var d =  F[1][0] * L[0][1] + F[1][1] * L[1][1];

    F[0][0] = a;
    F[0][1] = b;
    F[1][0] = c;
    F[1][1] = d;

}

// In this function, F is raised to the power of n
// and stores the result in F
void matrix_pow(var F[2][2], int n)
{
    if (n == 0 || n == 1)
        return;
    // initializing
    var L[2][2] = {{1,1},{1,0}};

    // multiply M n-1 times
    for(int i = 2; i <= n; i++){
        matrix_mult(F,L);
    }
}

// final function that will return Fib numbers
var fibonacci(var n)
{
    // initializing
    var F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    matrix_pow(F, n - 1);
    return F[0][0];

}

int main()
{
    for(int i=0;i<25;i++)
        cout<<fibonacci(i*20)<<endl;
    return 0;
}
```

c.

```
// Created by Matvey Skripchenko.
// Student number: 250899673
//

#include <iostream>
#include <cstdlib>
#include <cstring>

// defining the constant
#define var unsigned long long

using namespace std;


// This function will multiply 2 matrices F and L
// and stores result in F
void matrix_mult(var F[2][2], var L[2][2]){
    var a =  F[0][0] * L[0][0] + F[0][1] * L[1][0];
    var b =  F[0][0] * L[0][1] + F[0][1] * L[1][1];
    var c =  F[1][0] * L[0][0] + F[1][1] * L[1][0];
    var d =  F[1][0] * L[0][1] + F[1][1] * L[1][1];

    F[0][0] = a;
    F[0][1] = b;
    F[1][0] = c;
    F[1][1] = d;

}

// In this function, F is raised to the power of n
// and stores the result in F.
// This function is changed from asn1_b
void matrix_pow(var F[2][2], int n)
{
    if (n == 0 || n == 1)
        return;
    // initializing
    var M[2][2] = {{1,1},{1,0}};
    matrix_pow(F, n / 2);
    matrix_mult(F, F);
    // checking remainder
    if (n % 2 != 0)
        matrix_mult(F, M);
}

// final function that will return Fib numbers
var fibonacci(var n)
{
    // initializing
    var F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    matrix_pow(F, n - 1);
    return F[0][0];

}

int main()
{
    for(int i=0;i<25;i++)
        cout<<fibonacci(i*20)<<endl;
    return 0;
```

d.

After using the bash time command I got about 1min and 26 seconds for asn1(a). For asn1(b) and asn1(c) the times were pretty much the same and about 0.004 seconds. The reason why asn1(a) implementation takes so long is because of a lot of repeated work being done and it has an exponential time complexity. For asn1(b) the time complexity is $O(n)$ and this is much faster/efficient implementation as we n times multiply the matrix to itself what avoids additional work. Same relates to asn1(c), but the method for the power has been modified, where we pretty much halve things down for n what allows for $O(log(n))$ time complexity.

e.

If we are trying to compute $F(50)$ in asn1(a) using int type of 4 bytes, then we would run into integer overflow problem as 4 bytes will not be long enough what would cause an overflow. For asn1(b), $F(500$ is computed precisely, because I am using the unsigned long long integer type what allows to avoid the integer overflow issue.