EECE7205: Fundamental of Computer Engineering
Homework 6

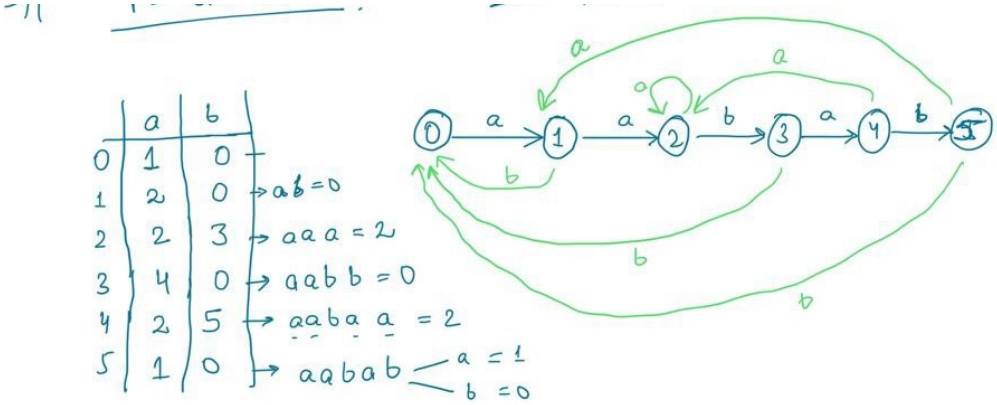
Ankesh Kumar (002208893)

Problem 1

TODO String Matching Problems

Construct the string-matching automaton for the pattern P = "aabab" over the alphabet Σ= a, b.

In this, I first created the Transition table, where the first column is the states, and rest columns as a,b. and filled the table with required state where the approached.
Below figure gives the transition table and also its corresponding created finite-machine automata.



ii. Show the steps of applying the finite automata matcher algorithm to search for P in the text T = "abaaaababa".

Steps in searching P in T using - finite automata matcher algorithm
Pattern P = "aabab"
Text T = "abaaaababa"
Finite Automata Matcher Steps:
In first step, we create a finite automaton that represents the given pattern P.
Then we initialise the state with 0, where no character is yet matched.
We keep on reading the character and make a table of state (Ti) by one by one and transitions between states according to the next character in the input. Each state represents the length of the longest prefix of P that is a suffix of the text read so far.

$T(i)$	-	a	b	a	a	a	a	b	a	b	a
State of $T(i)$	0	1	0	1	2	2	2	3	4	**5**	

As, we see that at 5, we reach end of state and here we are able to find the pattern in the text.
Whenever the automaton reaches the final state (state = length of P), it indicates that the pattern P has been found in the text T. The position in the text where this occurs can be calculated as the current position in the text minus the length of P plus 1.
Upon reaching the final state, note the index in T where P is found.

iii. Implement the suffix array of text T in (ii) and show the step to use the array to search for pattern "aba".

Text T = "abaaaababa"
Pattern to search in T: "aba"
Suffix Array Construction:
A suffix array is a sorted array of all suffixes of a given text. For T, we list all suffixes, sort them alphabetically, and then construct the array of their starting positions in T.
Here we noting the starting positions of each sorted suffix in T, leading to the suffix array.
Creating a suffix array identifying the starting positions of sorted suffixes in T.

j	$T[j:]$
1	abaaaababa
2	baaaababa
3	aaaababa
4	aaababa
5	aababa
6	ababa
7	baba
8	aba
9	ba
10	a

Table 1: List of all suffixes of text T.

Sorted Suffixes Alphabetically

i	$SA[i]$	$T[SA[i]:]$
1	10	a
2	3	aaaababa
3	4	aaababa
4	5	aababa
5	8	aba
6	1	abaaaababa
7	6	ababa
8	9	ba
9	2	baaaababa
10	7	baba

Table 2: Suffixes of T sorted alphabetically with suffix array indices.

While searching for "aba" using the suffix array, we can locate it using a binary search. To follow the pattern, start the binary search. If "aba" is not found or the search space is exhausted, compare it to the middle element of the sorted suffix list, adjust the search range, and continue the process until "aba" is found.

Problem 2

TODO Fibonacci sequence - Dynamic vs Recursive Programming

a) Comment on the running time of each function.

Comparing the running time of each function, we each to the following conclusion :

- **FibonacciD** function uses dynamic programming to calculate the Fibonacci sequence by storing previously calculated Fibonacci numbers in the Temp_Array and accessing them when necessary. The efficiency of FibonacciD is determined by the number of distinct values it calculates. Since each Fibonacci number is computed just once and saved for future reference, the runtime of FibonacciD is directly proportional to n.
- **FibonacciR**: This function calculates the Fibonacci sequence using recursion without any dynamic programming enhancements. It performs repetitive calculations, leading to an exponential time complexity. For every Fibonacci Recursive call, it calls itself recursively twice until it reaches the base case, resulting in numerous function calls and repetitive computations.

b) For each function, find the big O asymptotic notation of its running time growth rate.

Big O asymptotic notation of the running time growth rate:

- Fibonacci Dynamic: The time complexity of Fibonacci Dynamic increases linearly in relation to n, making its asymptotic notation O(n).
- Fibonacci Recursive : The time complexity of Fibonacci Recursive increases exponentially with respect to n, making its asymptotic notation O(2^n)

Implementation screenshot

```
n = 5
 FibonacciD(n) = 5
 FibonacciR(n) = 5akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram/HW6$ ./HW6_Sol2
n = 10 •
 FibonacciD(n) = 55
 FibonacciR(n) = 55akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram/HW6$ ./HW6_Sol2
n = 20
 FibonacciD(n) = 6765
 FibonacciR(n) = 6765akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram/HW6$ ./HW6_Sol2
n = 30
 FibonacciD(n) = 832040
 FibonacciR(n) = 832040akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram/HW6$ ./HW6_Sol2
n = 40
 FibonacciD(n) = 102334155
 FibonacciR(n) = 102334155akuma67@LAPTOP-DCEM5HVC:~/EECE7205CProgram/HW6$
```

Problem 3

TODO Rod-cutting Problem - Dynamic vs Recursive Programming

Solution

Conclusion

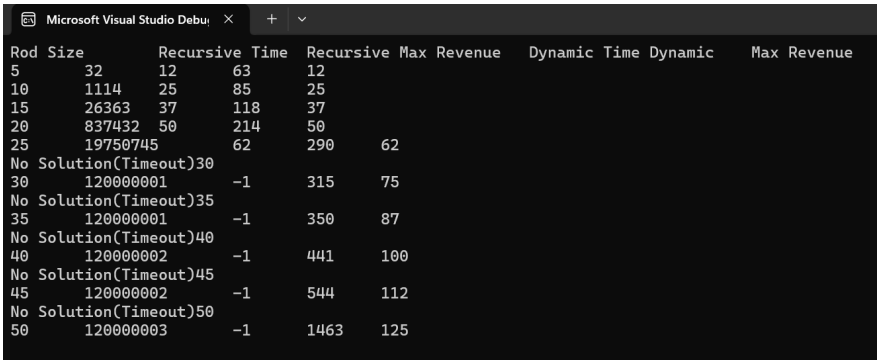
Based on the data, it is clear that using dynamic programming is much more efficient than the recursive approach.

Table 3: Rod Cutting Performance Comparison

Rod Size	Recursive Time	Recursive Max Revenue	Dynamic Time	Dynamic Max Revenue
5	32	12	63	12
10	1,114	25	85	25
15	26,363	37	118	37
20	837,432	50	214	50
25	19,750,745	62	290	62
30	No Solution	–	315	75
35	No Solution	–	350	87
40	No Solution	–	441	100
45	No Solution	–	544	112
50	No Solution	–	1,463	125

The recursive method's time complexity grows exponentially as the size of the rod increases, while the dynamic approach shows a much slower growth rate in computation time. For smaller rod sizes (5 to 20), both methods work well in finding the maximum revenue in a reasonable amount of time.
However, when the rod size goes beyond 30, the recursive approach takes more than 2 mins to find a solution within the given constraints, resulting in a "No Solution" status which shows the computational efficiency.
On the other hand, the dynamic approach continues to yield results even for larger rod sizes, maintaining a manageable computation time and delivering a maximum revenue outcome.

Implementation screenshot



Problem 4

TODO Letter encoding - Huffman Algorithm

Huffman Encoding Tree

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

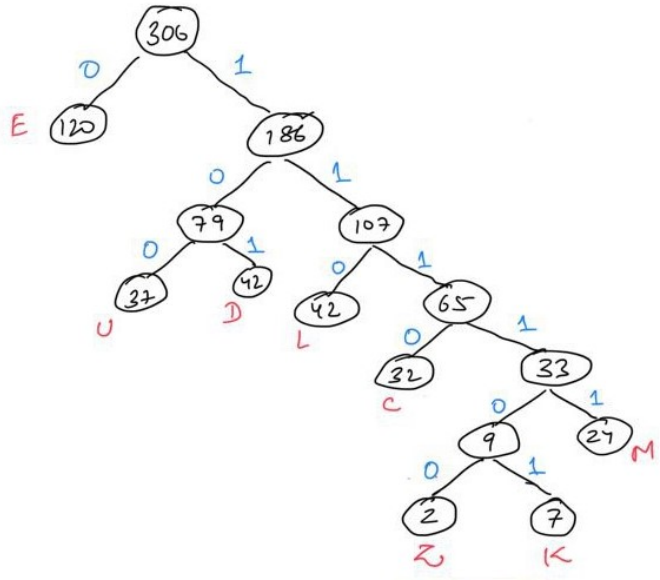
Steps :

Step 1: Extracted two min-priority from the queue:

Step 2: Create a new internal node with these two nodes as children and assign to this new node a frequency equal to the sum of the two nodes' frequencies.

Step 3: Construct the final Huffman Tree according to above steps.

Step 4: Assigned '0' for moving left and '1' for moving right.



Step 5: Traversed from the previous step and formed the binary Huffman codes. Recorded these in a table or map, which will be used for encoding and decoding the data.

Encoding allocated for each character.

Letter	Frequencies	Encoding allotted
Z	2	111100
K	7	111101
M	24	111111
C	32	1110
U	37	100
D	42	101
L	42	110
E	120	0

Table 4: Letter Frequencies and Encodings

This is the submission for the assignment 6.