g) Okan Çiftçi 18303912

$T(n) = 16T(n/4) + n!$ $\quad f(n) = n!$ $\quad n^{\log_b^a} = n^{\log_4^{16}} = n^{\log_4^{4^2}} = n^2$

$f(n) \geq n^{\log_b^a}$ Case 3:

regularity check

$a\,f(n/b) \leq (f(n))$ for some $c < 1$

$16\left(\frac{n}{4}\right)! \leq c.n!$ for some $c < 1$

when $c = 0{,}5$ holds

so $\Rightarrow$ $T(n) = \Theta(n!)$

f) $T(n) = 6.T(n/3) + n^2\log(n)$

$\log_3^3 = 1$

$f(n) = n^2.\log(n)$ $\quad n^{\log_b^a} = n^{\log_3^6} = n^{\log_3^3 + \log_3^2} = n.n^{\log_3^2}$

$\Theta(f(n)) > \Theta(n^{\log_b^a})$ Case 3:

condition

$a.f(n/b) \leq c f(n)$ $\quad c < 1$

$6.\left(\frac{n}{3}\right)^2.\log\left(\frac{n}{3}\right) \leq n^2\log(n).c$

$\overset{2}{\cancel{6}}.\frac{n^2}{\underset{3}{\cancel{9}}}\log\left(\frac{n}{3}\right) \leq n^2.\log(n).c$

$\frac{2n^2}{3}\log(n/3) \leq n^2.\log(n).c$ for $c = 0{,}9$ holds

so $\Rightarrow$ $T(n) = \Theta(n^2\log(n))$

$a=2,\ b=2$

e) $T(n) = \sqrt{2}.T(n/2) + \log n$ $\quad f(n) = \log n$

$n^{\log_a^b} = n^{\log_2^{\sqrt{2}}} = n^{\log_2^{2^{1/2}}} = n^{1/2} = \sqrt{n}$

Case 1: $f(n)$ grows polynomially slower than $n^{\log_b^a}$

$\Theta(\log n) < \Theta(\sqrt{n})$

$T(n) = \Theta(n^{\log_2^{\sqrt{2}}}) = \Theta(\sqrt{n})$

[illegible] [illegible]

d) $T(n) = T(n-2) + n^2$

Recursion Tree

height is $\frac{n}{2}$

$$\begin{tikzcd}
T(n) \arrow[d] \arrow[r, dashed] & \Theta(n^2) \\
T(n-2) \arrow[d] \arrow[r, dashed] & \Theta((n-2)^2) = \Theta(n^2) \\
T(n-4) \arrow[d, dashed] \arrow[r, dashed] & \Theta((n-4)^2) = \Theta(n^2) \\
\Theta(1) \arrow[r, dashed] & \Theta(n^2)
\end{tikzcd}$$

Then Total runtime is

$T(n) = \frac{n}{2} \cdot n^2 = \Theta(n^3)$

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

Master Method
$a = 4$
$b = 2$

$n^{\log_b a} = n^{\log_2 4} = n^2 \quad f(n) = n^2$

$f(n) = n^{\log_b a}$

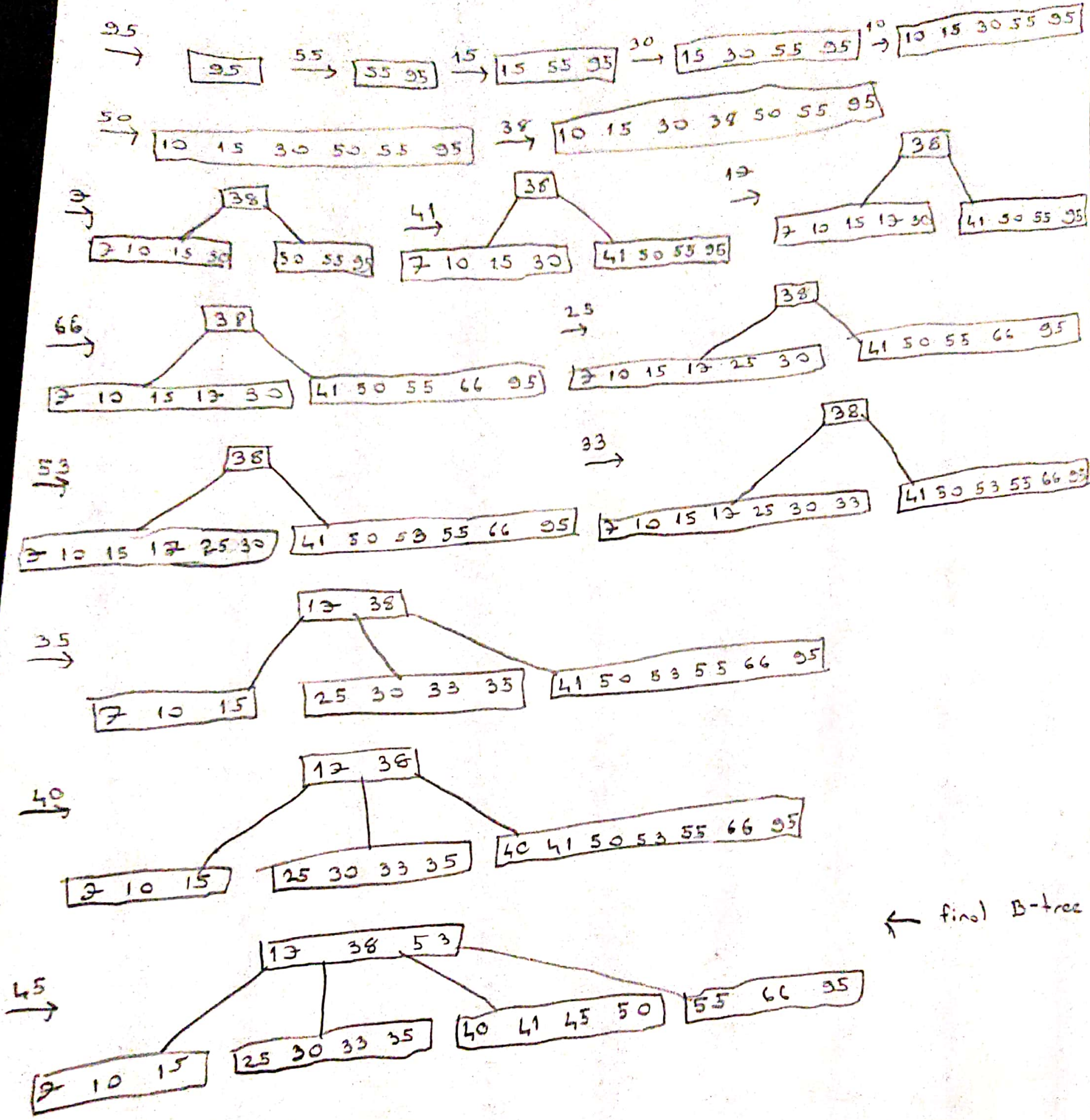
Case 2 :

$T(n) = \Theta\left(n^{\log_b a} \log(n)\right) = \Theta\left(n^2 \log(n)\right)$

a) ?

3)

a) N = {95, 55, 15, 30, 10, 50, 38, 7, 41, 17, 66, 25, 53, 33, 35, 40, 45}



b) 5

Okan Çiftçi
283038012

Question2

a)

Algorithm 1 String Compression algorithm

```
Compress(string: Input)
  $strLen \leftarrow len(string)$
  $newStr \leftarrow$ ””
  $i \leftarrow 0$
  while $i$ is less than $strLen/2$ do
      $newStr \leftarrow newStr + string[i]$
      $i \leftarrow i + 1$
  end while
  return newStr
```

Compress function take a string as input and return compressed version of given string. It basically create a new variable to accumulate characters come from input string. New string accumulate half of given string with while loop and when it's reach half of string return the accumulated string. If given string is even then it return n/2 characters which n is length of given string if given string is odd then it return n/2 + 1 characters to contain middle character.

Algorithm 2 Find Palindrome on Table Algorithm

```
findPalindrome(X, Y, strLen1, strLen2 table)
if $strLen1 == 0$ or $strLen2 == 0$ then
    return ""
end if
if $X[strLen1 - 1] == Y[strLen2 - 1]$ then
    return findPalindrome(X,Y, strLen1-1 ,strLen2-1 , table) + X[strLen-1]
end if
if $table[strLen1 - 1][strLen2] > table[strLen1][strLen2 - 1]$ then
    return findPalindrome(X,Y, strLen1-1,strLen2, table)
end if
return findPalindrome(X,Y, strLen1,strLen2-1, table)
```

findPalindrome is a recursive function aim to find longest palindrome using table. In first if condition is a base condition to finish recursion. It checks if string is reach to end if it's then return empty string.

Second if condition check if last character of X and Y matches if it's match then call findPalindrome function but pass length of string with minus 1 and return this function output and concatenate with last character since characters matched.

If last character of X and Y character is different then we need to use table. If a top cell of the current cell has bigger than the left cell we don't use current character of string in X and call findPalindrome function with strLen1 -1 and rest parameters are constant and return result.

If a left cell of the current cell has bigger than the top cell then don't use the current character of string Y and call findPalindrome function with strLen2 - 1 and rest parameters are constant

We do these operations to shift cells according to character and find longest palindrome.

Algorithm 3 Longest Common Subsequence algorithm

```
LCS(X, Y, strLen, table)
$i, j \leftarrow 1, 1$
while $i \neq strLen + 1$ do
    while $j \neq strLen + 1$ do
        if $X[i-1] == Y[j-1]$ then
            $table[i][j] \leftarrow table[i-1][j-1] + 1$
        else
            $table[i][j] \leftarrow max(table[i-1][j], table[i][j-1])$
        end if
        $j \leftarrow j + 1$
    end while
    $j \leftarrow 1$
    $i \leftarrow i + 1$
end while
return table
```

LCS function aim is to find the length of longest common subsequence between X and Y. It fill table in bottom-up approach, it basically check each index of given first string and second string if characters match then increment related index with 1 if characters don't match then move go up or left according to which cell has a higher value.

Algorithm 4 Find Compress Function

```
Find Compress(string: Input)
  $strLen \leftarrow len(string)$
  $reverseString \leftarrow reverse(string)$
  $table \leftarrow array[strLen + 1][strLen + 1]$
  $table \leftarrow LCS(string, reverseString, strLen, table)$
  $palindromeStr \leftarrow longestPalindrome(string, reverseString, strLen, strLen, table)$
  $compressedPalindromeStr \leftarrow Compress(palindromeStr)$
  return compressedPalindromeStr | (latex bug) =0
```

Find Compress function take a string as input and firstly find reverse of given string to use on longest common subsequence. It create table and assign each cell as 0 to accumulate overlaps in given string then it use LCS function to find longest common subsequence and fill to table. After the fill of table we call longestPalindrome function to find longest palindrome in table then we call compress function and give input palindromeStr and find compressed version of palindrome string then return the compressed version.

b)

Algorithm 5 String Decompression algorithm

```
Decompress(string: Input)
  $strLen \leftarrow len(string)$
  if $strLen$ is even then
      while $strLen \neq 0$ do
          $string \leftarrow string + string[strLen - 1]$
          $strLen \leftarrow strLen - 1$
      end while
  else
      $strLen \leftarrow strLen - 1$
      while $strLen \neq 0$ do
          $string \leftarrow string + string[strLen - 1]$
          $strLen \leftarrow strLen - 1$
      end while
  end if
  return string
```

Decompress function take a string as input and return decompressed version of given string. It basically checks a length of string if length of string is even then it's start from end of string and concatenate with first string. If length of string is odd then it's start from end-1 index of string and concatenate with first string. We start end-1 index since there is a middle character and already in compressed version of given string.

c)

findCompress function firstly call reverse function then create a n+1xn+1 table and assign 0 each cell in table then call LCS, longestPalindrome and Compress functions in order. Time complexity of findCompress function will be O$(n^2 + n^2 + n + n + n) = $ O(n^2) Space complexity of findCompress function will be O$(n^2 + n + 1 + 1 + n) = $ O(n^2)

longestPalindrome function don't create any data structure then space complexity will be O(1). It is a recursive function, if we analyze longestPalindrome function on time complexity is O(n) where n is length of string.

LCS function have nested 2 loops to fill table in a bottom-up manner. Since it uses 2 two for loop as nested, time complexity will be O(n^2) where n is length of given string. Function don't use any extra data structure space complexity will be O(1).

Reverse function is built-in function which take a string input and returns reversed string. It uses a for loop. Reverse function have O(n). time complexity since use one loop. Space complexity will be O(1) since it don't use any data structure.

Compress function only uses 1 while loop. If we look space complexity. Compress function uses a extra string with (n/2) or (n/2+1) character so space complexity will be O(n). If we look time complexity it uses only one loop so time complexity will be O(n).

Decompress function uses one while loop, inside of loop it's simply concatenate related character with given string and return decompressed string. According to big O notation, proposed algorithm contains O(n) time complexity. It's space complexity will be O(1) since we don't use any data structure.

d)

1. If we give input as "IYTECENG611116CENGIYTE" longest palindrome subsequence will be "TEC611116CET"
2. If we give input as "TEC611116CET" as input our compress function output will be "TEC611"

QUESTION4

a)

In disk hardware, number of page reads method needs B-tree and it's t value. So to find optimal t value we need to find worst-case equation of height attribute in tree. If we analyze worst case height of a B-tree it will be $\log_t \frac{n+1}{2}$. We can derive this equation from B-tree construction rule. If n is greater than 1 which n is node count and t is greater or equal to 2 h will be $\log_t \frac{n+1}{2}$. We will use a function named as f which multiply (a + bt) with worst case of height function $f(t) = (a+bt)\log_t \frac{n+1}{2}$. So we need to find optimal value of this function on the t parameter. We need to take derivative according to t and assign it to zero to find appropriate t value. If we calculate the derivative of given f function, output will be $f'(t) = b\log_t \frac{n+1}{2} - (a+bt)\frac{ln\frac{n+1}{2}}{tln(t)^2}$. If we set the our derivative to 0 and simplifying the function remaining part will be $t(ln(t) - 1) = \frac{a}{b}$. Then we can replace a and b with real values. If we create an equation based on e over parts of given equation parts, t will be 3.18096. We can set height as 3 to minimize search time in B-tree. In this problem first we derived worst case complexity of height in B-tree and then multiply with given equation (a + bt) then take derivative to find approximate t value.

b)

Binomial Heaps To understand binomial heaps we need to first understand binary heap. It's a tree data structure. Binary heap used to implement priority queue which supports insert, delete, extract operations in O($log(n)$). time. In computer science there are different use cases of priority queues such as dijkstra's algorithm, prim's algorithm for minimum spanning tree, etc. Binomial heaps provides faster merge and union operation rather than binary heaps. Binomial heap is a collection of Binomial tree where each binomial tree follows min-heap property. Difference between binary and binomial heap is how the heaps are structured. In binary heap, the heap is a single tree. In binomial heap, heap is collection of smaller trees. In binomial heap union operation is O($log(n)$) and find-min operation is O($log(n)$), in binary heap union operation is O(n) and find-min operation is O($log(1)$). According to operation we can use binary or binomial heap.

Fibonacci Heap

Fibonacci heaps is a collection of trees with min-heap or max-heap property. In fibonacci heap, trees can have any shape. In binomial heap every tree has to be a binomial tree. Fibonacci heap uses a pointer to a minimum value so that all tree roots are connected with doubly circular linked list. Both these data structure used to implement priority queue. In binomial heap insert, find-min union operations have O($log(n)$) time complexity. In fibonacci heap amortized time complexity in insert, find-min, union operations are constant time. According to time complexity fibonacci heap is more efficient rather than binomial heap.

2-3-4 Heaps 2-3 heap idea comes from 2-3 tree. 2-3 heap structure is more rigid than Fibonacci heap. Fibonacci heap is based on binary linking but 2-3 heap base on ternary linking. 2-3 heap more efficient because of inking process and tree adjustments after node deletion or insertion. 2-3 heap have less amortized time complexity in decrease-key operation. In Theory of 2-3 heap work experiment shows that 2-3 heap is better than Fibonacci heap by about %20. in Dijikstra algorithm since it often uses decrease-key operation. 3-4 heap proposed to counter 2-3 heap data structure. In Theory of 3-4 heap work experiment shows 3-4 heap performed better than 2-3 heap in dijikstra algortihm and insert and decrease-key operation.

In general we can use fibonacci heap rather than binomial heap. 2-3-4 heaps are hard to implement but in amortized analysis more efficient than fibonacci heap.