Data Structure and Algorithms

Group 13
Ayush Sahu

Problem 1

Task 1.1

vector v

7	5	4	2

k = 7

vector w (length of 8)

0	1	2	3	4	5	6	7

w[i] = 0 for all 0 ≤ i ≤ k.

vector w

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

w[v[i]] = w[v[i]] + 1

vector w

0	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7

vector w

0	0	0	0	0	1	0	1
0	1	2	3	4	5	6	7

vector w

0	0	0	0	1	1	0	1

0 1 2 3 4 5 6 7

vector w

0	0	1	0	1	1	0	1
0	1	2	3	4	5	6	7

vector v2

2	4	5	7

Pseudo Code:

```
function BucketSort(v)
     v2 = NewVector(Length(v))
     k = MaxElement( v)                                    n    O(n)
     w= New Vector( k+1)                                   1    O(1)
     for i = 0 to k                                        p
          w[i] = 0                                         p    O(p)
     end for
     for i=0 to v.size                                     n
          w[v[i]] = w[v[i]] + 1                            n    O(n)
     end for
     for i= 0 to k+1                                       p
          if w[i] > 0                                      n
               add index to v2                        n    O(n)
          end if
     end for
     return v2                                        1
end function
```

C++ Code :

```
void newSort(vector<int> &v, int n){
int k = max_element(v.begin(),v.end());
vector<int> w[k+1];
for(int i=0;i<=k;i++){
w[i]=0;
}
for(int i=0;i<v.size();i++){
w[v[i]]++;
}
int main(){
vector<int> v;
int n;
cout<<"Enter the no of elements: ";
cin>>n;
cout<<"\n";
cout<<"Enter the elements : \n";
for(int i=0;i<n;i++){
cin>>v[i];
}
newSort(v,n);
cout<<"Sorted array is : \n";
for(int i=0;i<n;i++){
if(v[i]==1)
cout<<i<<" ";
return 0;
}
}
```

Task 1.2

pseudo code
time complexity

```
function BucketSort(v)
     v2 = NewVector(Length(v))                     1
     k = MaxElement( v)                            1
     w= NewVector( k+1)                            1    O(1)
     for i = 0 to k                                p
          w[i] = 0                                 p    O(p)
     end for
     for i=0 to v.size                             n
          w[v[i]] = w[v[i]] + 1                    n    O(n)
     end for
     for i= 0 to k+1                               n
          if w[i] is 1                             n
```

```
                add index to v2                 n    O(n)
            end if
        end for
        return v2                               1
end function
```

```
T(n) = O(1) + ∑(i=0 to p)+ ∑(i=0 to n-1) + ∑(i=0 to n-1)
     = O(1) + O(p+1) + O(2n)
     = O(p) + O(n)
( p can be bigger than n )
Linear
```

Task 1.3

The biggest disadvantage is that the input data is constricted to positive integers.

For example, if we take input numbers in vector v as 7, 6, -1, 4, -5 to be sorted and going on with the process. At first, we create a vector w of length greatest element in vector+1 so it is 8. And then we initialize w[i]=0, for all 0 ≤ i ≤ k. And then applying w[v[i]] = w[v[i]] + 1 for every 0 ≤ i < v.size. And the result is

vector w

0	0	0	1	0	1	1	0
1	2	3	4	5	6	7	8

And in the next step we will get result as

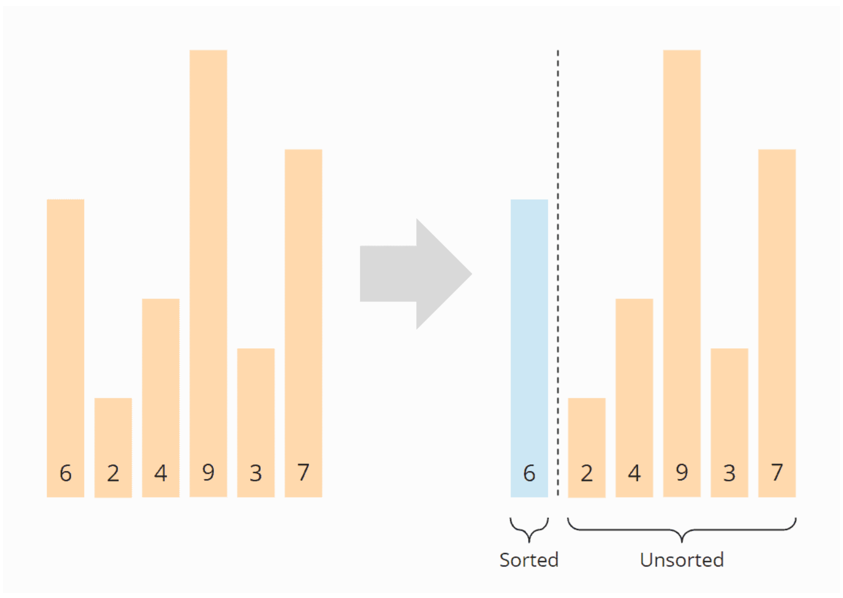
4	6	7

This is not correct as we cannot update the value of negative integers as the index are whole numbers. It would be the same problem with fractions also.

Task 1.4

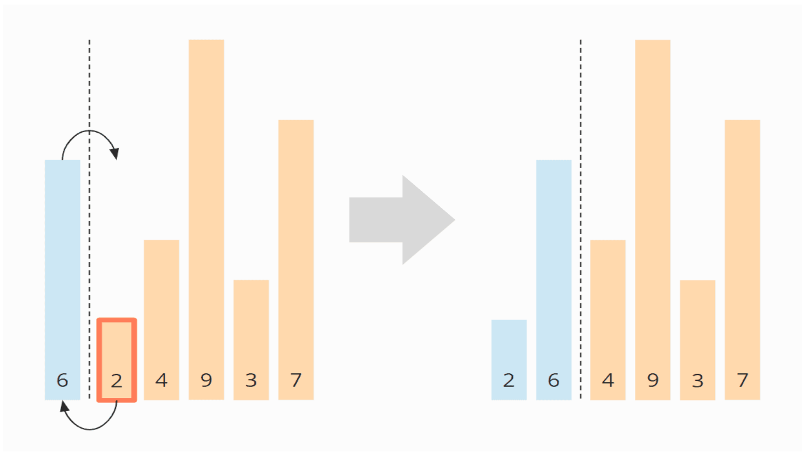
Insertion sort Algorithm

Step 1: First, we divide the array into a left, sorted part, and a right, unsorted part. The sorted part already contains the first element at the beginning, because an array with a single element can always be considered sorted.

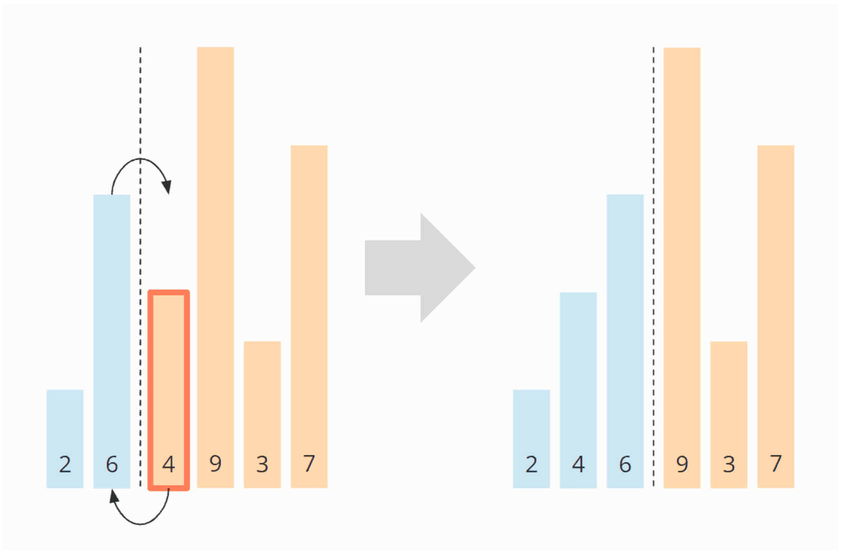


Step 2: Then we look at the first element of the *unsorted* area and check where, in the sorted area, it needs to be inserted by comparing it with its left neighbor.

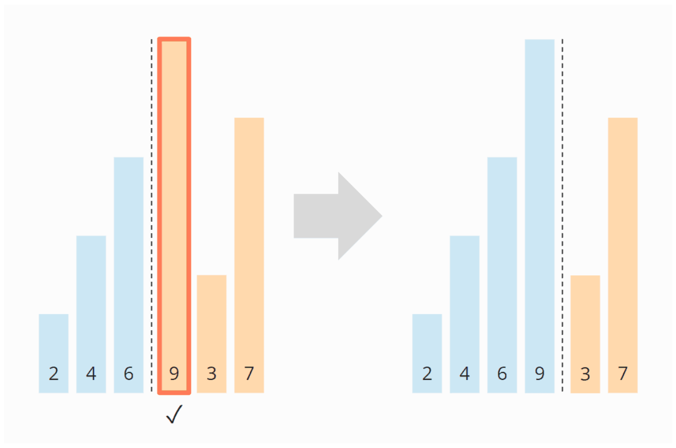
In the example, the 2 is smaller than the 6, so it belongs to its left. In order to make room, we move the 6 one position to the right and then place the 2 on the empty field. Then we move the border between sorted and unsorted area one step to the right:



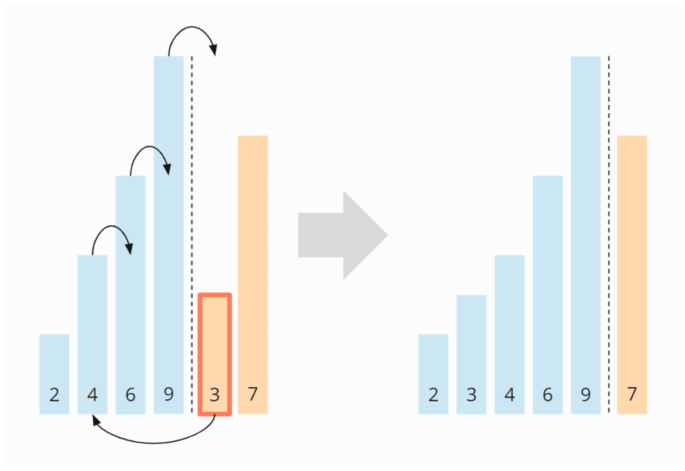
Step 3 : We look again at the first element of the unsorted area, the 4. It is smaller than the 6, but not smaller than the 2 and, therefore, belongs between the 2 and the 6. So we move the 6 again one position to the right and place the 4 on the vacant field:



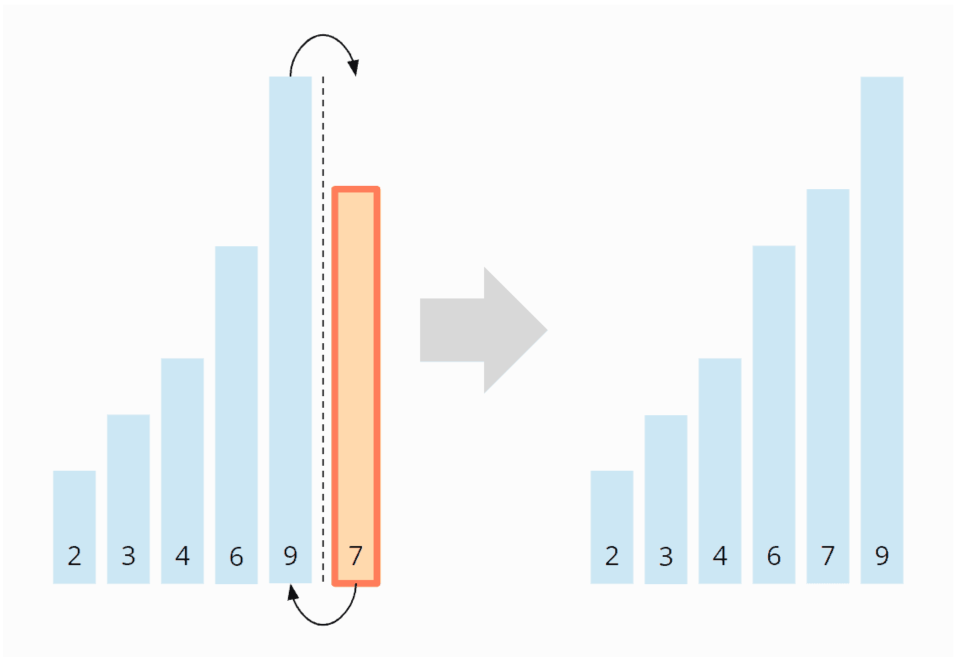
Step 4: The next element to be sorted is the 9, which is larger than its left neighbor 6, and thus larger than all elements in the sorted area. Therefore, it is already in the correct position, so we do not need to shift any element in this step:



Step 5: The next element is the 3, which is smaller than the 9, the 6 and the 4, but greater than the 2. So we move the 9, 6 and 4 one position to the right and then put the 3 where the 4 was before:



Step 6: That leaves the 7 - it is smaller than the 9, but larger than the 6, so we move the 9 one field to the right and place the 7 on the vacant position:

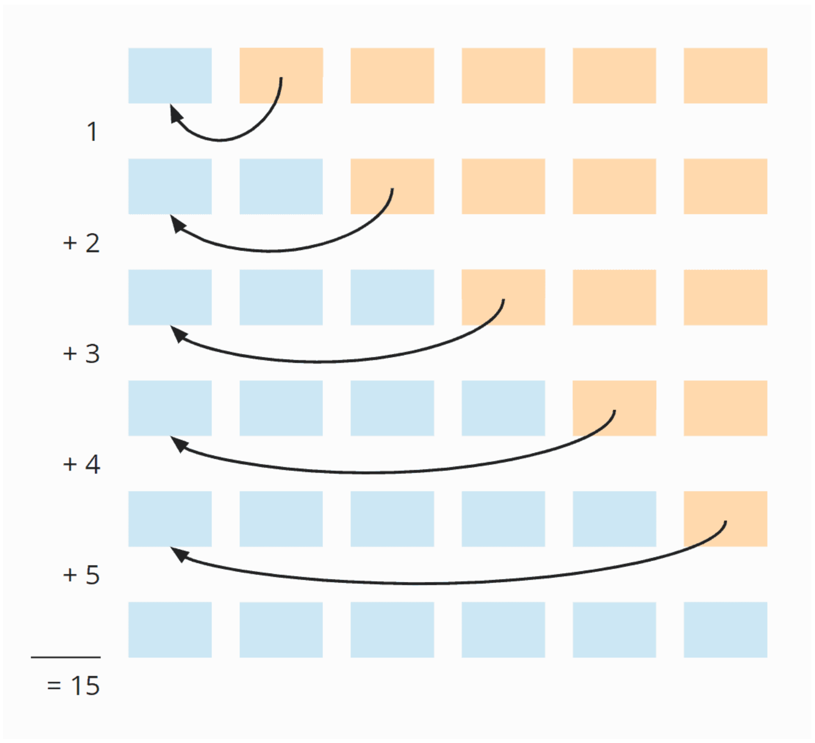


The array is now completely sorted

<u>Worst Time Complexity:</u>

In the worst case, the elements are sorted completely descending at the beginning.

In the following diagram, this is demonstrated by the fact that the arrows always point to the far left:



So, if there are n elements, it goes like *$n \times (n - 1) \times ½$*

When we multiply this out, we get:

$½ n^2 - ½ n$

Even if we have only half as many operations as in the average case, nothing changes in terms of time complexity – the term still contains n^2, and therefore follows:

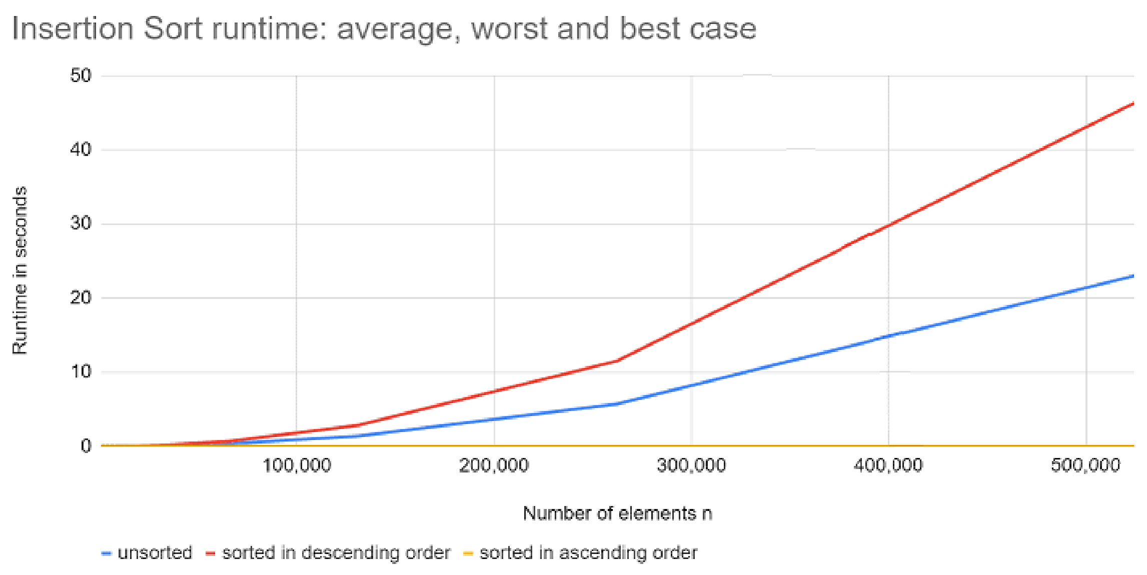
The worst-case time complexity of insertion sort is: $O(n^2)$

Here is the result for Insertion Sort after 50 iterations

n	*time*
32,768	*0.042 ms*
65,536	*0.084 ms*

131,072	*0.168 ms*
262,144	*0.351 ms*

Graph for insertion sort:



From graph, we can see that the ratio T(n)/f(n) is constant for all the values.

PROBLEM 2

Data Structure Use:

- STL Queue
- Node->int id;
  ->Node *followingFriends;

```
        ->Node *followingAdverseries;
```

Pseudo-Code and Algorithm:

```
//The main node is the people for who we want to know who are
his/her friends

//Created a list to put all the friends of the main node
friendsOfMainNode

//We add the main Node to the queue Push mainNode in Queue

//We study all nodes of the graph

//The study is over when we don't have anymore Node to study in
the queue while(Queue is not empty)

//We take a Node to study currentNode = first Node in Queue

//We study if is a friend or an adversary of main Node
if currentNode is in friendsOfMainNode array
    isFriend = true
else
    isFriend = false


if (isFriend)
    //We add all the following friends to the list and to the
queue
    for all Node in followingFriends array
        add Node in friendsOfMainNode
        add Node in Queue
    //We add all adversaries only to the queue
    for all Node in followingAdversaries array
        add Node in Queue
else
    //We add all the following adversaries to the list and to the
queue
    for all Node in followingAdversaries array
        add Node in friendsOfMainNode
        add Node in Queue
    //We add all friends only to the queue
    for all Node in followingFriends array
        add Node in Queue
```

<u>Time Complexity Analysis:</u>

We name “n” the number of nodes in the social network graph.
We name “m” the number of edges in the social network graph.

The algorithm will study all nodes in the graph.

So, what is the complexity of a node’s study ? For each node, we watch after his/her friends/adversaries. That means, watch after all nodes with which he/she has a link (an edge). In fact, we cannot really know what is one node’s study’s complexity because it depends of his/her following friends/adversaries.

But, because we know the number of edges in the graph, we know that for all nodes, all edges will be studied.

For each node, for each neighbor, the cost is O(1) : add to the queue, add in the list if necessary etc.

Finally, the complexity is n (all nodes) times m (all edges) times O(1) equal O(nm).

<u>C++ Code:</u>

<u>List.cpp</u>

```
#include <iostream>
#include "List.h"

using namespace std;

// -- Element -- //

Element::Element(Node * data, Element * nextElement, Element *
previousElement) {
    if (data == NULL) {
        cout << "Cannot create an Element with empty datas. Need a
non-null node.";
        //Amélioration : lever une erreur et quitter le
constructeur
    }
    else {
        this->data = data;
        this->nextElement = nextElement;
        this->previousElement = previousElement;
    }
}

Element::~Element() {
    //Empty for now
}

// Getter

Node * Element::getData() {
    return this->data;
}

Element * Element::getNextElement() {
    return this->nextElement;
}

Element * Element::getPreviousElement() {
    return this->previousElement;
}

// Setter

void Element::setNextElement(Element * nextElement) {
    this->nextElement = nextElement;
}

void Element::setPreviousElement(Element * previousElement) {
    this->previousElement = previousElement;
}


// -- List -- //
```

```
List::List () {
    this->firstElement = NULL ;
    this->lastElement = NULL ;
    this->numberElement = 0 ;
}

List::~List () {
    //Empty for now
}

// Getter

Element * List::getFirstElement () {
    return this->firstElement;
}

Element * List::getLastElement () {
    return this->lastElement;
}

// Other functions

void List::insertLast (Element * element) {
    if (this->sizeList() == 0) {
        this->firstElement = element;
        this->lastElement = element;
    }
    else {
        this->lastElement->setNextElement(element);
        element->setPreviousElement(this->lastElement);
        this->lastElement = element;
    }
    this->numberElement++;
}

void List::removeLast () {
    if (this->sizeList() == 0) {
        cout << "Empty list. No node to remove";
    }
    else if (this->sizeList() == 1) {
        this->firstElement = NULL;
        this->lastElement = NULL;
        this->numberElement--;
    }
    else {

this->lastElement->getPreviousElement()->setNextElement(NULL);
        this->lastElement =
this->lastElement->getPreviousElement();
        this->numberElement--;

    }
}

void List::printList () {
    Element * current = this->firstElement;
    for (int i = 0; i < this->sizeList(); i++) {
        current->getData()->printNode();
        current = current->getNextElement();
        cout << " -> ";
    }
    cout << " NULL";
}

int List::sizeList () {
    return this->numberElement;
}
```

List.h

```
#ifndef DEF_NODELIST
#define DEF_NODELIST

#include "Node.h"

class Element {
    private:
        Node * data;
        Element * nextElement;
        Element * previousElement;

    public:
        Element(Node * data, Element * nextElement, Element *
previousElement);
        ~Element();

        // Getter
        Node * getData();
        Element * getNextElement();
        Element * getPreviousElement();

        // Setter
        void setNextElement(Element * nextData);
        void setPreviousElement(Element * previousElement);

};

// Définition de la classe Liste
class List {
    private:
        Element * firstElement;
        Element * lastElement;
        int numberElement;

    public:
        List ();
        ~List ();

        // Getter
        Element * getFirstElement ();
        Element * getLastElement ();

        // Other functions
        void insertLast (Element * element);
        void removeLast ();
        void printList ();
        int sizeList ();

};



#endif
```

node.cpp

```cpp
#include <iostream>
#include "node.h"

using namespace std;

Node::Node(int id, List * followingFriends, List *
followingAdversaries) {
    this->id = id;
    this->followingFriends = followingFriends;
    this->followingAdversaries = followingAdversaries;
}

Node::~Node() {}

// Getter

int Node::getID() {
    return this->id;
}

List * Node::getFollowingFriends() {
    return this->followingFriends;
}

int Node::getNumberFollowingFriends() {
    return (*this->followingFriends).sizeList();
}

List * Node::getFollowingAdversaries() {
    return this->followingAdversaries;
}

int Node::getNumberFollowingAdversaries() {
    return (*this->followingAdversaries).sizeList();
}

// Other functions

bool Node::hasFriends() {
    return (this->getFollowingFriends() == 0);
}

bool Node::hasAdversaries() {
    return (this->getFollowingAdversaries() == 0);
}

void Node::printNode() {
    cout << this->getID();
}

bool Node::toCompare(Node other) {
    return (this->getID() == other.getID());
}
```

<u>node.h</u>

```
#ifndef DEF_NODE
#define DEF_NODE
#include "List.h"


class Node {
    private:
        int id;
        List * followingFriends;
        List * followingAdversaries;

    public:
        Node(int id, List * followingFriends, List *
followingAdversaries);
        ~Node();

        // Getter
        int getID();
        List * getFollowingFriends();
        int getNumberFollowingFriends();
        List * getFollowingAdversaries();
        int getNumberFollowingAdversaries();

        // Other functions
        bool hasFriends() ;
        bool hasAdversaries();
        void printNode();
        bool toCompare(Node other);
};

#endif
```

social-network.cpp

```
#include <iostream>
#include <queue>
#include <string>
#include "Node.h"
#include "List.h"

using namespace std;

Node __initAdversaryGraph() {
    //Init the social network graph
     //Not implemented
}

bool isInNodeList (Node node, List * nodesList) {
    Element * currentNode = nodesList->getFirstElement();
    bool found = false;
    for (int i = 0; i < nodesList->sizeList(); i++) {
        if (node.toCompare(*(currentNode->getData()))) {
            found = true;
        }
        currentNode = currentNode->getNextElement();
    }
    return found;
}

void searchFriends(Node mainNode){

    //I create a list to put all the friends of the main node
    List * friendsOfMainNode = new List() ;

    //We add the main Node to the queue
    queue <Node> nodesToStudy ;
    nodesToStudy.push(mainNode) ;

    //We study all nodes of the graph
    //The study is finished when we don't have anymore Node to
study in the queue
    while (!(nodesToStudy.empty())) {
        //We take the Node to study
        Node currentNode = nodesToStudy.front() ;
        nodesToStudy.pop() ;

        //We study if is a friend or an adversary of main Node
        bool isFriend = isInNodeList(currentNode,
friendsOfMainNode);

        if (isFriend) {
            //We add all the following friends to the list and to
the queue
            List * followingFriends =
currentNode.getFollowingFriends();
            Element currentElement =
*(followingFriends->getFirstElement()); //We create a copy of the
first element of the list
            for (int i = 0; i <
(currentNode.getNumberFollowingFriends()); i++) {
                // add Node in friendsOfMainNode
                friendsOfMainNode->insertLast(&currentElement);
                // add Node in Queue
                nodesToStudy.push(*(currentElement.getData()));
                // take the next Node
                currentElement =
*(currentElement.getNextElement());
            }
            //We add all adversaries only to the queue
            List * followingAdversaries =
currentNode.getFollowingAdversaries();
            currentElement =
*(followingAdversaries->getFirstElement()); //We create a copy of
the first element of the list
            for (int i = 0; i <
(currentNode.getNumberFollowingAdversaries()); i++) {
                // add Node in Queue
                nodesToStudy.push(*(currentElement.getData()));
                // take the next Node
                currentElement =
*(currentElement.getNextElement());
            }
        }
        else {
            //We add all the following adversaries to the list and
to the queue
            List * followingAdversaries =
currentNode.getFollowingAdversaries();
            Element currentElement =
*(followingAdversaries->getFirstElement()); //We create a copy of
the first element of the list
            for (int i = 0; i <
(currentNode.getNumberFollowingAdversaries()); i++) {
                // add Node in friendsOfMainNode
                friendsOfMainNode->insertLast(&currentElement);
                // add Node in Queue
                nodesToStudy.push(*(currentElement.getData()));
                // take the next Node
                currentElement =
*(currentElement.getNextElement());
            }
            //We add all friends only to the queue
            List * followingFriends =
currentNode.getFollowingFriends();
            currentElement =
*(followingFriends->getFirstElement()); //We create a copy of the
first element of the list
            for (int i = 0; i <
(currentNode.getNumberFollowingAdversaries()); i++) {
                // add Node in Queue
                nodesToStudy.push(*(currentElement.getData()));
                // take the next Node
                currentElement =
*(currentElement.getNextElement());
            }
        }

    }

}
```

```
int main() {

    Node mainNode(8, NULL, NULL);
    cout << "ID : " << mainNode.getID();
    return 1;
}
```

PROBLEM 3

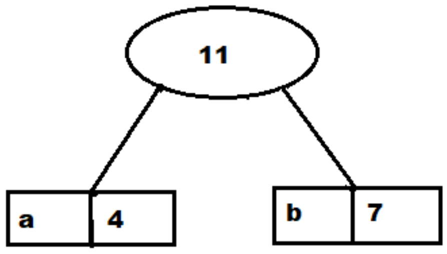
Algorithm

The input is an array of unique characters with their frequencies representing their occurrences. The output is Huffman tree using priority queue.For example, the input are

Character	**Frequency**
a	4
b	7
c	10
d	12
e	16
f	40

Step 1 : Created a leaf node for each unique character and build a min heap of all nodes (Min heap is used as a priority queue). Initially, the least frequent character is taken as the root.

Step 2 : Extract two nodes with minimum frequency from the priority queue. Add a new internal node with frequency 4+7 = 11

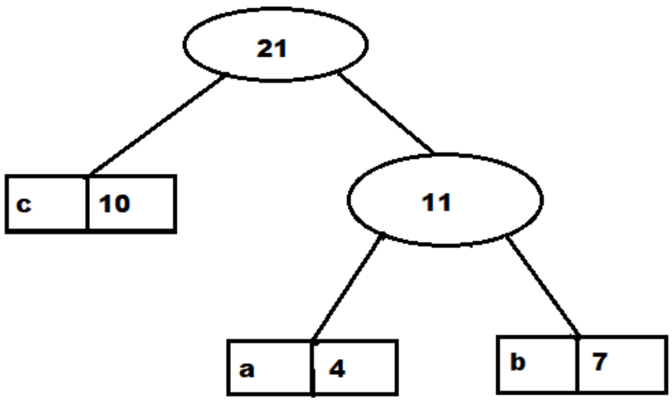


Now min heap contains 5 nodes out of which 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

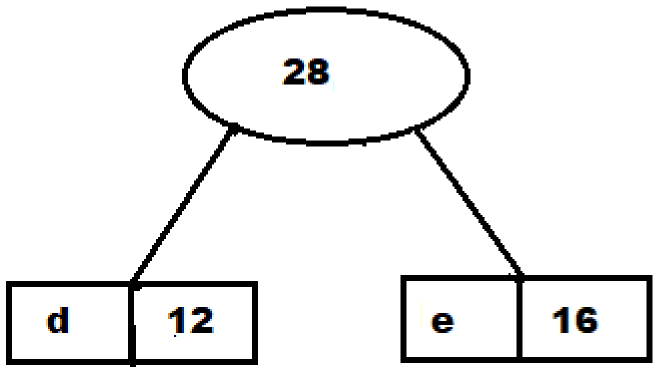
Character	**Frequency**
c	10
Internal Node	11
d	12
e	16
f	40

Step 3 :

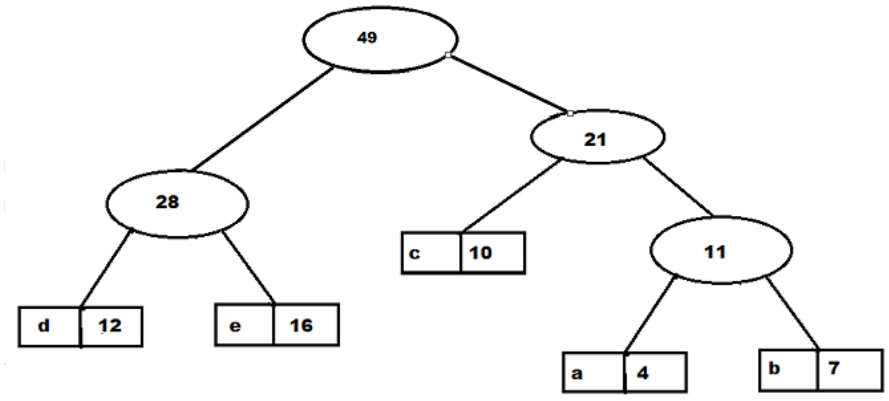
Again extract minimum frequency two nodes from the priority queue and make an internal node with 10+11 = 21. Repeat these steps till we reach the root.



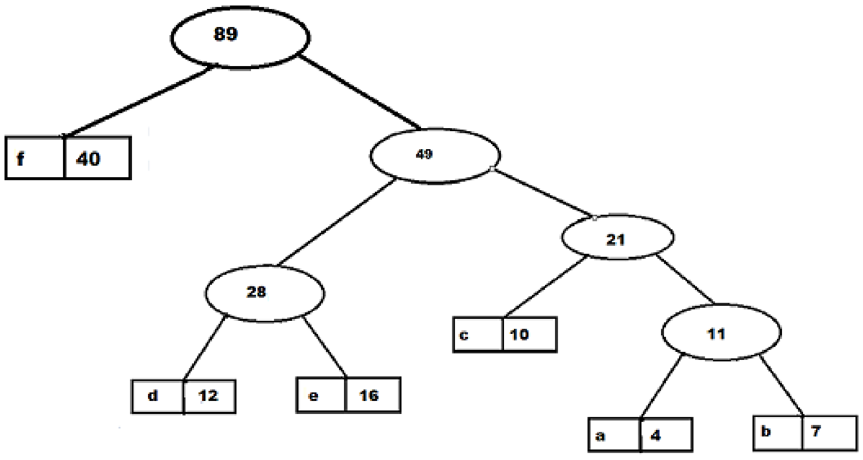
Character	Frequency
d	12
e	16
Internal Node	21
f	40



Character	**Frequency**
Internal Node	21
Internal Node #2	28
f	40



Character	**Frequency**
f	40
Internal Node	49



Now the queue contains only one node:

Character	**Frequency**
Internal Node	89

<u>Step to print huffman tree</u> :

Traverse the tree starting from the root. Write 0 to the left child and 1 to the right child moving from left to right.

The codes are as follows:

Character	**Code-word**
f	0
d	100
e	101
c	110
a	1110
b	1111

<u>C++ Code</u>

```
class Tree{

public:
Tree( int w, char c);
Tree( int w, Tree* t1, Tree* t2);
~Tree();
int getWeight() const;
void printTree(vector<char>& bitString) const;
```

```
private:
Tree* left;
Tree* right;
int weight;
char c;
};

struct TreeWrapper{
TreeWrapper(){
tree=NULL;
}

TreeWrapper(Tree* t){
tree = t;
}

bool operator() (const TreeWrapper &tw) const{
return tree->getWeight() > tw.tree->getWeight();
}
Tree* tree;
};

Tree* generateTree(priority_queue<Tree* , vector<Tree*>,
TreeWrapper> pq){
while(pq.size!=1){
Tree* left = pq.top();
pq.pop();
Tree* node = new Tree(left->freq + right->freq , '&');
node->left = left;
node->right = right;

pq.push(node);
}
return pq.top();
}

void printTree(Tree* root, int arr[], int top){

if(root->left){
arr[top]=0;
printTree(root->left,arr,top+1);
}
```

```
if(root->right){
arr[top]=1;
printTree(root->right,arr,top+1);
}

if(!root->left && !root->right){
for(int i=0;i<top;i++){
cout << arr[i] <<" ";
}
cout<<" : "<<root->data<<endl;
}
}

void TreeCode(char data[], int freq[] , int size){
priority_queue(Tree*, vector<Tree*>,TreeWrapper> pq;

for(int i=0;i<size;i++){
Tree* newNode = new Tree(data[i], freq[i]);
pq.push(newNode);
}

Tree* root = generateTree(pq);
int arr[MAX_SIZE] , top = 0;
printTree(root,arr,top);
}

int main(){
char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
int freq[] = { 5, 9, 12, 13, 16, 45 };
int size = sizeof(data)/sizeof(data[0]);
TreeCode(data, freq, size);
return 0;
}
```

Pseudo Code :

```
Procedure Huffman(C):  // C is the set of n characters and related
information
n = C.size
Q = priority_queue()
```

```
for i = 1 to n //O(n)
  nodeI = node(C[i])
  Q.push(nodeI)
end for
while Q.size() is not equal to 1
  Z = new node()
  Z.left = x = Q.getMinWeightAndRemoveIt() //O(n) * O(n-1) *
O(n-2) …
  Z.right = y = Q.getMinWeightAndRemoveIt() //O(n) * O(n-1) *
O(n-2) …

  Z.frequency = x.frequency + y.frequency //O(1)
  Q.push(Z) //O(1)
end while
//O(n) + O(n-1) + O(n-2) + … = E(O(n-i)) with i between 0 and n-1
= O(n(n+1)) = O(n²)
Return Q //O(n^3)
```

Time Complexity Analysis :

Using a priority queue to store the weight of each tree, each iteration requires O(logn) to determine the cheapest weight and insert the new weight. There are O(n) iterations, one for each item.
Hence, the time complexity of the algorithm is O(nlogn).

PROBLEM 4

Task 4.1

Pseudo Algorithm

```
function calculateComplexity(n) {

      If n == 1
           then T = 1
      Else //we use the recursive expression to calculate T(n)
         T = calculateComplexity(n - 1) +
calculateComplexity(ceil(n/2)) + n
```

```
      }

      return T
}
```

Task 4.2

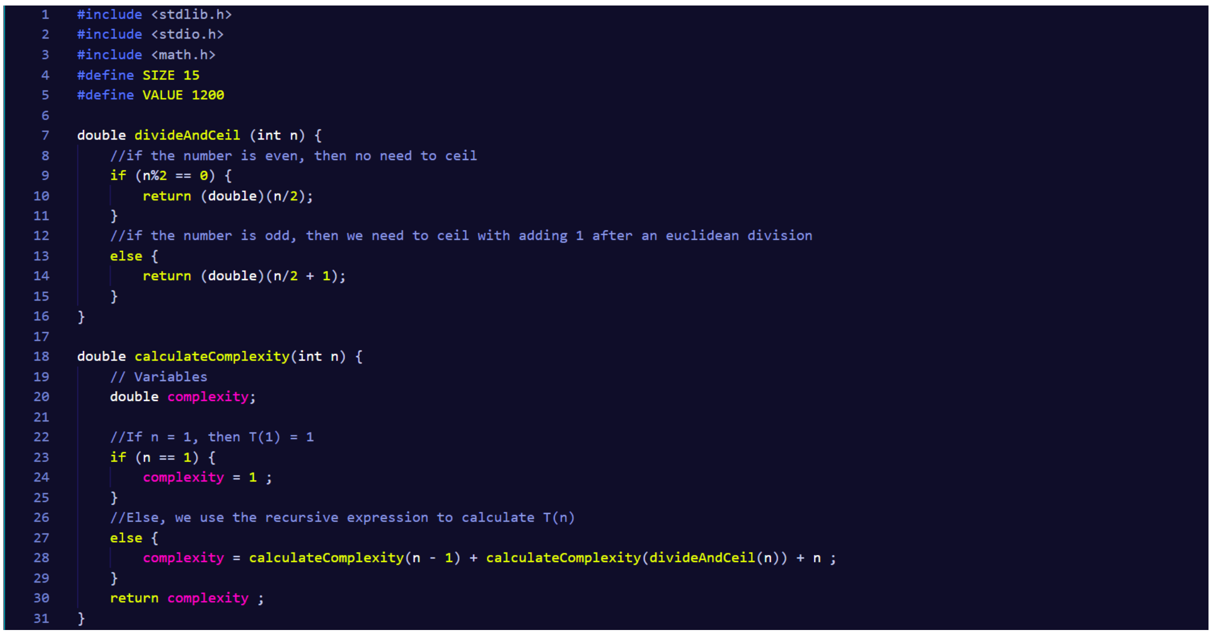
Pseudo Algorithm

```
function calculateComplexityDynamicly(n) {

      //We check in the array if you have already calculate the
value of T(n)
      If T[n] == 0

            If n == 1
                  then T[n] = 1
            Else //we use the recursive expression to calculate T(n)
                  T[n] = calculateComplexity(n - 1) +
calculateComplexity(ceil(n/2)) + n
            }
      }
      return T[n]
}
```

Algorithms for Task 4.1 and 4.2

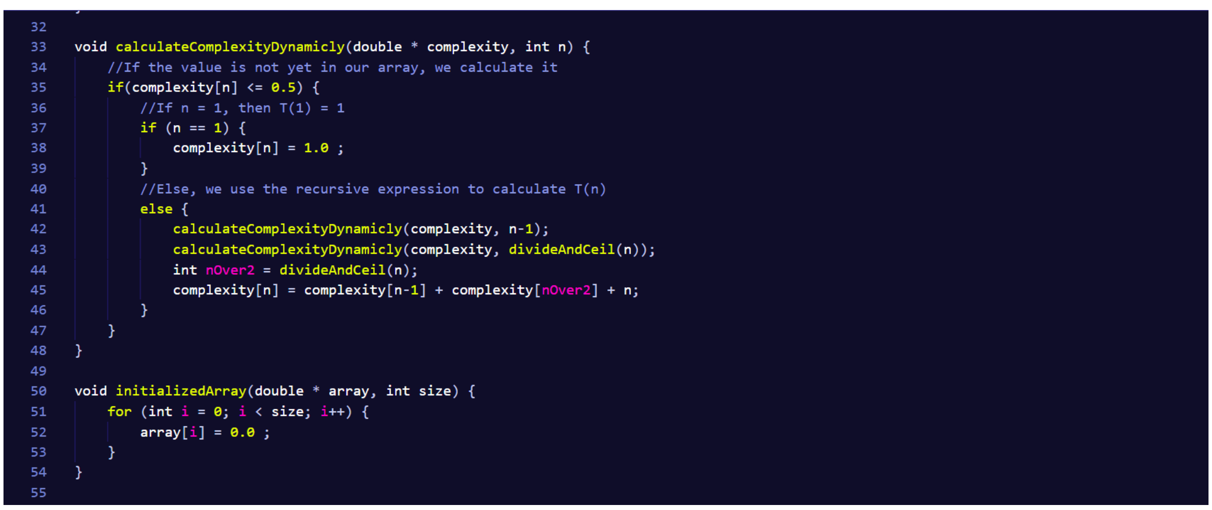


```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define SIZE 15
#define VALUE 1200

double divideAndCeil (int n) {
    //if the number is even, then no need to ceil
    if (n%2 == 0) {
        return (double)(n/2);
    }
    //if the number is odd, then we need to ceil with adding 1 after an euclidean division
    else {
        return (double)(n/2 + 1);
    }
}

double calculateComplexity(int n) {
    // Variables
    double complexity;

    //If n = 1, then T(1) = 1
    if (n == 1) {
        complexity = 1 ;
    }
    //Else, we use the recursive expression to calculate T(n)
    else {
        complexity = calculateComplexity(n - 1) + calculateComplexity(divideAndCeil(n)) + n ;
    }
    return complexity ;
}
```

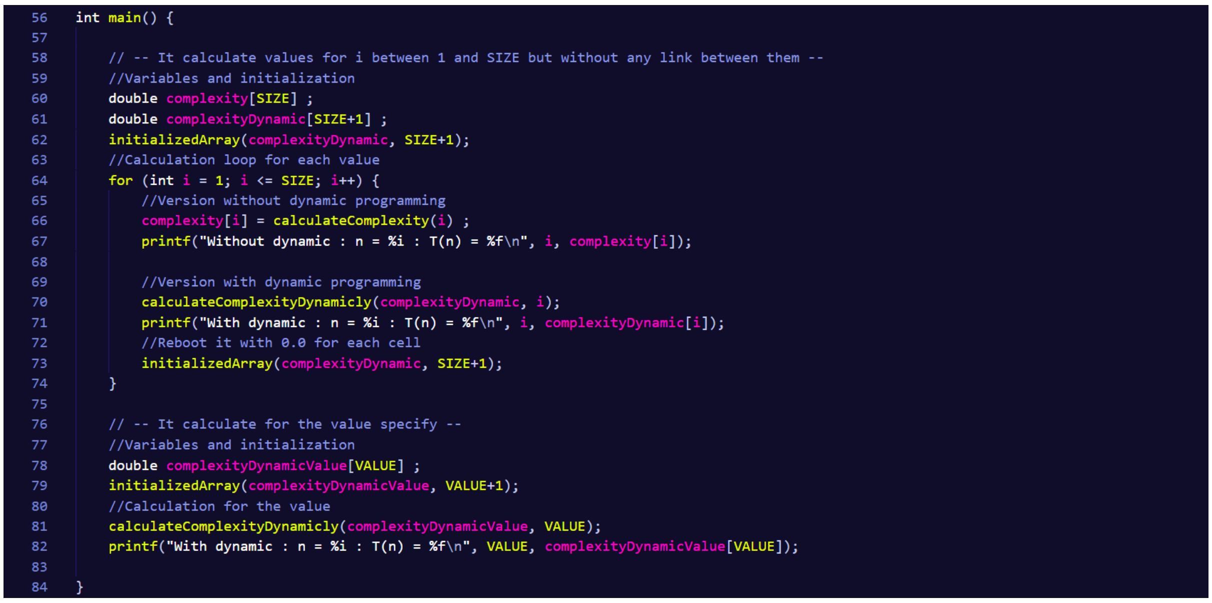


```c

void calculateComplexityDynamicly(double * complexity, int n) {
    //If the value is not yet in our array, we calculate it
    if(complexity[n] <= 0.5) {
        //If n = 1, then T(1) = 1
        if (n == 1) {
            complexity[n] = 1.0 ;
        }
        //Else, we use the recursive expression to calculate T(n)
        else {
            calculateComplexityDynamicly(complexity, n-1);
            calculateComplexityDynamicly(complexity, divideAndCeil(n));
            int nOver2 = divideAndCeil(n);
            complexity[n] = complexity[n-1] + complexity[nOver2] + n;
        }
    }
}

void initializedArray(double * array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = 0.0 ;
    }
}

```



```c
int main() {

    // -- It calculate values for i between 1 and SIZE but without any link between them --
    //Variables and initialization
    double complexity[SIZE] ;
    double complexityDynamic[SIZE+1] ;
    initializedArray(complexityDynamic, SIZE+1);
    //Calculation loop for each value
    for (int i = 1; i <= SIZE; i++) {
        //Version without dynamic programming
        complexity[i] = calculateComplexity(i) ;
        printf("Without dynamic : n = %i : T(n) = %f\n", i, complexity[i]);

        //Version with dynamic programming
        calculateComplexityDynamicly(complexityDynamic, i);
        printf("With dynamic : n = %i : T(n) = %f\n", i, complexityDynamic[i]);
        //Reboot it with 0.0 for each cell
        initializedArray(complexityDynamic, SIZE+1);
    }

    // -- It calculate for the value specify --
    //Variables and initialization
    double complexityDynamicValue[VALUE] ;
    initializedArray(complexityDynamicValue, VALUE+1);
    //Calculation for the value
    calculateComplexityDynamicly(complexityDynamicValue, VALUE);
    printf("With dynamic : n = %i : T(n) = %f\n", VALUE, complexityDynamicValue[VALUE]);

}
```

Time complexity analysis

T(n) = T(n - 1) + T(ceil(n/2)) + n

So, to know the complexity of T(n) we need to discuss the complexity of T(n - 1) and T(ceil(n/2)).

Because there is a memory, we will calcul only once T(n’) for each n’. So for which n’ do you need to know T(n’) to calculate T(n) ? All 0 < n’ < n because of the T(n – 1) part.
And it’s all. Yes, knowing T(n’) for all 0 < n’ < n is required for T(n - 1) part and enough for T(ceil(n’/2)) part.

Now that we know all T(n’) that we need to calculate, we need to know how it costs to calculate one T(n’). Because we will calculate all T(n’) for 0 < n’ < n, we can start with the easiest one T(1) and then T(2) etc. until T(n). Each time there is a sum of three terms. Each term will be read in the array of previous values. So this is an O(1) complexity.

Finally, we made n calculations with an O(1) complexity. Then, the complexity of T(n) calculation is O(n) only.

Task 4.3

The most efficient algorithm to calculate T(n) is the one of Task 4.2. Indeed, the Task 4.2. algorithm has a “memory” of all previous calculations he made since we call it with the n value that we are interested to know the T(n). To understand it better, we will take an example :
We want to calculate T(4) :
Step by step calculation with Task 4.1. algorithm :

```
T(4) = T(3) + T(2) + 4
with
     T(3) = T(2) + T(2) + 3
     with
          T(2) = T(1) + T(1) + 2
          with T(1) = 1
          and T(1) = 1
          so T(2) = 4
     and
          T(2) = T(1) + T(1) + 2
          with T(1) = 1
          and T(1) = 1
          so T(2) = 4
     so T(3) = 11
and
     T(2) = T(1) + T(1) + 2
     with T(1) = 1
     and T(1) = 1
     so T(2) = 4
so T(4) = 19
```

For Task 4.1 Algorithm, thanks to colors, we can see that we calculate T(2) three different times (blue, green, purple). But it’s value never changes.

This is where the Task 4.2 Algorithm “memory” starts to be useful.

Lets see the example for the Task 4.2 algorithm :

```
T(4) = T(3) + T(2) + 4
with
     T(3) = T(2) + T(2) + 3
     with
          T(2) = T(1) + T(1) + 2
          with T(1) = 1
          and T(1) = 1
          so T(2) = 4
     and
          T(2) = 4 (// We know it because we already calculate it
just before)
     so T(3) = 11

and
     T(2) = 4 (// We know it because we already calculate it just
before)
so T(4) = 19
```

For Task 4.2 Algorithm, thanks to colors, we can see that we calculate T(2) once (purple) and read it twice (blue, green). So, we have optimized calculation time with previous calculation “memory”. Indeed, to read the value is less expensive than to calculate it. This is the principle of dynamic programming used with recursive programs.

Finally, for T(4) we saved 2 calculation’s time, imagine with a bigger number. It really worth it.

This kind of programming is useful here because we have two “branches” of calculation : T(n - 1) and T(ceil(n/2)). So at one point, the same calculation can appear for each branch. Each time we have two or more branches, dynamic programming is useful. One very famous recursive program using dynamic programming because of two branches is Fibonacci calculation.