# Tema 0
# Proiectarea Algoritmilor

Dobre Gigi-Alexandru 322CD

Aprilie 2021

FACULTATEA DE AUTOMATICA SI CALCULATOARE
UNIVERSITATEA POLITEHNICA BUCURESTI

## Contents

# 1 Divide and Conquer

### 1.1 The description of the algorithm

In computer science, divide and conquer is an algorithm design paradigm based on recursively breaking down a problem into more subproblems of the same related type until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

### 1.2 The GCD (greatest common divisor) problem

Given a vector of n natural elements, using the divide and conquer approach, compute the greatest common divisor of the elements from which the vector is made.

### 1.3 The description of the chosen problem, using a natural language

The problem itself is not very hard, but it is a perfect example for the divide and conquer algorithm. Firstly, we need to calculate the position of the element in the middle. After that, we need to divide the problem into subproblems, using two recursive calls of the same function. For the first one, the beginning is the position of the first element in the vector and the end is the one we found after calculating the middle. For the second one, the beginning is the position we found and the end is the position of the last element in the vector. It goes without saying that, for every recursive call, the positions might differ, meaning that the first and the last position will change.

This way, we obtain the GCD of the left side and the GCD of the right side, but to do that, we need further calculations. We will process the value of the left and right side until they are equal, and return one of them. The processing is made using the Euclidian algorithm by substraction. The solutions of the subproblems are combined to obtain the solution of the initial problem.

### 1.4 Pseudocode

**Algorithm 1** greatestCommonDivisor(vector[N], start, end)

```
if start == end then
    return vector[start];
else
    mid ← start + (end - start) / 2;
    left ← greatestCommonDivisor(vector, start, mid);
    right ← greatestCommonDivisor(vector, mid + 1, end);
while left != right do
    if left >right then
        left ← left - right;
    else
        right ← right - left;
return left;
```

### 1.5 Efficiency analysis

The divide and conquer algorithms have a good time complexity, if the dimensions of the subproblems are aproximately equals. If the combinations steps are missing, the speed of these algorithms is even bigger (ex. binary search). In most of the cases, one of which we found ourselves in, dividing the problem halves the dimension of it. Speaking of time complexity, for the divide and conquer approach, it is
$T(n) = a * T(n/b) + O(n^k)$.
* a $->$ the number of sub-problems we divide the original one in : 2
* $n/b ->$ the dimension of the subproblems : n/2
* $O(n^k) ->$ the complexity for the Euclidian algorithm by substraction: O(n) in the worst case (ex. gcd(n, 1)).
$a = b^k (2 = 2^1) => T(n) = O(n^k * log_b n) = O(n * log_2 n) = O(nlogn)$

The algorithm presented is an effiecient one, but there can be one way to improve it, by changing the Euclidian algorithm by substraction with the binary Euclidian algorithm which has a complexity of O(logn).

### 1.6 Example of practical use

Calculate the greatest common divisor for 12, 30, 24, 36, 72, 18.

```
                              (12 30 24 36 72 18)
                   (12 30 24)                        (36 72 18)
        (12 30)                  (24)      (36 72)               (18)
(12)            (30)                    (36)     (72)
          6                      24         36                    18
                       6                               18
                                   6
```

## 2 Greedy

### 2.1 The description of the algorithm

In computer science, a greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless, a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

### 2.2 Dijkstra's shortest path algorithm

Given an undirected graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph. The graph is represented using an adjacency matrix.

### 2.3 The description of the chosen problem, using a natural language

Finding the shortest paths from a given source to all vertices is one of the most known problems when talking about a graph and the Dijkstra algorithm, which is based on the greedy approach, solves it very well.

In order to obtain the solutions, we have to follow some simple steps:

* we must create a shortest path tree set, in which we will know which vertices are included in this tree, whose distance from source is calculated.

* we must assign a distance value for all the vertices, initially the upper limit of the data type used; the distance from the source node to itself is 0.

* while there are still vertices that are not in the spt, we pick a vertex with a minimum distance and include it in spt; then we update all the distances of the adjacent vertices of the vertex we picked.

### 2.4 Pseudocode

**Algorithm 2** dijkstraST(graph[V][V], src)

```
//new vector to store the shortest distance from src to i
dist[V];
//new vector, sptSet[i] will be true if i is included in spt or the shortest distance is found
sptSet[V];
for i = 0 to V do
    dist[i] ← upperLimit(infinite);
    sptSet[i] ← 0;
dist[src] ← 0;
for count = 0 to V - 2 do
    u ← minDistance(dist, sptSet);
    sptSet[u] ← 1;
    for v = 0 to V - 1 do
        // v not in sptSet; edge from u to v; weight from src to v through u smaller than current value
        if !sptSet[v] and graph[u][v] and dist[u] ! = upperLimit and dist[u] + graph[u][v] < dist[v] then
            dist[v] ← dist[u] + graph[u][v];
printSolution(dist);
```

**Algorithm 3** minDistance(dist[V], sptSet[v])

```
1: min ← upperLimit;
2: for v = 0 to V - 1 do
3:     if sptSet[v] == 0 and dist[v] <= min then
4:         min ← dist[v];
5:         minIdx ← v;
6: return minIdx;
```

### 2.5 Efficiency analysis

The time complexity for both calculating the minDistance and printing the solution is O(V), where V is the number of vertices the graph has. Even so, these do not define the time complexity of the algorithm.

That is determined by the for in for, which makes the time complexity of the Dijkstra algorithm to be O($V^2$). This can be improved significantly by changing the representation of the graph from matrix of adjacency to list of adjacency, so in the end it will be O($E * logV$), where E is the number of Edges.

The Greedy Choice Property is making whatever choice seems best at the moment and then solve the subproblems that arise later. As we can observe, this property is present in our algorithm by selecting the nearest vertex that does not yet belong to the set of vertices picked so far. This way we obtain the local optimal solution which will eventually lead us to the global one.

The Optimal Structure is also included in the above explanation because the global optimal solution contains the optimal solutions to the subproblems.

### 2.6 Example of practical use

The graph and the cost of every edge:

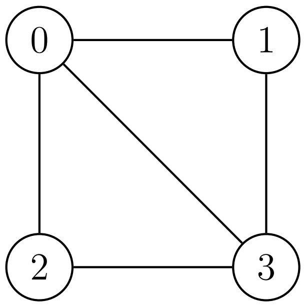
0 - 1; cost = 6

0 - 2; cost = 3

0 - 3; cost = 3

1 - 3; cost = 10

2 - 3; cost = 7



Calculate the shortest paths from source node = 3;

| The node | Distance from 3 |
|---|---|
| 0 | 3 (3 − > 0) |
| 1 | 9 (3 − > 0 − > 1) |
| 2 | 6 (3 − > 0 − > 2) |
| 3 | 0 |

## 3 Dynamic programming

### 3.1 The description of the algorithm

In computer science, the dynamic programming is mainly an optimization over recursion, meaning that we can store the results of subproblems for later use. It significantly reduces the time complexities from exponential to polynomical.

### 3.2 Minimize the cost to make all the adjacent elements distinct in an array problem

Given two integer arrays - vector and cost of size N -, the task is to make all adjacent elements distinct at minimum cost. cost[i] denotes the cost to increment ith element by 1.

### 3.3 The description of the chosen problem, using a natural language

Before we begin solving the problem, we have to observe that an element needs to be increased maximum twice, which gives us an idea on how to proceed further. We will use an 2D array, one dimesion has the length N, representing the elements, and the other 3 (no increase, one increase, two increases).

dp[i][j] is the minimum cost required to make ith element distinct from its adjacent elements using j increments and it can be calculated using the following formula: dp[i][j] = j * cost[i] + (minimum from previous element if both elements are different).

### 3.4 Pseudocode

**Algorithm 4** minCostIncrement(vector[N], cost[N], N)

```
// the dp vector is dp[N][3]
dp[0][0] ← 0;
dp[0][1] ← cost[0];
dp[0][2] ← cost[0] ∗ 2;
for i = 1 to N - 1 do
    for j = 0 to 2 do
        minimum ← upperLimit(infinit);
        // current element is not equal to previous non-increased element
        if j + vector[i] != vector[i - 1] then
            minimum ← min(minimum, dp[i - 1][0]);
        // current element is not equal to previous element after being increased by 1
        if j + vector[i] != vector[i - 1] + 1 then
            minimum ← min(minimum, dp[i - 1][1]);
        // current element is not equal to previous element after being increased by 2
        if j + vector[i] != vector[i - 1] + 2 then
            minimum ← min(minimum, dp[i - 1][2]);
        dp[i][j] ← j ∗ cost[i] + minimum;
solution ← upperLimit;
for i = 0 to 2 do
    solution ← min(solution, dp[N - 1][i]);
return solution;
```

### 3.5 Efficiency analysis and reccurence

Given the algorithm, for which is not hard to see the time comlexity, we can say it is a pretty good algorithm because the complexity is determined by the for in for, which is O(3 $*$ N). In the end it resumes to O(N).

It is a fast algorithm because for a problem like this, we would've probably needed to check every time if one change in the future/present will cause a problem in the past. That being said, I don't think there is a way to make this algorithm more efficient and still keep the dynamic programming approach.

Speaking of reccurence, the formula is dp[i][j] = j * cost[i] + (minimum from previous element). The j * cost[i] represents the cost to increment the current element with j. We need to add the minimum from previous element in order to keep track of the optimal solution, so this way at the end we will know for sure what is the minimum cost to make all the adjacent elements distinct.

### 3.6 Example of practical use

Given N = 5, vector = [1, 1, 2, 2, 3] and cost = [3, 2, 5, 4, 2], make all adjacent elements distinct at minimum cost.

```
dp[0][0] = 0;
dp[0][1] = 3;
dp[0][2] = 3 ∗ 2 = 6;

i = 1:
* j = 0:
        * 0 + 1 != 1 (F) − > go on;
        * 0 + 1 != 1 + 1 (A) − > minimum = min(Inf, dp[0][1]) = min(Inf, 3) = 3;
        * 0 + 1 != 1 + 2 (A) − > minimum = min(3, dp[0][2]) = min(3, 6) = 3;
=>dp[1][0] = 0 ∗ 2 + 3 = 3;

* j = 1:
```

* $1 + 1 != 1$ (A) $->$ minimum = min (Inf, dp[0][0]) = min(Inf, 0) = 0;
* $1 + 1 != 1 + 1$ (F) $->$ go on;
* $1 + 1 != 1 + 2$ (A) $->$ minimum = min(0, dp[0][2]) = min(0, 6) = 0;

=>dp[1][1] = $1 * 2 + 0 = 2$;

* j = 2:
    * $2 + 1 != 1$ (A) $->$ minimum = min(Inf, dp[0][0]) = min(Inf, 0) = 0;
    * $2 + 1 != 1 + 1$ (A) $->$ minimum = min(0, dp[0][1]) = min(0, 3) = 0;
    * $2 + 1 != 1 + 2$ (F) $->$ go on;

=>dp[1][2] = $2 * 2 + 0 = 4$;

dp[1][0] = 3;
dp[1][1] = 2;
dp[1][2] = 4;

i = 2:
* j = 0:
    * $0 + 2 != 1$ (A) $->$ minimum = min(Inf, dp[1][0]) = min(Inf, 3) = 3;
    * $0 + 2 != 1 + 1$ (F) $->$ go on;
    * $0 + 2 != 1 + 2$ (A) $->$ minimum = min(2, dp[1][2]) = min(3, 4) = 3;

=>dp[2][0] = $0 * 5 + 3 = 3$;

* j = 1:
    * $1 + 2 != 1$ (A) $->$ minimum = min(Inf, dp[1][0]) = min(Inf, 3) = 3;
    * $1 + 2 != 1 + 1$ (A) $->$ minimum = min(3, dp[1][1]) = min(3, 2) = 2;
    * $1 + 2 != 1 + 2$ (F) $->$ go on;

=>dp[2][1] = $1 * 5 + 2 = 7$;

* j = 2:
    * $2 + 2 != 1$ (A) $->$ minimum = min(Inf, dp[1][0]) = min(Inf, 3) = 3;
    * $2 + 2 != 1 + 1$ (A) $->$ minimum = min(3, dp[1][1]) = min(3, 2) = 2;
    * $2 + 2 != 1 + 2$ (A) $->$ minimum = min(2, dp[1][2]) = min(2, 4) = 2;

=>dp[2][2] = $2 * 5 + 2 = 12$;

dp[2][0] = 3;
dp[2][1] = 7;
dp[2][2] = 12;

i = 3:
* j = 0:

        * 0 + 2 != 2 (F) − > go on
        * 0 + 2 != 2 + 1 (A) − > minimum = min(Inf, dp[2][1]) = min(Inf, 7) = 7;
        * 0 + 2 != 2 + 2 (A) − > minimum = min(7, dp[2][2]) = min(7, 12) = 7;
=>dp[3][0] = 0 ∗ 4 + 7 = 7;

* j = 1:
        * 1 + 2 != 2 (A) − > minimum = min(Inf, dp[2][0]) = min(Inf, 3) = 3;
        * 1 + 2 != 2 + 1 (F) − > go on;
        * 1 + 2 != 2 + 2 (A) − > minimum = min(3, dp[2][2]) = min(3, 12) = 3;
=>dp[3][1] = 1 ∗ 4 + 3 = 7;

* j = 2:
        * 2 + 2 != 2 (A) − > minimum = min(Inf, dp[2][0]) = min(Inf, 3) = 3;
        * 2 + 2 != 2 + 1 (A) − > minimum = min(3, dp[2][1]) = min(3, 7) = 3;
        * 2 + 2 != 2 + 2 (F) − > go on;
=>dp[3][2] = 2 ∗ 4 + 3 = 11;

dp[3][0] = 7;
dp[3][1] = 7;
dp[3][2] = 11;

i = 4:
* j = 0:
        * 0 + 3 != 2 (A) − > minimum = min(Inf, dp[3][0]) = min(Inf, 7) = 7;
        * 0 + 3 != 2 + 1 (F) − > go on;
        * 0 + 3 != 2 + 2 (A) − > minimum = min(7, dp[3][2]) = min(7, 11) = 7;
=>dp[4][0] = 0 ∗ 2 + 7 = 7;

* j = 1:
        * 1 + 3 != 2 (A) − > minimum = min(Inf, dp[3][0]) = min(Inf, 7) = 7;
        * 1 + 3 != 2 + 1 (A) − > minimum = min(7, dp[3][1]) = min(7, 7) = 7;
        * 1 + 3 != 2 + 2 (F) − > go on;
=>dp[4][1] = 1 ∗ 2 + 7 = 9;

* j = 2:
        * 2 + 3 != 2 (A) − > minimum = min(Inf, dp[3][0]) = min(Inf, 7) = 7;
        * 2 + 3 != 2 + 1 (A) − > minimum = min(7, dp[3][1]) = min(7, 7) = 7;
        * 2 + 3 != 2 + 2 (A) − > minimum = min(7, dp[3][2]) = min(7, 11) = 7;
=>dp[4][2] = 2 ∗ 2 + 7 = 11;

dp[4][0] = 7;
dp[4][1] = 9;
dp[4][2] = 11;
Answer: The minimum cost is min(dp[4][0], dp[4][1], dp[4][2]) = min(7, 9, 11) = 7;

## 4 Backtracking

### 4.1 The description of the algorithm

In computer science, backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem. The problem I chose for this type of algorithm is an enumeration problem, meaning we need to find all feasible solutions.

### 4.2 Unique subsequences of length K with given sum

Given an array - vector of N integers - and two numbers - K and S -, the task is to print all the subsequences of length K with the sum S.

### 4.3 The description of the chosen problem, using a natural language

The problem consists of finding all the possible combinations, choosing only K numbers, between the elements of a vector, that have their sum equal to S. For that, we use a recursive approach having two conditions to stop, if we've reached the maximum numbers for a combination, or the end of the vector. When we've found the necessary numbers, we check the sum, printing them if it is ok.

## 4.4 Pseudocode

**Algorithm 5** uniqueSubSeq(vector[N], result[N], N, K, S, vectorPos, resultPos)

```
// we have a subsequence of length K
if resultPos == K then
    S2 ← 0;
    for i = 0 to resultPos - 1 do
        S2 ← S2 + result[i];
    // check to see if the sum of the subsequence is equal to the target sum
    if S2 == S then
        // printing every element in the result array
        print(result);
    return;
if vectorPos < len then
    uniqueSubSeq(vector, result, N, K, S, vectorPos + 1, resultPos);
    result[restultPos] ← vector[vectorPos];
    uniqueSubSeq(vector, result, N, K, S, vectorPos + 1, resultPos + 1);
```

## 4.5 Efficiency analysis

As we can observe from the algorithm itself, or even from the example below, we don't care in which order we add the number to obtain the necessary sum. That being said, we don't have permutations, but instead, we have combinations. In this case, the time complexity to solve the problem would be O($N!/(K! * (N - K)!)$). One way to improve this algorithm, but only speaking of spatial complexity, is to try and change the recursive approach to an iterative one.

## 4.6 Example of practical use

Given vector[] = [5, 8, 12], N = 3, K = 2, S = 20, print all the subsequences of length K with sum S.

For simplicity, we will have uniqueSubSeq = f, vector = v, result = r, vectorPos = vP, resultPos = rP.

* f(v, r, 3, 2, 20, 0, 0)
  0 == 2 (F)
  0 < 3 (A) $->$ f(v, r, 3, 2, 20, 1, 0)

* f(v, r, 3, 2, 20, 1, 0)
0 == 2 (F)
1 < 3 (A) − > f(v, r, 3, 2, 20, 2, 0)

* f(v, r, 3, 2, 20, 2, 0)
0 == 2 (F)
2 < 3 (A) − > f(v, r, 3, 2, 20, 3, 0)

* f(v, r, 3, 2, 20, 3, 0)
0 == 2 (F)
3 < 3 (F) − > we go back to f(v, r, 3, 2, 20, 2, 0)

* f(v, r, 3, 2, 20, 2, 0)
r[0] = v[2] = 12
f(v, r, 3, 2, 20, 3, 1)

* f(v, r, 3, 2, 20, 3, 1)
1 == 2 (F)
3 < 3 (F) − > we go back to f(v, r, 3, 2, 20, 2, 0)

* f(v, r, 3, 2, 20, 2, 0)
we've reached the end − > we go back to f(v, r, 3, 2, 20, 1, 0)

* f(v, r, 3, 2, 20, 1, 0)
r[0] = v[1] = 8
f(v, r, 3, 2, 20, 2, 1)

* f(v, r, 3, 2, 20, 2, 1)
1 == 2 (F)
2 < 3 (A) − > f(v, r, 3, 2, 20, 3, 1)

* f(v, r, 3, 2, 20, 3, 1)
1 == 2 (F)
3 < 3 (F) − > we go back to f(v, r, 3, 2, 20, 2, 1)

* f(v, r, 3, 2, 20, 2, 1)
r[1] = v[2] = 12
f(v, r, 3, 2, 20, 3, 2)

* f(v, r, 3, 2, 20, 3, 2)
2 == 2 (A) − > S2 = 8 + 12 = 20 = S − > Sol: 8, 12 − > we go back to f(v, r, 3, 2, 20, 2, 1)

* f(v, r, 3, 2, 20, 2, 1)

we've reached the end − > we go back to f(v, r, 3, 2, 20, 1, 0)

* f(v, r, 3, 2, 20, 1, 0)
we've reached the end − > we go back to f(v, r, 3, 2, 20, 0, 0)

* f(v, r, 3, 2, 20, 0, 0)
r[0] = v[0] = 5
f(v, r, 3, 2, 20, 1, 1)

* f(v, r, 3, 2, 20, 1, 1)
1 == 2 (F)
1 < 3 (A) − > f(v, r, 3, 2, 20, 2, 1)

* f(v, r, 3, 2, 20, 2, 1)
1 == 2 (F)
2 < 3 (A) − > f(v, r, 3, 2, 20, 3, 1)

* f(v, r, 3, 2, 20, 3, 1)
1 == 2 (F)
3 < 3 (F) − > we go back to f(v, r, 3, 2, 20, 2, 1)

* f(v, r, 3, 2, 20, 2, 1)
r[1] = v[2] = 12
f(v, r, 3, 2, 20, 3, 2)

* f(v, r, 3, 2, 20, 3, 2)
2 == 2 (A) − > S2 = 5 + 12 = 17 != S − > we go back to f(v, r, 3, 2, 20, 2, 1)

* f(v, r, 3, 2, 20, 2, 1)
we've reached the end − > we go back to f(v, r, 3, 2, 20, 1, 1)

* f(v, r, 3, 2, 20, 1, 1)
r[1] = v[1] = 8
f(v, r, 3, 2, 20, 2, 2)

* f(v, r, 3, 2, 20, 2, 2)
2 == 2 (A) − > S2 = 5 + 8 = 13 != S − > we go back to f(v, r, 3, 2, 20, 1, 1)

* f(v, r, 3, 2, 20, 1, 1)
we've reached the end − > we go back to f(v, r, 3, 2, 20, 0, 0)

* f(v, r, 3, 2, 20, 0, 0)
we've reached the end − > the algorithm has finished its execution

## 5 Comparative Analysis

### 5.1 Divide and Conquer

The Divide and Conquer algorithm is mainly used to deal with difficult problems by solving recursively the trivial cases and combining subproblems to the original problem.

The main advantages of this algorithm are:
* the efficiency: it was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, improving the asymptotic cost of the solution.
* parallelism: this algorithm can be used without any problems for execution in multi-processor machines, because the subproblems can be executed on different processors.

The main disadvantages of this algorithm are:
* time: sometimes the recursion is slow, outweighing the advantages.
* memory: if the problem is too large, the recursion calls might cause trouble when speaking about the memory used for the subproblems.
* solving the same subproblem multiple times: sometimes it is easier to save the solution to the repeated subproblems, using memoization.

### 5.2 Greedy

The Greedy algorithm is mainly used in optimization problems, always making a local optimal choice in hope that this will lead to a global optimal solution.

The main advantages of this algorithm are:
* finding the solution with this approach is easier than with others.
* less time complexity and an easier analyze of it (compared to Divide and Conquer).

The main disadvantages of this algorithm are:
* even with the correct algorithm, it is hard to prove why it is correct.

* no certainty that local optimal solution can be global optimal solution.

### 5.3 Dynamic Programming

The Dynamic Programming algorithm is mainly used to solve problems which can be split into multiple small problems, like the Divide and Conquer approach. The difference between them is that this time, the results of the subproblems are stored into a result array.

The main advantages of this algorithm are:
* it uses the memoization technique to store previous results for later use when coming across the same subproblem.
* reduction of time complexities from exponential to polynomial.
* it is exhaustive and guaranteed fo find the solution, unlike the Greedy approach.

The main disadvantages of this algorithm are:
* harder to use than other methods.
* more memory is used, not knowing if the previous values will be used.

### 5.4 Backtracking

The Backtracking algorithm is mainly used to solve computational/ constraint satisfaction problems which require to find all/ some solutions. It can only be applied to problems that admit the concept of a partial candidate solution.

The main advantages of this algorithm are:
* very easy to code: sometimes only few lines of recursive function code is necessary.
* computes all possible solutions, not only the best one.

The main disadvantages of this algorithm are:
* there can be other optimal algorithms for the given problem with better time complexities.
* because of recursion, it has a large space complexity.

### 5.5 Example of a problem that can be solved using multiple techniques of the ones above

Greedy vs Dynamic Programming

Given a set of activities with a start and an end time, find the maximum number of activities that don't intersect.

**Algorithm 1:** Greedy approach for the maximum number of non-intersecting activities problem

```
Data: activities: List of activites
Result: Returns the maximum number of non-intersecting activites
sortOnEndTime(activities)
currentEndTime ← $-\infty$
foreach activity ∈ activities do
    if currentEndTime < activity.startTime then
        result ← result + 1
        currentEndTime ← activity.endTime
    end
end
return result
```

**Algorithm 2:** Dynamic programming approach for the maximum number of non-intersecting activities problem

```
Data: n: Number of activities, activities: List of activites
Result: Returns the maximum number of non-intersecting activites
sortOnStartTime(activities);
DP[n + 1] ← 0
for i ←n to 1 do
    pick ← DP[i + 1]
    nextChoice ← getNext(i)
    leave ← 1 + DP[nextChoice]
    DP[i] ← max(pick, leave)
end
return DP[0];
```

For the Greedy approach, the way to obtain the optimal solution is to sort the activities in ascending order by their ending time. Choosing at every step the activity that will end first will offer more activities for the next steps. The time complexity for this algorithm is O(n $*$ logn) given by the sorting algorithm because the iteration complexity is O(n).

For the Dynamic approach, the way to obtain the optimal solution is to sort the activities in ascending order by their starting time. At the position i we will have stored in the dp vector the maximum number of activities in the range [i, n]. The time complexity for this algorithm is O(n$*$ logn) because the function getNext is called for every step from n to 1, having a time complexity of O(logn) (binary search).

| | **Greedy Approach** | **Dynamic Programming** |
|---|---|---|
| **Main Concept** | Choosing the best option that gives the best profit for the current step | Optimizing the recursive backtracking solution |
| **Optimality** | Only if we can prove that local optimality leads to global optimality | Gives an optimal solution |
| **Time Complexity** | Polynomial | Polynomial, but usually worse than the greedy approach |
| **Memory Complexity** | More efficient because we never look back to other options | Requires a DP table to store the answer of calculated states |

## References

[1] https://en.calameo.com/read/005335614a0594aa97420

[2] https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm

[3] https://en.wikipedia.org/wiki/Greedy_algorithm

[4] https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[5] https://www.geeksforgeeks.org/minimize-the-cost-to-make-all-the-adjacent-elements-distinct-in-an-

[6] https://www.geeksforgeeks.org/backtracking-introduction/

[7] https://www.geeksforgeeks.org/unique-subsequences-of-length-k-with-given-sum/

[8] https://www.kodnest.com/free-online-courses/algorithm-2/lessons/
advantages-and-disadvantages-of-divide-and-conquer/

[9] https://developerinsider.co/introduction-to-greedy-algorithms/

[10] https://medium.com/techie-delight/backtracking-practice-problems-and-interview-questions-6a17cb6

[11] https://www.baeldung.com/cs/greedy-approach-vs-dynamic-programming