Using Dynamic Programming to Choose Activities Effectively

Using Dynamic Programming to Choose Activites Effectively From a List of Activity Where Every Activity Has Its Own Considered Attributes

Muhammad Hasan (13518012)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
muhammadhasan50@gmail.com

***Abstract*—Every person has their own schedule of activities that they want to do. However, given the time they have, it's not always possible to do all the activities in their schedule, so choosing some activities in their schedule is the only option. Every activity might have its own priority value and the time needed to finish that activity, so it could be quite hard to choose activity effectively. This is where Dynamic Programming comes in place, by using Dynamic Programming we could maximize the total priority value in choosing the activities while also considering the time taken, thus giving us the option of choosing activities effectively.**

Keywords—dynamic programming, activity, priority, effective

I. Introduction

A list of activity is used to take an overview of all activities that is considered to be finished. Take for an example, a college student is given many tasks to do and they are all needed to be done by one month, in this case, that student might have a list of activities in one month including the task mentioned and maybe also including the other outer task the student has.

Choosing activities is sometimes a burden, when all activities cannot be done in the given time. It might not be good to choose activities randomly without any consideration, because every activity might have its own priority value, and choosing randomly could always be ineffective.

There are many considered attributes an activity could have. For an example, an activity could have an attributes such as the priority value and the time needed to finish the activity. So choosing activities effectively could be very hard if there is a lot of activities to be done. While this problem might seem hard, but this type of problem is actually a pretty good demonstration problem that can be solved with dynamic programming.

Dynamic Programming is one of the best strategies to solve problems in maximizing/minimizing value while also considering attributes and constraints. In this paper, the author would like to show how dynamic programming could solve various problem on choosing activities.

II. Theory

A. *Choosing Activity Problem*

This problem is actually a self-thought problem and could be generalized as the following:

> Given a positive integer n denoting the number of activities to be chosen, and a list of activities with n activites:
>
> $$a_1, a_2, a_3, \dots, a_n$$
>
> Every activity has the same attributes set (S), that defines the property of the activity. The set S must have a priority value as a member in it. The value of an attribute m from the activity a is denoted with $a(m)$ where a is an activity and $m \in S$. We are also given a constraint statement (C) that must not be violated.
>
> Choose the optimal sequence of activities in the given list of activities that gives the highest total priority value while also fulfilling constraints in C.

Let's have an example, consider a list of activities with 5 activity:

$$a_1, a_2, a_3, a_4, a_5$$

Every activity has the attribute set $S = (p, t)$. Where p is a priority value and t is the time taken to finish the activity (in some unit time). Let's say we have the attributes assigned to a value in this table below:

a (activity)	p (priority value)	t (time taken)
1	10	5
2	40	4
3	30	6
4	50	3
5	70	13

The constraint statement (C) is as follows, The total time taken from the activity chosen must not exceed 10. With all that in mind, find the optimal subset!

Formally, if R is the set of activities chosen then we must have $\sum_{a \in R} a(t) \leq 10$ and the value of $\sum_{a \in R} a(p)$ is maximized.

Let's try out some solution, consider choosing R as the subset solution with:

$$R = (a_1, a_2, a_5)$$

We will have a total priority value of:

$$\sum_{a \in R} a(p) = a_1(p) + a_2(p) + a_5(p)$$
$$= 10 + 40 + 70$$
$$= 120$$

But keep in mind that we would have a total time taken value of:

$$\sum_{a \in R} a(p) = a_1(t) + a_2(t) + a_5(t)$$
$$= 5 + 4 + 13 = 22 > 10$$

This violates C and cannot be considered as a solution.

If we try to list all solution possibility that does not violates C, then we could have the possible solution shown at this table:

R (subset chosen)	Total priority value	Total time taken
(a_1)	10	5
(a_2)	40	4
(a_3)	30	6
(a_4)	50	3
(a_1, a_2)	50	9
(a_1, a_4)	60	8
(a_2, a_3)	70	10
$\boldsymbol{(a_2, a_4)}$	**90**	**7**
(a_3, a_4)	80	9

From the table above, we could conclude that the optimal subset to choose is (a_2, a_4) with a total priority value of 90, and a total time taken of 7.

The example shown is only one of the possibilities that the choosing activity problem have, in other cases it could have a different set of attributes for the activity and also a different constraint statement.

B. Dynamic Programming

Dynamic Programming is an algorithmic technique often used to solve various problems. The algorithmic technique behind dynamic programming is usually based on a starting state of the problem, and a recurrent formula or relation between successive states. A state of the problem usually represents a sub-solution, i.e. a partial solution or a solution based on a subset of a given input. The states are built one by one, based on the previously built states [1].

Dynamic programming is usually used for two types of problem [2]:

- **Finding an optimal solution**: We want to find a solution that is as large/small as possible.
- **Counting the number of solution**: We want to calculate the total number of possible solution.

One of the concept of dynamic programming is that if sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problem [3].

To have a better understanding in dynamic programming, let's try out solving a classical problem called the 0-1 Knapsack Problem with dynamic programming.

The 0-1 Knapsack Problem is generally described as the following:

Given a set of n items, numbered from 1 up to n, each with a weight value of w_i and a value of v_i, along with a maximum capacity of W. We would like to have:

$$\sum_{i=1}^{n} v_i x_i \text{ maximized}$$

While also having:

$$\sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}$$

Here x_i represents the number of instances of item i to be included in the knapsack.

Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weight is less than or equal to the knapsack's capacity

Now, let us put values into the general problem so that we could have an example to work with. Let say we have a knapsack capacity of $W = 8$, and 4 items with given values shown in the table below:

Item	Value	Weight
1	2	1
2	2	3
3	5	4
4	6	5

One of the possible solution is to choose the item 2 and the item 4, this gives us a total value of 8 and a total weight of 8. However, this is not the optimal solution.

The optimal solution is to choose the item 1, item 2, and item 3, this gives us a total value of 9 and a total weight of 8.

To solve this problem with dynamic programming we will have find a good defined state that we could work with. Notice that, the important thing is to consider taking or not taking an item and also to consider the knapsack capacity. So, we could have a dynamic programming state as the following:

$$dp_{i,j} = \text{max value of item 1 to } i \text{ with a knapsack value of } j$$

In this definition, $dp_{i,j}$ is used to store the maximum value of item we will have if we only consider item 1 to item i with a knapsack value of j.

Now that we have a defined state, we'll have to find a recursive statement to find the relationship between states. First, we'll have to consider the base state. If we have zero element, then we will always have:

$$dp_{0,j} = 0$$

No matter what the knapsack weight is, we can't have any value to take from so the value item is always zero. And if the knapsack weight is zero, we would also have:

$$dp_{i,0} = 0$$

Logically, every item must have a weight to it, so we can't actually take any item if the knapsack weight is zero, thus giving us a value item of zero.

If we have at least one element, then we can either take the element or not take that element. It depends on the condition. So we could have a recursive statement as given in the following equation:

$$dp_{i,j} = \begin{cases} dp_{i-1,j}, \text{if } w_i > j \\ \max\left(dp_{i-1,j}, dp_{i,j-w_i} + v_i\right), \text{if } w_i \leq j \end{cases}$$

From here we are actually finished with our dynamic programming statements. Let's wrap it up into one equation:

$$dp_{i,j} = \begin{cases} 0, \text{if } i = 0 \text{ or } j = 0 \\ dp_{i-1,j}, \text{if } w_i > j \\ \max\left(dp_{i-1,j}, dp_{i,j-w_i} + v_i\right), \text{if } w_i \leq j \end{cases}$$

It's easy to implement this in code, now that we have the formula. And once we have the answer, we could use backtrack to find the sequence of activity.

There is actually two approach of implementing dynamic programming, one is the *top down* approach and the other is the *bottom up* approach.

The top down approach is essentially filling up values from top to down while the bottom up approach is the opposite. In Author's opinion, it's a good practice to always use the bottom up approach using iterative rather than to use the top down approach while using the recursive function. This usually makes faster run time, because using recursive function would need additional timework from using stacks.

Now let's try making a pseudocode of this algorithm with the bottom up approach. The key here is to iterate from zero to n with the i values, and also to iterate from zero to W with the j values. So we will have to for loop and within those two loops we will use the equation we have. The final answer will be the value of $dp_{n,W}$. So this is the pseudocode we will have:

```
# Fill in dynamic programming tables
for i in [0...n]:
 for j in [0...w]:
  if i==0 or j==0:
   dp[i][j]←0
  else if w[i]>j:
   dp[i][j]←dp[i-1][j]
  else:
   dp[i][j]←max(dp[i-1][j],dp[i][j-
w[i]]+v[i])

max_value←dp[n][w]

# Backtrack to find sequence answer
i←n
j←W
answer←[]
while i>0:
 if dp[i][j]==dp[i-1][j]:
  # Skip activity-i
  i←i-1
 else:
  # Take activity-i
  answer.append(i)
  j←j-w[i]
  i←i-1
reverse(answer)
```

Figure 1. Pseudocode of Using Dynamic Programming to Solve 0-1 Knapsack

It is easy to see that this algorithm will run in $O(nW)$ time complexity and also the same for its memory complexity, where n is the number of item and W is the capacity of the knapsack.

Notice that, $dp_{i,j}$ is actually a table to be filled with. So if we implement this algorithm we would have the following table values:

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	2	2	2	2	2	2	2	2
2	0	2	2	2	4	4	4	4	4
3	0	2	2	2	5	7	7	7	9
4	0	2	2	2	5	7	8	8	**9**

We could see from the table values that the answer is:

$$dp_{n,W} = dp_{4,8} = 9$$

The good thing about using dynamic programming here is that we could actually see solution for sub-problems too.

This 0-1 Knapsack Problem is actually very similar to the example problem in the choosing activity problem, so we could already see where the dynamic programming would be used.

III. Using Dynamic Programming to Solve Choosing Activity Problem

In this section we will try solving several choosing activity problems by using dynamic programming.

A. *Choosing Activity Problem with Attributes Including Priority Value and Time Taken to Finish Activity*

To start off, let us demonstrate the use of dynamic programming with the problem given in the example problem of choosing activity problem. The problem is stated as below:

> Given a list of activity of n activites:
>
> $$a_1, a_2, a_3, \dots, a_n$$
>
> Every activity has an attribute set S:
>
> $$S = (p, t)$$
>
> Where $p \rightarrow$ priority value and $t \rightarrow$ time taken to finish activity.
>
> Find a subset of activities chosen such that the total time taken does not exceed T and the total priority value is maximized.

Let's have the same values given in the problem example. We will have $T = 10$, and the other values are shown in the following table:

a (activity)	p (priority value)	t (time taken)
1	10	5
2	40	4
3	30	6
4	50	3
5	70	13

Notice that this problem is pretty similar to what was given in the 0-1 Knapsack Problem Statement. The key here is that the time is actually equivalent to the weight in the knapsack problem. So, we could actually use the exact method as before.

Let us define $dp_{i,j}$ as the maximum total value from a_1 up to a_n with $T = j$, then we will have the following equation:

$$dp_{i,j} = \begin{cases} 0, \text{if } i = 0 \vee j = 0 \\ dp_{i-1,j}, \text{if } a_i(t) > j \\ \max\left(dp_{i-1,j}, dp_{i,j-a_i(t)} + a_i(p)\right), \text{if } a_i(t) \leq j \end{cases}$$

In this definition:

$a_i(t) =$ the value of time attribute of activity-i

$a_i(p) =$ the value of priority value of activity-i.

Then we could have the following pseudocode:

```
# Fill in the dynamic programming tables
for i in [0...n]:
 for j in [0...T]:
  if i==0 or j==0:
   dp[i][j]←0
  else if a[i].t>j:
   dp[i][j]←dp[i-1][j]
  else:
   dp[i][j]←max(dp[i-1][j],dp[i][j-
a[i].t]+a[i].p)

max_value←dp[n][T]

# Backtrack to find sequence answer
i←n
j←T
answer←[]
while i>0:
 if dp[i][j]==dp[i-1][j]:
  # Skip activity-i
  i←i-1
 else:
  # Take activity-i
  answer.append(i)
  j←j-a[i].t
  i←i-1
reverse(answer)
```

Figure 2. Pseudocode of Dynamic Programming to Solve Choosing Activity Problem Section III.A.

The algorithm implemented will have a time complexity and a memory complexity of $O(nT)$ where n is the number of activity and T is the time limit.

The answer to the problem is $dp_{n,T}$. The value of $dp_{i,j}$ is given in the following table:

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	40	40
3	0	0	0	30	40	40	40	70	70	70	70
4	0	0	0	30	40	40	50	70	70	80	90
5	0	0	0	30	40	40	50	70	70	80	**90**

We could see from the table that the answer is:

$$dp_{n,T} = dp_{5,10} = 90$$

Which is the same correct answer found in the example problem explanation.

B. Choosing Activity Problem with Attributes Including Priority Value, Time Taken to Finish Activity, and Deadline Time of The Activity.

Consider having a deadline for an activity. If the time is over the due deadline, then that activity can no longer be taken. This is often found when having tasks with different deadlines. In general, the problem in this section can be stated as the following:

Given a list of activities of n activities:

$$a_1, a_2, a_3, \dots, a_n$$

Every activity has an attribute set S:

$$S = (p, t, d)$$

Where:

$$p = \text{priority value}$$

$$t = \text{time taken to finish activity}$$

$$d = \text{deadline of the activity}$$

Find an optimal sequence of activities (R), such that the total priority value is maximal. Take in mind the constraint (C) for this problem is that for every $a \in R$, the time in which we take the activity (s) is valid, that is:

$$s \leq a(t) - a(d)$$

Informally, every time an activity is taken, it must be a valid time taken.

Let's take an example, for the given value shown in this table:

a (activity)	p (priority value)	t (time taken)	d (deadline)
1	4	3	7
2	5	2	6
3	6	3	7

In this example, we cannot take all the activity, because it will violate C. So, let's try taking two activity, assume we take a_1 then a_2 then we will have a total priority value of 9. If we try to take all possibility, we would have an optimal solution by taking a_3 then a_1 giving us a total priority value of 11.

Notice that, the order of taking the activity matters, so any two subsets having the same element with different order might have different take on the constraint. So, with that in mind, it's a good idea to sort ascending the activity first by their due deadline.

Now that the activities are sorted by their deadline, we could have better idea of making a recursion. But first, Let's try to find a good state of dynamic programming to work with.

We could see that time is an important consideration to take on this problem, so for every time we might have a different maximum answer. As a result, we could have a state of dynamic programming defined as the following:

$$dp_{i,j} = \max \text{value of first } i \text{ activites at time } j$$

In this definition, $dp_{i,j}$ is used to store the maximum priority value considering $a_1, a_2, \dots, a_i$ (after it is sorted) at time j.

For the base case, if we have either zero activity or zero time than we would have:

$$dp_{i,j} = 0, \text{if } i = 0 \ \vee \ j = 0$$

The recursion or transition we have will take consider of taking the current activity or not, so we will have a transition as follow:

$$dp_{i,j} = \begin{cases} dp_{i-1,j}, \text{if } a_i(t) > j \\ \max(dp_{i-1,j}, dp_{i-1,j-a_i(t)} + a_i(p)), \text{if } a_i \leq j \wedge j < a_i(d) \end{cases}$$

This is actually quite similar to the definition we have in 0-1 Knapsack Problem.

To wrap all dynamic programming states, we will have the following equation:

$$dp_{i,j} = \begin{cases} 0, \ \text{if } i = 0 \vee j = 0 \\ dp_{i-1,j}, \text{if } a_i(t) > j \\ \max(dp_{i-1,j}, dp_{i-1,j-a_i(t)} + a_i(p)), \text{if } a_i \leq j \wedge j < a_i(d) \end{cases}$$

Notice that the answer for this problem can be in any time j, since j have to be less than or equal to the largest deadline, we could say $j \leq T$, where T = longest deadline in the activity. So we will have:

$$\text{answer} = \max_{0 \leq j \leq T} dp_{n,j}$$

Now all that left is to implement the code. Note that, because we sort the list of activities, we might have the number order activity to be different. So, let us store initial index of activity denoted by $a_i(idx)$. Here is the pseudocode:

```
# sort the activity by their deadline first
sort(a, deadline)
# set T as the largest deadline value
T←a[n].d
# fill in the dynamic programming tables
for i in [0...n]:
 for j in [0..T]:
  if i==0 or j==0:
   dp[i][j]←0
  else:
   dp[i][j]←dp[i-1][j]
   if j<a[i].d and j-a[i].t>=0:
    dp[i][j]←max(dp[i][j],dp[i-1][j-
a[i].t]+a[i].p)
# find maximum priority value
max_value←0
pos←0
for i in [0...T]:
 if dp[n][i]>max_value:
  max_value←dp[n][i]
  pos←i
# use backtrack to find chosen activities
answer←[]
i←n
j←pos
while i>0:
 if dp[i][j]==dp[i-1][j]:
   # skip activity
   i←i-1
 else:
   # take activity
   answer.append(a[i].idx)
   j←j-a[i].t
   i←i-1
reverse(answer)
```

Figure 3. Pseudocode of Dynamic Programming to Solve Choosing Activity Problem Section 3.B.

We could see that the algorithm used here will have a time complexity and a memory complexity of $O(nT)$ where n is the number of activities and T is the longest deadline from the list of activity.

The value $dp_{i,j}$ is given in the following table:

i \ j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	0	0
2	0	0	5	5	5	9	9	0
3	0	0	5	6	6	**11**	**11**	0

We could see from the table above that the answer is 11, which is to take a_3 then a_2.

C. Choosing Activity Problem with Attributes Including Priority Value, Start Time, and End Time.

All the problems we have solved, have the same start time that the activity could be taken. Consider every activity to have its own start time and end time. In those range time from start time to end time, we can only do one activity.

In this problem, we can't have two activities colliding together at the same time. An example of this problem is when we are needed to choose some activities in which every activity has its own fixed time to finish that activity.

In general, the problem in this section can be stated as the following:

Given a list of activity with n activities:

$$a_1, a_2, a_3, \ldots, a_n$$

Every activity has an attribute set S:

$$S = (p, s, e)$$

Where:

$$p = \text{priority value}$$

$$s = \text{start time}$$

$$e = \text{end time}$$

Find an optimal sequence of activities (R), such that the total priority value is maximal. Take in mind the constraint (C) for this problem is that there is no collision between any activities in R

We will have an example for the activity values as shown in this following table:

a (activity)	p (priority value)	s (start time)	e (end time)
1	2	1	4
2	5	5	6
3	1	2	3
4	8	4	9
5	1	7	9

From this example, it could be proven that, the optimal way we could choose is to take a_3 then a_4 giving us a maximum total priority of 9.

Notice that, it is better to sort the activity first by their start time, so we could have a more organized activity. This sorting is also useful so that we can see the transition between states easier.

After sorting the activity, notice that for every activity a_i and a_j $(i < j)$ (in the order of sort) we could combine this activity if and only if $a_i(e) < a_j(s)$. By using this fact, we could have a defined state of dynamic programming as follow:

$$dp_i = \text{maximum value having } a_i \text{ and it's combination}$$

With this defined state, it's actually easier to construct the dynamic programming properties than what we have solved previously.

Let us define the base state for this problem. If we have zero activity then:

$$dp_0 = 0$$

Otherwise we could iterate back to the value we have before and combine it.

$$dp_i = a_i(p) + \max_{0 \le j < i}(dp_j, \text{where } a_j(e) < a_i(s))$$

Note that, for every $1 \le i \le n$ we will have to make the inequality $a_0(e) < a_i(s)$ to always hold true.

To implement this code, remember that because we sort the list of activities, we might have the number order activity to be different. So, let us store every initial index of activity denoted by $a_i(idx)$.

We could use two for loops and an array to store transition, so we could backtrack and find the sequence solution. Here is the pseudocode:

```
# sort the activity by start time first
sort(a, start_time)
# fill in the dynamic programming tables
back = [0 for i in [0...n]]
for i in [0...n]:
 if i == 0:
  dp[i]←0
 else:
  dp[i]←a[i].p
  for j in [0...i]:
   if j==0 or a[j].e<a[i].s:
    cur←dp[j]+a[i].p
    if cur>dp[i]:
     dp[i]←cur
     back[i]←j
# find the maximum priority value
max_value←0
pos←0
for i in [0...n]:
 if dp[i]>max_value:
  max_value←dp[i]
  pos←i
# backtrack to find sequence activity
answer←[]
i←pos
while i>0:
 answer.append(a[i].idx)
 i←back[i]
reverse(answer)
```

Figure 4. Pseudocode of Dynamic Programming to Solve Choosing Activity Problem Section 3.C.

We could see that the implemented algorithm will have a time complexity of $O(n^2)$ and a memory complexity of $O(n)$ where n is the number of activity.

An interesting thing about this problem is we could actually reduce the time complexity to $O(n \log n)$ by using an advanced data structure such as *segment tree*, but it will not be explained further in this paper.

The value for dp_i is given in the following table:

i	Value
1	2
2	1
3	**9**
4	7
5	8

We could see from the table above that the maximum total priority value is 9, this is of course by chosing a_3 then a_4.

IV. Program Testing

This section will show a screenshot of program testing for every problem in section III. The code is implemented with C++ and is tested with the given PC specification:

Component	Description
Operating System	Windows 10 Home Single Language 64-bit
CPU	Intel Core i7 @ 1.80GHz Kaby Lake-U/Y 14nm
RAM	16GB DDR3
Motherboard	ASUSTek COMPUTER INC. UX430UNR (UE31)
Storage	512GB SanDisk SSD
Graphics	Intel UHD Graphics 620 2047MB NVIDIA GeForce MX150

All of the source code and test cases in this section could be seen at https://github.com/muhammadhasan01/IF2211-Strategi-Algoritma/tree/master/ProgramMakalah.

A. *Program Testing for Problem Section III.A*

- Test Case I

In this test case, it will include the exact same value given in the example problem. The program will read the file testSectionIIIA_1.in. Here is the output of the program:

```
Time Limit (T) = 10
There are 5 activities with a format of a = (p, t):
(10,5),(40,4),(30,6),(50,3),(70,13)

Maximal total priority value = 90
The optimal chosen sequence of activity is 2 activities:
2 4
Execution Time : 0.00000ms
```

Figure 5. Screenshot Output for Test Case I Problem III.A.

- Test Case II

In this test case, we will have $n = 100$ activities with $T = 100$. The program will read the file testSectionIIIA_2.in. Here is the output of the program:

```
Time Limit (T) = 100
There are 100 activities with a format of a = (p, t):
(73,12),(37,79),(66,56),(33,30),(34,69),(37,50),(80,59),(58,
16),(43,42),(81,49),(95,29),(13,4),(19,93),(37,6),(38,92),(7
1,93),(96,8),(83,6),(43,72),(52,31),(47,94),(37,72),(41,19),
(7,97),(72,92),(88,85),(90,88),(19,52),(35,60),(94,43),(75,1
8),(60,38),(32,45),(19,73),(95,20),(23,64),(39,59),(23,51),(
24,71),(89,74),(83,45),(75,79),(35,11),(51,98),(39,95),(40,3
5),(47,41),(66,8),(89,81),(68,22),(37,25),(41,82),(3,77),(84
,36),(10,24),(59,92),(20,11),(74,64),(68,64),(73,1),(28,64),
(20,13),(26,6),(77,46),(78,99),(100,68),(90,7),(64,58),(25,9
6),(13,69),(68,83),(50,23),(52,30),(62,66),(81,27),(63,60),(
87,98),(94,100),(13,34),(19,79),(31,42),(7,25),(73,77),(89,6
5),(69,66),(87,30),(75,5),(85,8),(42,66),(48,6),(7,77),(60,3
2),(45,95),(7,98),(6,45),(61,93),(1,43),(84,78),(61,59),(61,
24)

Maximal total priority value = 860
The optimal chosen sequence of activity is 13 activities:
1 12 14 17 18 35 48 60 63 67 87 88 90
Execution Time : 0.00000ms
```

Figure 6. Screenshot Output for Test Case II Problem III.A.

- Test Case III

In this test case, we will have $n = 1000$ activities with $T = 1000$. The program will read the file testSectionIIIA_3.in. Because, there is too much of the output, the screenshot output will only include the results, and so here is the output of the program:

```
Maximal total priority value = 25745
The optimal chosen sequence of activity is 41 activities:
20 32 74 79 86 103 151 155 251 281 287 291 326 360 365 407
456 468 469 487 488 523 547 640 642 649 657 671 712 760 773
 776 808 825 831 843 851 877 935 960 993
Execution Time : 25.00000ms
```

Figure 7. Screenshot Output for Test Case III Problem III.A.

B. *Program Testing for Problem Section III.B*

- Test Case I

In this test case, it will include the exact same value given in the example problem. The program will read the file testSectionIIIB_1.in. Here is the output of the program:

```
There are 3 activities with a format of a = (p, t, d):
(4 3 7),(5 2 6),(6 3 7)

Maximum total priority value = 11
The optimal chosen sequence of activity is 2 activities:
2 3
```

Figure 8. Screenshot Output for Test Case I Problem III.B.

- Test Case II

In this test case, we will have $n = 100$ activities and random values ranging between 1 to 100. The program will read the file testSectionIIIB_2.in. Here is the output of the program:

```
There are 100 activities with a format of a = (p, t, d):
(29 69 36),(67 69 19),(40 35 6),(64 86 21),(73 66 52),(27 25 79),
(25 24 70),(70 31 42),(95 75 67),(76 54 15),(38 83 94),(38 24 10)
,(95 96 13),(18 46 17),(51 50 9),(60 13 47),(57 80 14),(71 72 14)
,(22 58 58),(54 52 53),(66 78 20),(72 25 6),(45 53 10),(48 40 29)
,(98 95 36),(57 25 62),(17 61 74),(41 80 39),(89 32 83),(29 83 53
),(3 22 16),(31 34 77),(33 37 40),(31 15 33),(26 78 10),(91 68 5)
,(11 18 84),(47 31 88),(41 27 95),(29 3 82),(70 10 41),(51 28 85)
,(3 73 73),(32 79 86),(29 85 11),(69 70 2),(93 4 67),(27 29 10),(
7 67 29),(76 73 90),(31 93 12),(58 86 93),(49 77 47),(97 82 40),(
9 80 52),(58 44 78),(86 82 16),(73 65 13),(24 77 90),(59 62 58),(
26 15 22),(52 49 97),(48 16 58),(49 37 17),(76 23 28),(69 89 1),(
32 70 40),(46 8 40),(81 91 35),(7 12 72),(4 21 89),(86 64 98),(58
 24 21),(14 64 52),(66 33 36),(66 61 93),(88 9 98),(38 47 50),(27
 85 16),(90 76 60),(34 90 52),(71 43 33),(27 32 4),(57 94 68),(62
 33 5),(79 70 83),(100 52 91),(7 84 100),(29 73 22),(64 100 72),(
28 76 78),(89 36 92),(68 29 14),(33 15 32),(21 84 3),(39 51 16),(
28 90 16),(23 88 32),(41 82 16),(38 67 12)

Maximum total priority value = 494
The optimal chosen sequence of activity is 7 activities:
68 41 16 63 47 92 77
```

Figure 9. Screenshot Output for Test Case II Problem III.B.

- Test Case III

In this test case, we will have $n = 1000$ activities. With random values ranging between 1 to 1000. The program will read the file testSectionIIIA_3.in. Because, there is too much of the output, the screenshot output will only include the results, and so here is the output of the program:

```
Maximum total priority value = 26282
The optimal chosen sequence of activity is 42 activities:
279 13 233 633 601 780 363 531 85 810 220 636 628 997 88 153 345
 11 679 838 143 34 712 752 267 330 95 761 596 29 687 412 591 797
 683 955 870 843 269 2 894 163
```

Figure 10. Screenshot Output for Test Case III Problem III.B.

C. Program Testing Problem Section III.C

- Test Case I

In this test case, it will include the exact same value given in the example problem. The program will read the file testSectionIIIC_1.in. Here is the output of the program:

```
There are 5 activities with a format a = (p, s, e):
(2,1,4),(5,5,6),(1,2,3),(8,4,9),(1,7,9)

Maximum total priority value = 9
The optimal chosen sequence of activity is 2 activities:
4 3
Execution Time: 0.00000ms
```

Figure 11. Screenshot Output for Test Case I Problem III.C.

- Test Case II

In this test case, we will have $n = 100$ activities with start time and end time ranging between values of 1 to 2000. The program will read the file testSectionIIIC_2.in. Here is the output of the program:

```
There are 100 activities with a format a = (p, s, e):
(62,78,133),(54,60,126),(15,88,155),(85,79,136),(97,31,38
),(69,27,62),(79,41,71),(21,46,99),(3,100,110),(28,49,64)
,(84,5,10),(3,37,121),(61,71,100),(58,20,55),(63,9,67),(4
,38,109),(37,48,55),(62,95,169),(98,28,124),(12,25,38),(3
0,89,144),(69,23,63),(37,10,74),(73,26,124),(55,43,123),(
28,26,64),(66,72,170),(98,32,72),(12,75,165),(90,63,134),
(13,44,114),(78,66,80),(25,89,120),(89,50,83),(74,87,179)
,(61,92,174),(15,1,48),(2,78,167),(60,50,148),(90,52,102)
,(91,95,143),(74,20,95),(71,22,44),(62,94,159),(45,57,83)
,(66,98,164),(25,94,126),(100,81,172),(70,23,81),(38,32,9
1),(86,86,132),(54,8,23),(9,43,51),(37,2,70),(24,65,130),
(87,7,18),(30,97,159),(25,39,58),(40,71,80),(82,62,64),(3
,64,139),(73,25,36),(55,26,78),(82,31,104),(82,42,122),(6
3,48,87),(56,2,57),(42,54,113),(44,17,30),(72,72,149),(91
,30,84),(91,50,68),(62,58,77),(12,61,98),(26,80,98),(72,5
8,103),(49,75,132),(74,82,99),(74,95,139),(22,1,53),(26,8
0,115),(16,90,125),(81,64,156),(4,58,119),(21,94,161),(95
,73,143),(69,88,94),(24,8,77),(42,3,30),(46,94,132),(79,7
,27),(79,45,83),(58,68,118),(80,96,180),(53,80,92),(11,72
,102),(71,44,71),(43,55,115),(10,60,108),(23,79,151)

Maximum total priority value = 582
The optimal chosen sequence of activity is 8 activities:
41 87 32 60 17 5 69 11
Execution Time: 0.00000ms
```

Figure 12. Screenshot Output for Test Case II Problem III.C.

- Test Case III

In this test case, we will have $n = 1000$ activities. The start time and end time will have a value ranging in 1 to 2000. The program will read the file testSectionIIIC_3.in. Because, there is too much of the output, the screenshot output will only include the results, and so here is the output of the program:

```
Maximum total priority value = 18660
The optimal chosen sequence of activity is 27 activities:
268 187 864 136 428 663 739 448 781 541 357 845 899 838 746
 989 960 505 581 76 902 80 714 1 954 107 630
Execution Time: 2.00000ms
```

Figure 13. Screenshot Output for Test Case III Problem III.C.

V. CONCLUSION

Choosing a lot of activity is sometimes very hard for us to handle ourselves. However, by using dynamic programming we could not only solve the problem effectively, but we could also see the entire process and the sub-problem that includes it.

VIDEO LINK AT YOUTUBE

To see a brief video about this paper, please refer to this link https://www.youtube.com/watch?v=Z5uXfaNTAR4&feature=youtu.be

ACKNOWLEDGEMENT

First and foremost, I would like to thank Allah Azza wa Jalla for the opportunity that He has given me, so that I could undertake all challenges and embrace all the support I had up until now. I would also like to thank Mr. Rinaldi Munir as the lecturer of Algorithm Strategy (IF2211) and to everyone involved, for their time and struggle in teaching and guiding me, so that I could learn many new knowledge about algorithms and also as a guidance for me to complete this paper.

REFERENCES

[1] Kapoor, Karan. Everything About Dynamic Programming. https://codeforces.com/blog/entry/43256 accessed on May 1, 2020

[2] Laaksonen, Antti. Competitive Programming Handbook. Draft July 3, 2018.

[3] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), Introduction to Algorithms (2nd ed.), MIT Press & McGraw–Hill, ISBN 0-262-03293-7 . pp. 344.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 02 Mei 2020

Muhammad Hasan (13518012)