**Algorithms** Fall 2019

# Lecture : Dynamic Programming

*Imdad Ullah Khan*

## Contents

# 1 Introduction

Many of the problems that we deal with in computing often involve solving problems of a smaller size before we are able to arrive at the solution. Methods to solve such problems often involve some form of recursion or iteration to arrive at the final result. A form of algorithmic design that we will look in this series of notes is called **Dynamic Programming**, which involves two key components, the substructure of the problem, and the process of memoization.

## 1.1 Recurrence relations revisited

We once again visit the Fibonacci sequence of numbers, which is a sequence of numbers where the $n$th Fibonacci number, $F_n$, is the sum of the $(n-1)$th Fibonacci number, $F_{n-1}$, and the $(n-2)$th Fibonacci number, $F_{n-2}$, provided $n > 2$. For $n = 0$ we have $F_0 = 0$, and for $n = 1$, we have $F_1 = 1$. Concisely, we have:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 \text{ if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}.$$

Generally, a naive method of computationally determining $F_n$ involves recursion, as in the pseudo-code below:

**Algorithm 1** : Recursive Fibonacci Number Computation

```
function FIB1($n$)
    if $n = 0$ then
        return 0
    else if $n = 1$ then
        return 1
    else
        return FIB1($n - 1$) + FIB1($n - 2$)
```

As is evident from the pseudo-code, for values of $n \geq 2$, the function is called for $n - 1$ and $n - 2$, and the results added. As we saw earlier, runtime such an algorithm is often expressed as a recurrence relation. In this case, the recurrence relation for $T(n)$, the time taken by the algorithm on input $n$ is given by

$$T_n = \begin{cases} 1 & \text{if } n = 0 \\ 2 \text{ if } n = 1 & \\ T_{n-1} + T_{n-2} + c & \text{if } n > 2 \end{cases},$$

where $c$ is a constant representing the time taken to add two integers. By the definition of $F_n$, we have $T(n) > F_n$. We can get a lower bound on $F_n$ as: $F_n \geq 2^{\frac{n}{2}}$ for $n \geq 10$, proved by induction.

*Proof.* The base case, when $n = 10$, is trivial, since $F_10 = 57 \geq 2^{10/2} = 2^5 = 32$. Assuming this to be true for all $k \leq n - 1$, we will show that $F_n \geq 2^{\frac{n}{2}}$. We know that $F_n = F_{n-1} + F_{n-1}$, and since we assumed that the inequality holds for every $k \leq n - 1$, it holds also for $n - 1$ and $n - 2$. Hence our inequality becomes the following which completes our proof.

$$\begin{aligned} F_n =& F_{n-1} + F_{n-2} \\ & \geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} \\ & \geq 2 \cdot 2^{\frac{n-2}{2}} \\ & = 2^{\frac{n}{2}-1+1} = 2^{\frac{n}{2}}. \end{aligned}$$

□

Given that we now know that $F_n \geq 2^{\frac{n}{2}}$ and $T(n) > F_n$ we can now easily say that $T(n) = \Omega(2^{\frac{n}{2}})$. As is evident from this relation, the recursive method gives us a time bound greater that exponential for computing a Fibonacci number, which is quite unreasonable. The next section looks into how might we go about reducing this time duration to a computationally feasible time.

## 1.2 Memoization

We saw that the recursive method for calculating a Fibonacci number has a time bound greater than exponential. Key insights may be gained if we were to look into the recursion tree for a recursive execution, for say $n = 6$

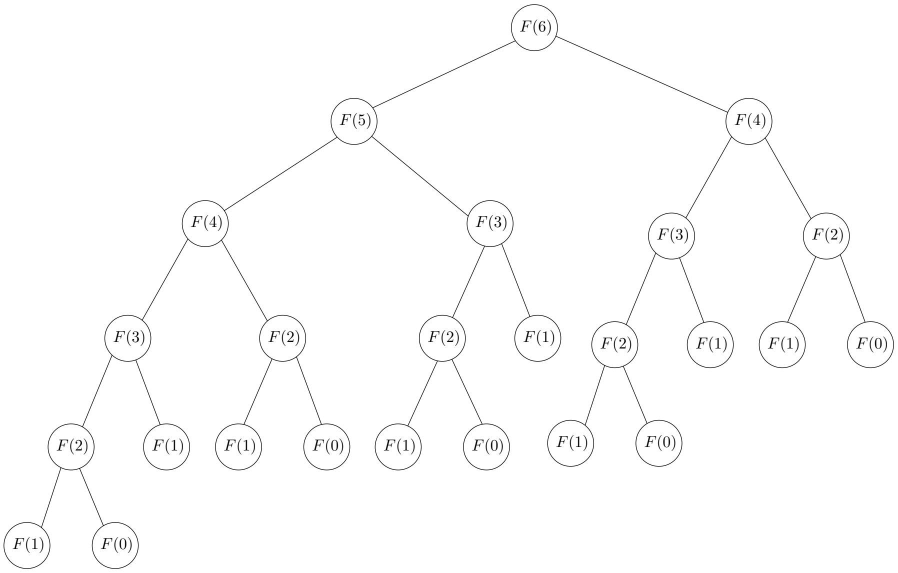


Figure 1: A recursion tree for execution of Algorithm 1 on input $n = 6$

From the figure, we can see that there are a lot of repeated calls i.e. calls to the recursive function with the same value. $F(4)$ is repeated twice, $F(3)$ 3 times, $F(2)$ 4 times and so on. It stands to reason that if we were to avoid such calls, we could significantly reduce the amount of computational steps required for our final problem. These calls to a reduced input sizes are known as *subproblems*, and by computing these subproblems once and storing its results for reuse, we are doing what is known as *memoization*.

### 1.2.1 Recursive Fibonacci with Memoization

We now present an updated version of Algorithm 1, which utilizes the concept of memoization to compute our final result. It is given as follows:

**Algorithm 2** : Recursive Fibonacci Number Computation with Memoization

```
for $i = 1$ to $n$ do                                ▷ Initially $F[i]$'s are unknown
    $F[i] \leftarrow \infty$
function FIB2($n$)
    if $n = 0$ then
        $F[0] \leftarrow 0$
        return 0
    else if $n = 1$ then
        $F[1] \leftarrow 1$
        return 1
    else
        if $F[n-1] \neq \infty$ then
            $temp1 \leftarrow F[n-1]$
        else
            $temp1 \leftarrow$ FIB2($n-1$)
            $F[n-1] \leftarrow temp1$
        if $F[n-2] \neq \infty$ then
            $temp2 \leftarrow F[n-2]$
        else
            $temp2 \leftarrow$ FIB2($n-2$)
            $F[n-2] \leftarrow temp2$
        return $temp1 + temp2$
```

We analyze this function's runtime by counting the number of recursive calls. We can see that a recursive call is made only when $F[i] = \infty$ and in each call one entry in $F$ is filled. In total, there would be $n$ calls to this function and in a given execution, there may be some further recursive calls and a constant number of comparisons, additions and array operations. Hence, the total time taken for an input of size $n$ is $O(n)$, which is substantially better than our first attempt.

## 1.3 The Bottom Up Approach

The previous problem showcased a dynamic programming approach which utilized recursion to solve subproblems, which were used to arrive at the final solution. Another approach in dynamic programming instead goes from bottom to up, computing the subproblems from the base case to the final solution. Again, we look into the $n$th Fibonacci number problem, and present an iterative method which utilizes memoization.

**Algorithm 3** : Iterative Fibonacci Number Computation with Bottom-up Approach

```
function FIB3($n$)
    for $i = 1$ to $n$ do                                  ▷ Initially $F[i]$'s are unknown
        $F[i] \leftarrow \infty$
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i = 2$ to $n$ do
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

This algorithm, again takes $O(n)$ time, and has lower overhead than the recursive method. All three algorithms discussed so far have their correctness following from the Fibonacci sequence definition.

# 2 Weighted Independent Set in Path Graphs

In any graph, an independent set of a graph $G$ is defined as a subset $S \subset V$, where $V$ is the set of vertices in $G$, such that no two elements in $S$ are adjacent to each other. As an example, consider the graph below

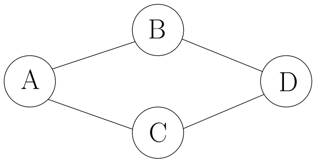
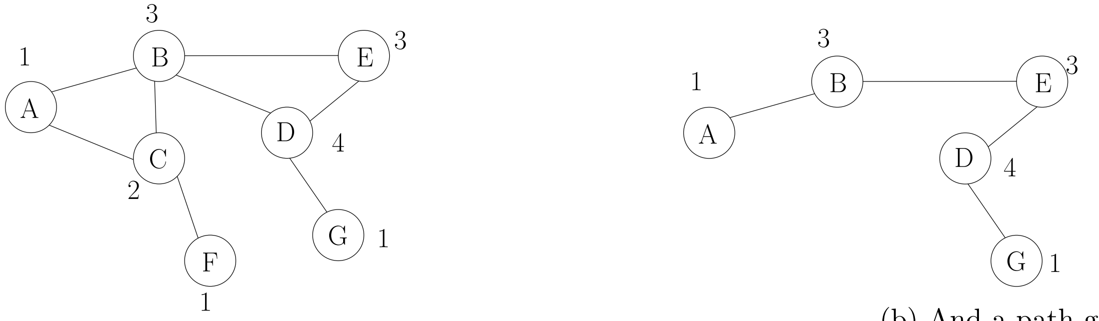


Figure 2: An example graph: The subsets $\{A\}$, $\{A, D\}$ $\{B, C\}$ are some independents set in this graph, since no pair in these subsets are adjacent.

**Problem** (The Max Independent Set Problem)**.** *Given a graph $G$ find an independent set in $G$ of maximum cardinality.*

With a small graph, finding a max independent set is quite easy, but is difficult for large graphs with complex connections. We will discuss this problem and other versions of it in quite detail when we study complexity theory.

We move on to a special case of this problem, where the input graphs are restricted to be path graphs. Note that given the structure of the input, it is very easy to see that in any path graph a max independent set is just a collection of alternating vertices in the paths order. The problem become interested when we consider node-weighted path graphs. Formally, given a graph $G = (V, E, w)$ where $V$ is the set of vertices, $E$ the set of edges and $w : V \to \mathbb{R}$ assign real weights to each vertex of $G$. For an example, consider the graphs below.



(a) An example node weighted general graph

(b) And a path graph

Figure 3: An example of node-weighted path graph

Given a node-weighted graph, weight of a subset of vertices is defined to be the sum of weights of vertices in the subset. Combining our definition of the independent set and the weighted graph, we now describe a weighted independent set, which is simply an independent set of any weighted graph. In the weighted graph above, the set $\{A, E, G\}$ is an independent set with weight $1 + 3 + 1 = 5$, and $\{A, F, D\}$, another independent set has weight $1 + 2 + 4 = 7$.

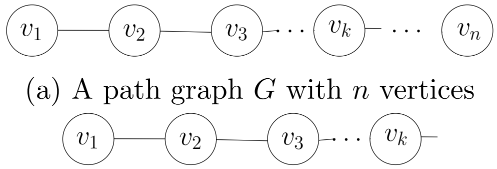
Since the weighted independent set can be defined for any weighted graph, it can also be defined for a weighted path graph, which is simply a special case of the former. In particular, we would be interested in finding out the maximal weighted independent set of a path graph, which we describe in the following problem statement:

**Problem.** *Given a path graph with positive integer weights on vertices, find an independent set of largest weight.*

We first try to solve this problem in a brute force manner, which in this case, simply involves making all possible subsets, and find the one with the largest weight. But as with all brute force solutions, we would like to improve on our runtime, so we try out a greedy approach, due to the nature of the problem. We simply do the following; take the heaviest node in the graph, add it to our set, remove it and its neighbors, and repeat with the resulting vertices, till there are no vertices left. This approach, while well meaning, has the possibility of removing vertices that could have contributed more to the total. For example, consider a path graph with weights {1,7,10,7}, in order. In accordance with the greedy approach, we would select 10, and would remove the two 7's, leaving 1. This would give us a total weight of 11, whereas we could have done better, 14 if both 7's would have been selected.

A divide-and-conquer approach could also be considered, by dividing the graph into two subgraphs, computing the maximal weighted independent sets for both subgraphs, and then combining the two. Note that combining the two would not result in a feasible solution, due to the possibility of conflicts occurring at the ends of the two subgraphs. For instance, in a graph consisting of {4,5,5,1}, dividing equally would give {4,5} and {5,1}. To avoid conflict, assuming left precedence, we would have to select 1 at the right side, whereas a better solution would be to keep 5 at the right and select 4 at the left. Another divide and conquer algorithm could be to put all odd vertices into $S_1$ and all even vertices into $S_2$. These both sets would be independent, and by taking the one that is heavier, we should have a maximal weighted independent set. But this would also not work, consider that the graph is as such, {1,2,10,2,3,10}. Dividing into even and odd sets would give us {1,10,3} and {2,2,10}, with either set being selected. A better set would have been {1,10,10}.

Let us now consider what the optimal solution would look like. We first order our vertices, and give them a number, $v_1, \dots, v_n$. We denote with $G_k$ the path graph made up by $v_1, v_2, \dots, v_k$ (the first $k$ vertices in the path).



(a) A path graph $G$ with $n$ vertices

(b) A path graph $G_k$ with $k$ vertices

Figure 4: The current setup to our problem.

## Recursive Formulation

We now define our notation for the optimal solution and value of the optimal solution for $G_k$. The optimal solution for $G_k$ is $Opt - Soln(k)$, which gives the set of vertices from $G_k$ that gives the maximum weighted independent set for $G_k$, and the value is $Opt(k)$, which is simply the sum of values of the vertices in $Opt - Soln(k)$. Our actual input is $G$ with $n$ vertices, and hence we want to compute $Opt - Soln(n)$ and $Opt(n)$. We now discuss the structure of the optimal solution without knowing how to compute it. We take inspiration from our earlier divide-and-conquer attempts, that we can compose the solution of the larger problem by using the solutions for a smaller subproblem. We can see from the structure of the problem that there are two possibilities for the $v_n$ vertex and its value. Either it is included in the optimal solution, meaning that $v_n \in Opt - Soln(n)$, or it is not. In the latter case, since $v_n \notin Opt - Soln(n)$ and $Opt - Soln(n)$ is an independent set in $G$, $Opt - Soln(n)$ is an independent set in $G_{n-1}$, Furthermore, $Opt - Soln(n)$ is the maximum weight independent set in $G_{n-1}$. This is because if there was a bigger weight ind. set in $G_{n-1}$, that set would also be independent in $G_n$, but of bigger weight, which would be a contradiction of our assumption of $Opt - Soln(n)$ being the optimal solution.
In the other similar but slightly more complicated case when $v_n \in Opt - Soln(n)$, by the property of the independent set, $v_{n-1}$ cannot be in $Opt - Soln(n)$, because having it would make our set non-independent. Mimicking our argument for the previous case, we can say that if $v_n \in Opt - Soln(n)$, then $Opt - Soln(n) \setminus \{v_n\}$ is an independent set in $G_{n-2}$. We also can say that $Opt - Soln(n) \setminus \{v_n\}$ is the optimal solution for $G_{n-2}$ using the previous case's argument, since having a ind. set in $G_{n-2}$ which has larger weight than $Opt - Soln(n) \setminus \{v_n\}$, would make it an independent set in $G_n$, but with higher weight, since $Value(Opt - Soln(n)) = Value(Opt - Soln(n) \setminus \{v_n\}) + w(v_n)$ (all weights are positive), which would contradict our assumption. For example if $Opt(n) = 100$ and $w(v_n) = 10$, Suppose there is an independent set of total weight 95 in $G_{n-2}$, then since this independent set is also an independent set in $G_n$, this combined with $v_n$ gives us an IS of weight $95 + 10 = 105$, which contradicts the fact that $Opt(n) = 100$.

With these two cases covered, we can now define the recurrence relation for our problem, which is:

$$Opt(n) = \max \begin{cases} w_n + Opt(n-2) & \text{if } v_n \in Opt - Soln(n) \\ Opt(n-1) & \text{if } v_n \notin Opt - Soln(n) \end{cases}$$

In other words we get that the max weight independent set in $G_n$ must be either

1. a max weight independent set in $G_{n-1}$
2. $v_n$ plus a max weight independent set in $G_{n-2}$.

We actually get something stronger if we know whether $v_n$ is part of MIS of $G_n$, then we recursively solve the problem in $G_{n-2}$, otherwise we recursively solve the problem in $G_{n-1}$ and be done. The problem is that we do not know which is the correct case, so we simply explore both possibilities, and take the **better** one, which should be clear. In general for any $k \geq 3$ we get that:

$$Opt(k) = \max \begin{cases} w_k + Opt(k-2) & \text{if } v_n \in Opt - Soln(k) \\ Opt(k-1) & \text{if } v_k \notin Opt - Soln(k) \end{cases}$$

We know just need to come up with the base cases for this recursion scheme. It is very easy to see that $Opt(1) = w(v_1)$, as a one vertex path has the one vertex as an independent set and it is the maximum (only) independent set since all weights are positive. Similarly it is clear that maximum weight IS in a two vertex path is the vertex with the larger weight, (both combined is not an IS, as there is an edge between the two vertices). So $Opt(2) = \max\{w(v_1), w(v_2)\}$. Now this readily gives us a recursive algorithm as follows (we only focus on finding the vale of optimal solution at this moment, i.e. given $P_n$, we want to find $Opt(n)$ at the moment).

**Algorithm 4** : Recursive Max WIS in Path Graph

```
function REC-WIS($k$)
    if $k = 1$ then
        return $w(v_1)$
    else if $k = 2$ then
        return $\max\{w(v_1), w(v_2)\}$
    else
        return $\max\{$REC-WIS$(k-1), w(v_k)+$REC-WIS$(k-2)\}$
```

This is clearly a recursive formulation of the brute force method, and will give an exponential $2^n$ runtime. But we can explore this idea further to improve upon it. Note that this a top down method, which typically gives us exponential time due to the number of repeated calls to smaller subproblems in separate subtrees. An example for a 6 vertex path graph is given in Figure 5

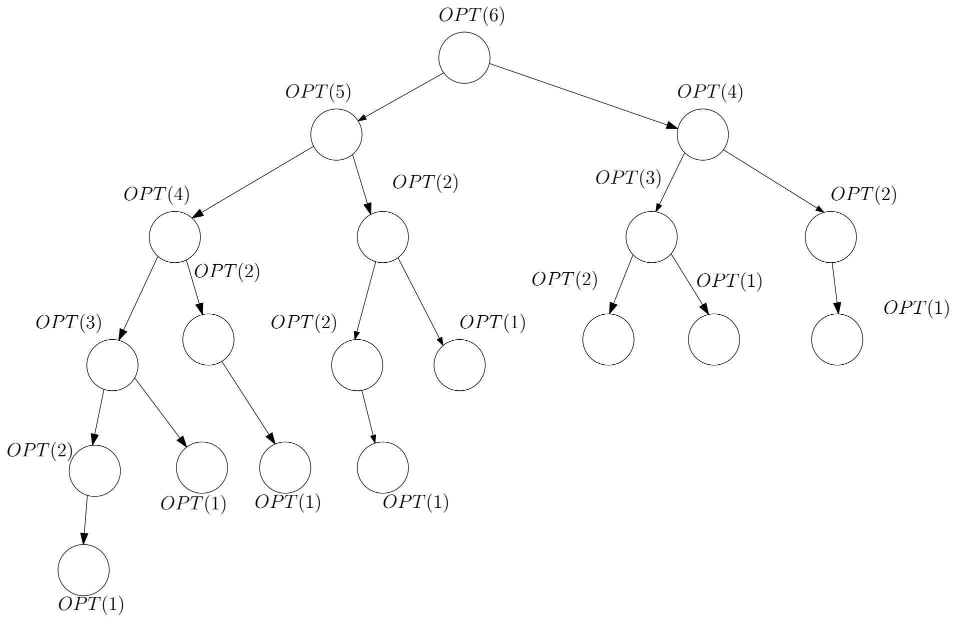


Figure 5: Recursion tree for computing a 6-vertex path graph's maximum weight independent set

We have discussed previously that the dynamic programming paradigm allows us that, once the solution to the problem has been expressed in terms of solutions to some subproblems, and the repetition of subproblems has also been identified, we can use either memoization, or a bottom up approach. We use an array of size $n$ to hold the value of the maximum weight independent for $G_i$ at the $i$th index. The following code shows an implementation that utilizes memoization to compute the MWIS solution:

**Algorithm 5** : Recursive WIS with Memoization

```
for $i = 1$ to $n$ do                                ▷ Initially $A[i]$'s are unknown or NULL
    $A[i] \leftarrow \infty$
function REC-WIS-MEMO($k$)
    if $k = 1$ then
        $A[1] \leftarrow w(v_1)$
        return $A[1]$
    else if $k = 2$ then
        $A[2] \leftarrow \max\{w(v_1), w(v_2)\}$
        return $A[2]$
    else
        if $A[k-1] \neq \infty$ then
            $temp1 \leftarrow A[k-1]$
        else
            $temp1 \leftarrow$ REC-WIS-MEMO($k-1$)
            $A[k-1] \leftarrow temp1$
        if $A[k-2] \neq \infty$ then
            $temp2 \leftarrow A[k-2]$
        else
            $temp2 \leftarrow$ REC-WIS-MEMO($k-2$)
            $A[k-2] \leftarrow temp2$
        return $\max\{temp1, w(v_k) + temp2$
```

This function's runtime can be analyzed from the number of recursive calls. We can see that a recursive call is made only when $A[i] = \infty$ and in each call to this function one entry in the $A$ array gets filled. So in total there are $n$ calls to this function and in a given execution of this function, there may be some further recursive calls and a constant number of comparisons and array operations, hence total time is $O(n)$ much better than the first recursive function.
The next function is a bottom up approach on the problem, and is as follows:

**Algorithm 6** : Max WIS in Path Graph with Bottom-up Approach

```
function WIS-BOTT-UP($n$)
    for $i = 1$ to $n$ do                            ▷ Initially $A[i]$'s are unknown
        $A[i] \leftarrow \infty$
    $A[1] \leftarrow w(v_1)$
    $A[2] \leftarrow \max\{w(v_1), w(v_2)\}$
    for $i = 3$ to $n$ do
        $A[i] \leftarrow \max\{A[i-1], w(v_i) + A[i-2]\}$
    return $A[n]$
```

This one clearly takes $O(n)$ time and has much less overload.

### Reconstruction Algorithm: Finding the Optimal Solution

Given the array filling algorithm above, which only computes the value of the max weight IS in path graph, (which only returns some number e.g. 110). It should be clear that we can augment the array $A$ to not only find the value but also the solution. It would need something like a linked list augmented to each $A[i]$ containing the Max WIS in each $G_i$ (what we denoted by $Opt - Soln(i)$). Correctness of this approach can be easily proved by the above recurrences, but this solution is very wasteful, it wastes a lot of space. Instead what we do and is usually done in the dynamic programming paradigm, is to back-track through the filled in array (the memo), to reconstruct the optimal solution. Here is the main idea that we exploit; $v_k$ belongs to Max WIS of $G_i$ if and only if $(w(v_i) + Opt(k-2)) \geq Opt(k-1)$, i.e. which branch did we take while filling in the array. Its proof follows from the above recurrence relation. Note that when there is a tie than we can go either way (this corresponds to multiple optimal solutions). Following is the reconstruction algorithm

**Algorithm 7** : Backtracking to find a Max WIS in $P_n$

```
function FIND-WIS($n$)
    $S \leftarrow \emptyset$
    $i = n$
    while $i \geq 1$ do
        if $w(v_i) + A[i-2] \geq A[i-1]$ then
            $S \leftarrow S \cup \{v_i\}$
            $i \leftarrow i - 1$
        else
            $i \leftarrow i - 1$
    return $S$
```

## 3 Weighted Interval Scheduling

One problem that comes up in many forms is resource management and task completion. Many jobs in the current world involve allocation of resources to particular tasks, which have a particular time to start at and have a duration of time before completion. These tasks also have certain awards associated with them, which makes it difficult to determine which tasks are to be assigned resources. As an example, consider a single server, which is capable of processing tasks given to it by users. There are a number of requests made by the users for processing time on the server, and each task needs to start at a particular time specific to the task, and has a duration for which the task will run. Each task also has a value which measures the benefit of completing that task.
We now define the problem in more formal terms.

**Problem.** *There are n requests, each request $r_i$ has $s(i)$ and $f(i)$, which are the start and finish times respectively. Implicitly, we have $d(i) = f(i) - s(i)$, which defines the duration of the ith task. In addition, we have $v(i)$, the value of the ith task. A request $r_i$ is* ***compatible*** *with $r_j$ if there is no time overlap i.e. [$s(i) < f(i) < s(j) < f(j)$] or [$s(j) < f(j) < s(i) < f(i)$]. Our goal is to select* *a subset of compatible requests that have the largest total value, the sum of all the intervals in the subset.*

Recall that the old interval scheduling problem, with a single resource, is simply a special case of this problem, where all intervals have value $v(i) = 1$. Hence, greedy strategies that did not work with that problem will not work here, such as taking the earliest start time, latest finish time, shortest interval, largest interval and least conflicting requests. Also, the greedy strategy that worked for the unweighed interval scheduling problem, which took the earliest finish time, will not work here, since it is possible to have requests which finish early, but do not have a high value. This is highlighted in Figure 6.

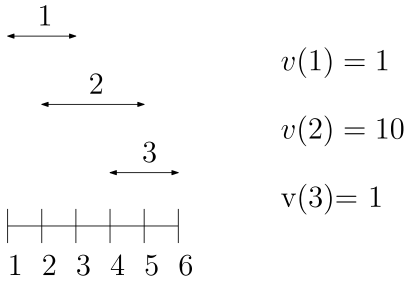


Figure 6: A counter-example for early finish time strategy in weighted interval scheduling

With the intervals given in Figure 6, the earliest finish time strategy would first pick $r_1$, which conflicts with $r_2$, but is compatible with $r_3$, which would be selected next. The total value of this subset is 2, while the optimal solution clearly is to select only $r_2$, which gives a total value of 10. It can be clearly seen that any natural greedy strategy that one might employ would not work, since a counter example can be shown for it.

Lets suppose that requests are sorted by their finishing time i.e. $f(1) \leq f(2) \leq \ldots \leq f(n)$. For request $j$ (with starting time $s(j)$ and $f(j)$), it is the $j$th request in the sorted order), we define $p(j)$ to be the largest index $i < j$, such that $r_i$ is compatible with $r_j$, i.e. for $r_j$, $p(j)$ is the first interval that ends before $r_j$ begins (going from right to left). An example is shown in Figure 7

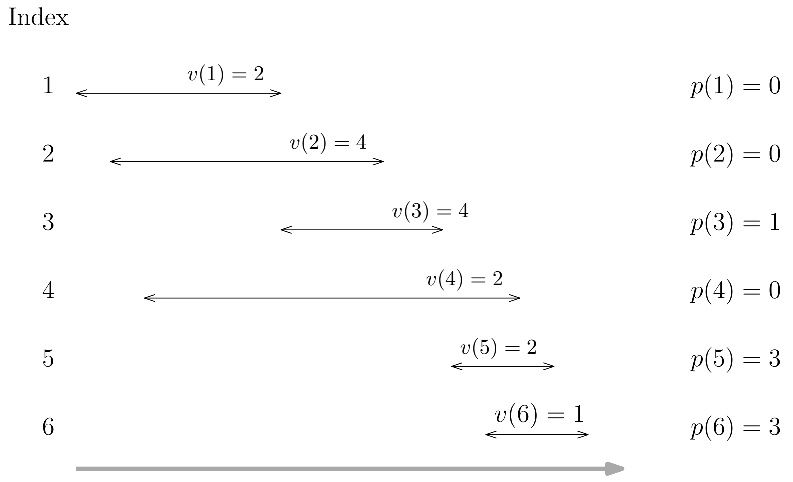


Figure 7: Intervals sorted by finishing time and with $p(i)$ assigned

Lets assume that we are given an optimal solution $\mathcal{O}$ of a weighted interval scheduling instance. We currently have no idea how to compute $\mathcal{O}$ or another equal value solution, but we can say for sure that either $r_n \in \mathcal{O}$ or $r_n \notin \mathcal{O}$. Furthermore, we know for sure that if $r_n \in \mathcal{O}$, then all intervals at the indices between $p(n)$ and $n$ cannot be part of $\mathcal{O}$, since by definition of $p(n)$, $p(n)+1, p(n)+2, \ldots, n-1$ end after $s(n)$, so they are not compatible with $r_n$. So if $r_n \in \mathcal{O}$, then we can surely exclude all requests $p(n)+1, p(n)+2, \ldots, n-1$ from consideration. Furthermore, this $\mathcal{O}$ in addition to $r_n$ must make up an optimal solution to the subproblem $r_1, r_2, \ldots, r_{p(n)}$. Because suppose optimal solution to $\{r_1, r_2, \ldots, r_{p(n)}\}$ is $\mathcal{O}^-$ and the total value of $\mathcal{O} \cap \{r_1, r_2, \ldots, r_{p(n)}\}$ has value less than the value of $\mathcal{O}^-$, then we can take $\mathcal{O}^- \cup r_n$ which is strictly greater than $\mathcal{O}$ and it is a feasible solution as all requests in $\{r_1, r_2, \ldots, r_{p(n)}\}$ are compatible with $r_n$. On the other hand, if $r_n \notin \mathcal{O}$, then the optimal solution to $\{r_1, \ldots, r_{n-1}\}$ is an optimal solution to $\{r_1, \ldots, r_{n-1}\}$.

So we have described an optimal solution to the problem $\{r_1, \ldots, r_{n-1}, r_n\}$ in terms of the optimal solution to the subproblem $\{r_1, \ldots, r_{n-1}\}$ and the other subproblem $\{r_1, r_2, \ldots, r_{p(n)}\}$, since one of these two possibilities must occur, we have :

$$OPT(r_1, \ldots, r_n) = \max\{v_n + OPT(r_1, \ldots, r_{p(n)}), OPT(r_1, \ldots, r_{n-1})\}$$

It is easy to see that $OPT(r_1, \ldots, r_j) = \max\{v_j + OPT(r_1, \ldots, r_{p(j)}), OPT(r_1, \ldots, r_{j-1})\}$. We now have a recursive formulation of the optimization problem, and so we need a base case, which is $OPT(\emptyset) = 0$. The complete recurrence relation is given by:

$$OPT(r_1, \ldots, r_j) = \begin{cases} \max\{v_j + OPT(r_1, \ldots, r_{p(j)}), OPT(r_1, \ldots, r_{j-1})\} & \text{if } j \geq 1 \\ 0 & \text{if } j = 1 \end{cases}$$

We can decide whether $r_j$ is part of the optimal solution, based on whichever or not $v_j + OPT(r_1, \ldots, r_{p(j)}) \geq OPT(r_1, \ldots, r_{j-1})$. If Yes, then $r_j$ is part of the solution, as its value is more than if we didn't include it and vice-versa. This gives us a complete recursive algorithm, to find value of an optimal solution. We can also find an optimal solution by noting down which side of the recursion tree we took. Here is the algorithm to find the value:

**Algorithm 8** : Recursive Weighted interval scheduling

```
sort requests by finishing time                ▷ it takes $O(n \log n)$ for $n$ requests
For each $r_j$ find $p(j)$                     ▷ with binary search this take $O(n \log n)$ time.
function COMPUTE-OPT-VALUE($n$)
    if $n = 0$ then
        return 0
    else
        $temp1 \leftarrow v_n$+COMPUTE-OPT-VALUE($p(n)$)
        $temp2 \leftarrow$COMPUTE-OPT-VALUE($n - 1$)
        if $temp1 \geq temp2$ then
            return $temp1$
        else
            return $temp2$
```

It's correctness directly follows by induction, using the above argument. Its runtime is exponential, due to its recursive nature. As an example, we take the set of intervals defined in Figure 7, and show its recursion tree in Figure 8

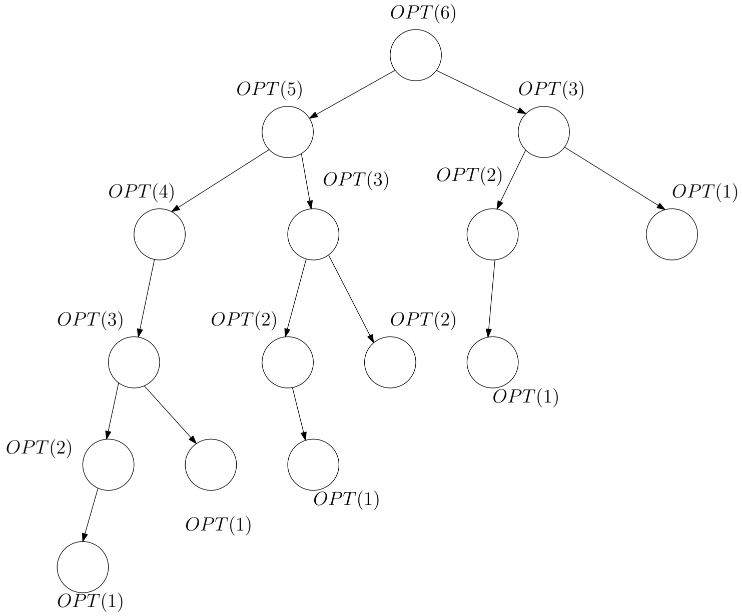


Figure 8: The tree of subproblems called by Compute-Opt on the problem instance of Figure 7

Again, this runtime is easily reducible due to the fact that we are using values of subproblems to compute the final answer, and hence can use memoization to compute the solution bottom-up, and hence reduce the runtime to linear. It is left as an exercise to the reader to devise the pseudo-code for this.

# 4 Knapsack Problem

Many world problems involve the problem of allocating space to a set of items, which have a certain value attached to them, as well as an amount of space they occupy. For example, logistical problems involve the transportation of freight in tankers, trucks and airplanes, which have a finite amount of space available, and the items in the freight have certain monetary value attached to them, as well as weight and/or space requirements. It is of considerable interest that the transporter maximize the profit they can earn, while staying within the limits of the space available. Such problems are commonly known as **Knapsack** problems, named so after the problem of a burglar having to fit items into his/her knapsack to maximize the total benefit to the burglar. Before we define the Knapsack problem in formal terms, we first look at a similar, but simpler problem.

**Problem.** *Given a set $U = \{a_1, a_2, \ldots, a_n\}$ and a positive integer $C$, find a subset $S \subset U$ such that $|S| \leq C$ and $\sum_{a_i \in S} a_i$ is as large as possible.*

This problem is known as the subset sum problem, where we want to find a subset of maximum sum, with a cardinality constraint. Many optimization problems can be modeled as such, for example $U$ could represent requests of the interval scheduling problem, and a server is available for $C$ units of time. We would like to select a subset of requests so that the server can be utilized as much as possible, while staying within the capacity constraints.
A more generalized version of this problem is as follows:

**Problem.** *There is a set $U$ =$\{a_1, \ldots, a_n\}$ of $n$ items, where each item has a weight $\{w_1, \ldots, w_n\}$, and a value $\{v_1, \ldots, v_n\}$. Given a positive integer $C$, find a subset $S \subset U$, such that $\sum_{a_i \in S} w_i \leq C$ and $\sum_{a_i \in S} v_i$ is as large as possible.*

As is evident, the previous problem is a special case of this problem, where $w_i = v_i = a_i$. This problem allows us to model problems such as weighted interval scheduling, where every interval has a duration and some value. We want to use a single server with upto $C$ units of time such that the total value of scheduled intervals is as large as possible. And as discussed before, it allows us to also model the Knapsack Problem. As has been evident from our previous discussions, greedy approaches to these problems are not successful. Consider $C = 100$ and $U = \{51, 50, 50\}$; a greedy approach could be to take the highest value item as long as the total does not go over capacity. This approach would select $\{51\}$ and stop, while clearly a better answer would be to take $\{50, 50\}$. Now consider $C = 100$ and $U = \{1, 50, 50\}$; another greedy approach would be to take the lowest weight items, but in this case, it would select $\{1, 50\}$, whereas a better option was to take $\{50, 50\}$.

A key step in solving the problems we have seen so far has been to try to express the problem in terms of a small number of subproblems, such that each subproblem can be solved easily from smaller subproblems. For the generalized problem, we will denote a problem instance in terms of three sets and an integer, which are $U = \{a_1, \ldots, a_n\}$, $W = \{w_1, \ldots, w_n\}$ and $V = \{v_1, \ldots, v_n\}$ and $C$, where $U$ is the set of items, $W$ is the set of their respective weights, $V$ the respective values and $C$ is our available capacity for the problem. We now define the optimal solution and its value.
Let $OPT(i)$ and $\mathcal{O}(i)$ be the value of the optimal solution and the optimal solution (the actual subset) respectively, of the (sub)problem $U = \{a_1, \ldots, a_i\}$, $W = \{w_1, \ldots, w_i\}$ and $V = \{v_1, \ldots, v_i\}$.

Clearly, our goal is to find $OPT(n)$ and $\mathcal{O}(n)$. We know that either $a_n \in \mathcal{O}(n)$ or $a_n \notin \mathcal{O}(n)$. In the later case we must have that $OPT(n) = OPT(n-1)$ and $\mathcal{O}(n) = \mathcal{O}(n-1)$. In the former case, we can not really consider $OPT(n-1)$ and $\mathcal{O}(n-1)$, we are cheating a little bit. Suppose we know that $a_n \in \mathcal{O}(n)$, then we have a little less capacity left to select remaining items from $U = \{a_1, \dots, a_{n-1}\}$. Notice that definition of $OPT(n)$ and $\mathcal{O}(n)$ depends on $n$ only (implicitly it depends on the sets $W$, $V$ and $U$ ($n$ elements in each)). But there is no mention of $C$, which is also part of the input and is an important constraint for the solution. Suppose $C$ was not there (or unlimited), then we can clearly take all of $U$ (or at least all element $a_i$ of $U$ such that $v_i \geq 0$, in case negative values are there). So we correct our definition as follows.

Let $OPT(i,c)$ and $\mathcal{O}(i,c)$ be the value of the optimal solution and the optimal solution (the actual subset) respectively, of the (sub)problem $U = \{a_1, \dots, a_i\}$, $W = \{w_1, \dots, w_i\}$ and $V = \{v_1, \dots, v_i\}$ and capacity $C$. Our goal is to find $OPT(n,C)$ (and $\mathcal{O}(n,c)$). Now we can completely define $OPT(n,C)$ in each cases when $a_n \in \mathcal{O}(n,C)$ and $a_n \notin \mathcal{O}(n,C)$. In the latter case clearly, $OPT(n,C) = OPT(n-1,C)$ and $\mathcal{O}(n,C) = \mathcal{O}(n-1,C)$, while in the former case, since $a_n \in \mathcal{O}(n,C)$, the remaining elements of $\mathcal{O}(n,C)$ are elements of $U \setminus \{a_n\} = \{a_1, \dots, a_{n-1}$ with weights $W \setminus \{w_n\} = \{w_1, \dots, w_{n-1}\}$ and values $V \setminus \{v_n\} = \{v_1, \dots, v_{n-1}\}$. But the total weight of elements in $\mathcal{O}(n,C) \setminus \{a_n\}$ must not exceed $C - w_n$. Hence in case $a_n \in \mathcal{O}(n,C)$, then $OPT(n,C) = OPT(n-1, C-w_n) + v_n$ and $\mathcal{O}(n,C) = \mathcal{O}(n-1, C-w_n) \cup \{a_n\}$. The recurrence relation we get now is as follows:

$$OPT(n,C) = \begin{cases} OPT(n-1, C-w_n) + v_n & \text{if } a_n \in \mathcal{O}(n,C) \\ OPT(n-1,C) & \text{if } a_n \notin \mathcal{O}(n,C) \end{cases}$$

It is easy to write the base case, when there is only one element in $U = \{a_1\}$, if $w_1 < c$, then $OPT(1,c) = v_1$, else $OPT(1,c) = 0$ and $\mathcal{O}(1,c) = \{a_1\}$ else $\mathcal{O}(1,c) = \emptyset$ (respectively). This suggests that we solve many subproblems; we need to know the value of the best solution using a subset of $\{a_1, \dots, a_{n-1}\}$ and capacity $C - w_n$ and for the else part with capacity $C$. So we will solve even more subproblems, we will determine the value of $OPT(i,c)$ for each $1 \leq i \leq n$ and for each $1 \leq c \leq C$. It is easy to see that if we have solutions to all of these subproblems then it is trivial to determine that value of $OPT(n,C)$, namely, $OPT(n,C) = \max\{OPT(n-1, C-w_n) + v_n, OPT(n-1,C)\}$, and in general, $OPT(i,c) = \max\{OPT(i-1, c-w_i) + v_i, OPT(i-1,c)\}$. The above recursive formulation readily gives us a recursive algorithm, but if we visualize the recursion tree, we can see a lot of redundancies. For example if $w_9 = 5, w_{10} = 5$ and $C = 100$, then we will have to solve $OPT(8,95)$ once for when in the branch when $a_{10}$ is included and $a_9$ is not included and once in the branch where the $a_9$ is included and $a_{10}$ is not included in the optimal solution. So as usual in the dynamic programming paradigm, we compute each value only once using a bottom up iteration, and memoize it. Since our problem is now two dimensional i.e. has two variables to consider as $OPT(i,c)$, our memoization will be in a 2-dimensional array. Following is the pseudocode

**Algorithm 9** : Bottom-uP iterative Computation of $OPT(i, c)$ using the above recursive formulation

```
function KNAPSACK($n, C$)
    for $i = 0$ to $n$ do                                  ▷ Initially $OPT[i][c]$'s are unknown
        for $c = 0$ to $C$ do
            $OPT[i][c] \leftarrow -\infty$
    for $c = 0$ to $C$ do
        $OPT[0][c] \leftarrow 0$                 ▷ when $i = 0 \implies U = \emptyset$, then clearly $OPT[0][.] = 0$
    for $i = 0$ to $n$ do
        $OPT[i][0] \leftarrow 0$            ▷ when $c = 0 \implies$ no capacity, then clearly $OPT[.][0] = 0$
    for $i = 1$ to $n$ do
        for $c = 0$ to $C$ do
            if $OPT[i-1][c-w_i] + v_i \geq OPT[i-1][c]$ and $c \geq w_i$ then
                $OPT[i][c] \leftarrow OPT[i-1][c-w_i] + v_i$
            else
                $OPT[i][c] \leftarrow OPT[i-1][c]\}$
    return $OPT[n][C]$
```

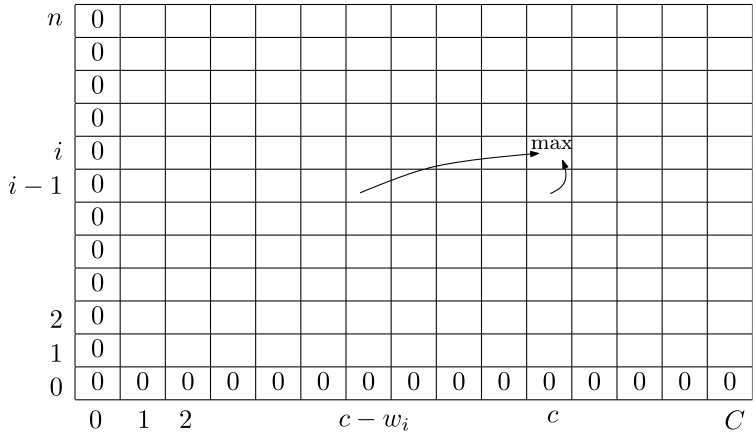


Figure 9: The matrix of optimal value of solution to the knapsack instance for all value of $i$ and $c$. $OPT[i][c]$ is computed from the two cells with arrow to the cell $OPT[i][c]$

## A sample run of the Knapsack problem

We now look at a sample problem to see how our proposed solution works. We take the problem instance where $U = \{a_1, a_2, a_3, a_4, a_5\}$, $W = \{1, 6, 5, 2, 8\}$, $V = \{3, 2, 8, 1, 5\}$ and $C = 15$. We set up our solution space as in the initial part of the algorithm(before we go beyond $OPT[0][.]$ or $OPT[.][0]$) as in Figure 10

| U | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 4 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 3 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 2 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 1 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | C |

$$W = \{1, 6, 5, 2, 8\}$$

$$V = \{3, 2, 8, 1, 5\}$$

Figure 10: The initial solution space

We begin by filling the row $OPT[1][.]$, which corresponds to the set $U = \{a_1\}$ and its respective derivative sets for $W$ and $V$. Since this includes only one item, with weight 1, we would take $a_1$ when the capacity goes to $w_1$, which is 1 for this problem. Subsequent capacity increases would not matter, so all cells onwards $OPT[1][1]$ would get the same value. The resultant problem space is shown in Figure 11.

| U | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 4 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 3 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 2 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 1 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | C |

$$W = \{1, 6, 5, 2, 8\}$$

$$V = \{3, 2, 8, 1, 5\}$$

Figure 11: After iteration for $OPT[1][.]$

Our second iteration now considers row $OPT[2][.]$, which now has two objects in set $U = \{a_1, a_2\}$. Recall that our condition to add $a_i$ to our solution requires that we have enough capacity, which is clearly not possible until $c \geq w_2$, which is 6. So for $OPT[2][1]$ to $OPT[2][5]$ we would put in the corresponding values in the previous row. When $c \geq 6$, we now consider whether to put $a_2$ in our solution. At $c = 6$, putting in $a_2$ would not leave any more capacity, so we will have $OPT[1][0] + v_2 = 2$ versus $OPT[1][6] = 3$. Since the latter is greater, we would put that value in $OPT[2][6]$. For subsequent values, we now have capacity that can also support $a_1$, so we take the sum of $v_2$ and the previous row's corresponding value. The end of this iteration results in the state shown in Figure 12

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $U$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ |  |
| 4 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ |  |
| 3 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ |  |
| 2 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |  |
| 1 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $C$ |

$$W = \{1, 6, 5, 2, 8\}$$

$$V = \{3, 2, 8, 1, 5\}$$

Figure 12: After iteration for $OPT[2][.]$

Our third iteration now considers row $OPT[3][.]$, which has set $U = \{a_1, a_2, a_3\}$. Again, till we reach capacity for $a_3$, we take the values of the previous row for the corresponding columns. At $c = 5$, we now can consider whether to put $a_3$ into or not. Our next best choice is $OPT[2][5]$, which equals 3. Since $v_3 = 8$, we include $a_3$, which leaves no space for other items, hence $OPT[3][5] = 8$. From $c = 6$ to $c = 10$, we do not have space for $a_2$ if we include $a_3$, but can consider $a_1$, so we have either $OPT[2][c - 5] + v_3 = 11$ or $OPT[2][c] = 3or5(c = 7$ onwards). At $c = 11$, if we include $a_3$, we still have space to consider $a_2$, which makes our choice either $OPT[2][6] + v_3 = 11$ or $OPT[2][11] = 5$. Since the former case is higher, we include $a_3$ in our answer, but $a_2$ still did not make the cut. When we move on to $c \geq 11$, we now have space for all three items, which makes our total 13. The state after this iteration is as in Figure 13

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $U$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ |  |
| 4 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ |  |
| 3 | 0 | 3 | 3 | 3 | 3 | 8 | 11 | 11 | 11 | 11 | 11 | 13 | 13 | 13 | 13 | 13 |  |
| 2 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |  |
| 1 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $C$ |

$$W = \{1, 6, 5, 2, 8\}$$

$$V = \{3, 2, 8, 1, 5\}$$

Figure 13: After iteration for $OPT[3][.]$

For the last two iterations, we simply show the results after the iterations in Figure 14 and 15, and leave it as an exercise for the reader to determine how the values changed.

| U | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 0 | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | -∞ | |
| 4 | 0 | 3 | 3 | 4 | 4 | 8 | 11 | 11 | 12 | 12 | 12 | 13 | 13 | 13 | 13 | 13 | |
| 3 | 0 | 3 | 3 | 3 | 3 | 8 | 11 | 11 | 11 | 11 | 11 | 13 | 13 | 13 | 13 | 13 | |
| 2 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | |
| 1 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | C |

$W = \{1, 6, 5, 2, 8\}$

$V = \{3, 2, 8, 1, 5\}$

Figure 14: After iteration for $OPT[4][.]$

| U | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 0 | 3 | 3 | 4 | 4 | 8 | 11 | 11 | 12 | 12 | 12 | 13 | 13 | 13 | 16 | 16 | |
| 4 | 0 | 3 | 3 | 4 | 4 | 8 | 11 | 11 | 12 | 12 | 12 | 13 | 13 | 13 | 13 | 13 | |
| 3 | 0 | 3 | 3 | 3 | 3 | 8 | 11 | 11 | 11 | 11 | 11 | 13 | 13 | 13 | 13 | 13 | |
| 2 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | |
| 1 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | C |

$W = \{1, 6, 5, 2, 8\}$

$V = \{3, 2, 8, 1, 5\}$

Figure 15: After iteration for $OPT[5][.]$

At the end, we return the value at $OPT[5][15]$, which is 16.

## Backtracking to find the actual solution

We know that the value of the optimal solution is in $OPT[n][C]$ the last cell of the matrix, But this doesn't give us the solution ($\mathcal{O}(n, C)$). We can also maintain a solution with each cell which would be a straightforward application of the above recurrence. But now while filling a cell $OPT[i][c]$ takes $O(1)$ time, to write a set with each cell will take $O(n)$ time. But now while filling a cell $OPT[i][c]$ takes $O(1)$ time, to write a set with each cell will take $O(n)$ time, so total runtime will be blown up by a factor of $O(n)$, as well as a lot of wastage of space. $\mathcal{O}(n, C)$ can be computed easily by backtracking for the cell $OPT[n][C]$. Recall the recurrence for the problem:

$$OPT(i, c) = \begin{cases} OPT(i-1, c-w_i) + v_i & \text{if } a_i \in \mathcal{O}(i, c) \\ OPT(i-1, c) & \text{if } a_i \notin \mathcal{O}(i, c) \end{cases}$$

From this we know that $a_i \in \mathcal{O}(i, c)$ iff $OPT[i-1][c-w_i] + v_i \geq OPT[i-1][c]$. For instance $a_n \in \mathcal{O}(n, C)$ iff $OPT[n-1][C-w_n] + w_n \geq OPT[n-1][C]$, so we just compare again $OPT[n-1][C-w_n] + w_n$ with $OPT[n-1][C]$ (both cells are already computed) and if the former is big we conclude that $a_n$ is part of the optimal solution ($\mathcal{O}(n, C)$), otherwise it is not. We use this observation to trace back the actual solution. A pseudocode for general $a_i$ is given as follows

**Algorithm 10** : Backtracking to find $\mathcal{O}(n, c)$ using the above recursive formulation

```
function FIND-SET($i$,$c$)
    if $i = 0$ or $c = 0$ then
        return $\emptyset$
    else
        if $OPT[i-1][c-w_i] + v_i \geq OPT[i-1][c]$ then
            return $a_i \cup$ FIND-SET($i-1$,$c-w_i$)
        else
            return FIND-SET($i-1$,$c$)
```

We call this function $Find - Set(n, C)$, it will return us the subset with optimal value. Since this function calls itself each time with strictly smaller value of $i$, its runtime is clearly $O(n)$.

### Runtime

Clearly when the table is filled in we know the value of the optimal solution from the cell $OPT[n][C]$ and we can find the solution by backtracking as mentioned above. A given cell of the matrix is filled in $O(1)$ from the two cells that are already filled in. So the runtime is proportional to the size of the matrix, which is $O(nC)$. Note that this runtime is not in terms of the size of input. Here $n$ is a parameter about the size of input the number of elements in the set or the size of the three arrays. But $C$ is actually the value of an input (not it's size). size of $C$ is like $O(\log C)$ bits. So strictly speaking this algorithm is not a polynomial time algorithm, and such algorithms are called pseudo polynomial time algorithms.

## 5 Sequence Alignment

A key problem that occurs when analyzing strings is the problem of sequence alignment, where strings are compared to each other to find sequences that are similar for a pair of strings. This problem comes up not for electronic dictionaries and spell-checking tools, but also for molecular biologists, who are interested in the similarities between DNA sequences between two organisms. We will use this problem of finding similarities between DNA sequences to illustrate the dynamic programming paradigm applied to sequence alignment.

A DNA sequence is represented as a sequence of characters, whose alphabet is $\{A, C, G, T\}$, which stand as the molecules that are present in the sequence. A DNA sequence encodes information for the construction of proteins that the particular organism uses for survival. Scientists use these sequences to determine the similarities between organisms who may be related evolutionarily. Before we move further, we define the notion of *similarity* as it applies to between two strings. Consider two strings $X$ and $Y$, where $X$ consists of the sequence of symbols $x_1x_2\ldots x_m$ and $Y$ consists of the sequence of symbols $y_1y_2\ldots y_n$. Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ as representing the different positions in the strings $X$ and $Y$, and consider a matching in these sets. A matching is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching $M$ of these sets is an *alignment* if there are no "crossing" pairs i.e. if $(i, j), (i', j') \in M$ and $i < i'$, then $j < j'$. In essence, an alignment gives a way of lining up two strings, by telling us which pairs of positions will be lined up with one another. For example, for the strings {'stop','tops'}, the alignment $\{(2, 1), (3, 2), (4, 3)\}$ aligns the subsequence 'top' in both strings.

We will define similarity based on finding the *optimal* alignment between $X$ and $Y$, as per the following criteria. Suppose $M$ is a given alignment between $X$ and $Y$. The following conditions are considered:

- A parameter $\delta$ 0, which defines a *gap penalty*. For each position of $X$ and $Y$ that is not matched in $M$ i.e. a gap, incurs a cost of $\delta$.

- Next, for each pair of letters $p, q$ in our alphabet, there is a *mismatch cost* of $\alpha_{pq}$ for aligning $p$ with $q$. Thus for each $(i, j) \in M$, the appropriate mismatch cost of $\alpha_{x_i y_j}$ for lining up $x_i$ with $y_j$. We generally assume $\alpha_{pp} = 0$ for each letter $p$ i.e. there is no mismatch cost to line up a letter with a copy of itself. This is not necessary.

- Finally, the *cost* of $M$ is the sum of its gap and mismatch costs, and we seek an alignment with minimal cost.

The process of finding such an alignment is known as *sequence alignment*. Generally, the choice of $\delta$ and $\alpha_{pq}$ affects which alignment is considered optimal, but in our case, we will consider these parameters a given. with our numerical definition of similarity between strings $X$ and $Y$, we will now try to design our algorithm.

In our previous approaches to problems using dynamic programing, we try first to define how the optimal solution would look like. We shall do the same for this problem. Lets begin by considering the last positions for both strings $m$ and $n$ in the optimal alignment $M$. There are two cases; either $(m, n) \in M$ or $(m, n) \notin M$ i.e. either the last symbols match to each other, or they don't. This by itself is not sufficient to give us a solution, so we augment it with the following lemma.

**Lemma 1.** *Let $M$ be any alignment of $X$ and $Y$. If $(m, n) \notin M$, then either the $m^{th}$ position in $X$ or the $n^{th}$ position in $Y$ is not matched in $M$*

*Proof.* We prove by contradiction. Consider that $(m, n) \notin M$, and there are numbers $i < m$ and $j < n$ such that $(m, j) \in M$ and $(i, n) \in M$. This contradicts our definition of an alignment: we have $(i, n)(m, j) \in M$ with $i < m$, but $n > i$, so the pairs $(i, n)$ and $(m, j)$ cross. □

Using this lemma, we now define 3 cases which can occur in our optimal alignment $M$, which also help us in formulating our recurrence. These are:

1. Either $(m, n) \in M$; or

2. the $m^{th}$ position of $X$ is not matched; or

3. the $n^{th}$ position of $Y$ is not matched.

Let $OPT(i, j)$ denote the minimum cost of an alignment between $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$. If case 1 holds, we pay $\alpha_{x_my_n}$ and then align $x_1x_2 \dots x_{m-1}$ against $y_1y_2 \dots y_{n-1}$; we get $OPT(m, n) = a_{x_my_n} + OPT(m-1, n-1)$. If case 2 holds, we pay a gap cost of $\delta$ since the $m^{th}$ position of $X$ is not matched, and then we align $x_1x_2 \dots x_{m-1}$ against $y_1y_2 \dots y_n$, for which we get $OPT(m, n) = \delta + OPT(m-1, n)$. Similarly for case 3, we get $OPT(m, n) = \delta + OPT(m-1, n)$.

Using the same argument for the subproblem of finding the minimum-cost alignment between $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$, we have the following lemma:

**Lemma 2.** *The minimum alignment costs satisfy the following recurrence for* $i \geq 1$ *and* $j \geq 1$*:*

$$OPT(i, j) = min[\alpha_{x_iy_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)]$$

*Moreover,* $(i, j)$ *is in an optimal alignment* $M$ *for this subproblem if and only if the minimum is achieved by the first of these values.*

With this recurrence defined, we can know devise our algorithm for determining the value of the optimal alignment easily. The pseudocode for doing so is as follows:

**Algorithm 11** : Sequence alignment using dynamic programming

```
function SEQALIGNMENT(X,Y)                                          ▷ $X, Y$ are our strings
    $OPT \leftarrow Array[m][n]$
    for i = 0 to m do
        $OPT[i][0] \leftarrow i\delta$
    for j = 0 to n do
        $OPT[0][j] \leftarrow j\delta$
    for j = 1 to n do
        for i = 1 to m do
            if X[i] == Y[j] then
                $OPT[i][j] = min[\alpha_{x[i]y[j]} + OPT[i-1][j-1], \delta + OPT[i-1][j], \delta + OPT[i][j-1]]$
            else
                $OPT[i][j] = min[\delta + OPT[i-1][j], \delta + OPT[i][j-1]]$
    return $OPT[m][n]$
```

Since this algorithm only gives us the value of the optimum solution, we would need to determine the actual alignment The second fact of the lemma allows us to do so by backtracking though $OPT$, starting at $OPT[m][n]$. Our recurrence shows us how the value would have been set, with case 1 being the case when $(i, j)$ were a valid pair in the optimal alignment. Hence, whenever we see that $OPT[i][j] = \alpha_{x[i]y[j]} + OPT[i-1][j-1]$, we include $(i, j)$ to our alignment, and move to $OPT[i-1][j-1]$. Else, we would either go to $OPT[i-1][j]$ or $OPT[i][j-1]$ depending on which would be the smaller.

# 6 Bellman Ford(Single Source Shortest Paths)

Recall that we discussed the problem of the single source shortest path in a weighted digraph when we discussed the Dijkstra algorithm. In particular, we presented a $O(|E| \log |V|)$ algorithm, called the Dijkstra algorithm that was able to calculate shortest paths for any pair of vertices for a weighted digraph $G = (V, E, W)$, where all weights were positive. We also discussed why Dijkstra was unable to work in the presence of negative edge weights, and any strategies to make Dijkstra work on such graphs, such as reweighting to remove negative edge weights, were unsuccessful. We repeat those arguments here for the ease of the reader.

We consider the following graph, and try to run Dijkstra over it:

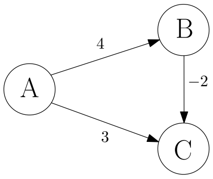


Figure 16: An example graph with negative weights

Dijkstra with source $A$ will add first $C$ and then $B$ to $R$ and selects the path $AC$ and $AB$ as shortest paths from $A$ to $B$ and $C$ respectively. Clearly the shortest path from $A$ to $C$ is the path $ABC$ with total weight 2. One way to deal with negative weight edges would be to multiply each weight with $-1$. This clearly doesn't solve the problem as the already positive weight edges will be become negative now. another way to deal with negative weight edges is to shift all the weights to the positive side, i.e. to add a large constant $C$ to all the weights. If the constant $C \geq \max_{e \in E} |w(e)|$, then clearly all the new weights will be now non-negative. Does this solve the problem? Consider the following graph in Figure 17

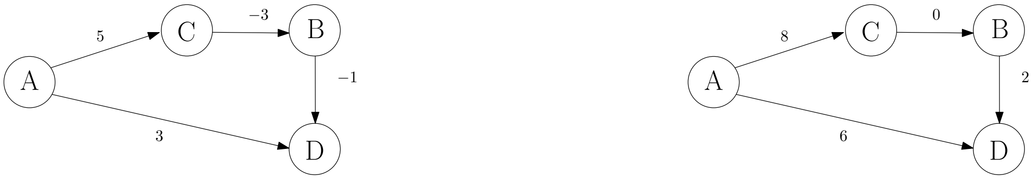


Figure 17: In the graph on the left, $d(A, D) = 1$ via the path $ACBD$. The largest absolute value edge weight is 3. We add 3 to all the weights. In the resulting graph, on the right, $AD$ is the shortest path from $A$ to $D$ with length 6.

A key question comes to mind at this point, which is why is it necessary to be able to take negative weight edges into account. This is necessary since there are real world scenarios that may involve negative weights, for instance, if we describe a graph of business transactions where selling results in positive revenue, and buying results in costs i.e. negative revenue. We may want to have paths, which would be particular series of transactions, with the least change in total revenue. With this in mind, we now look into an algorithm that helps us to find all shortest paths in a digraph with arbitrary edge weights. But before we do this, we first need some prerequisite conditions to our problem, which allow us to actually propose a solution.

## Negative cycles

Any digraph can have a number of cycles, which are simply paths that return to the source vertex. However, in a digraph that has negative edge weights can have what is known as a *negative cycle*, which is simply a cycle whose sum of edge weights is negative. Such cycles can be problematic to deal with, since their presence causes problems when computing shortest paths. In principle, if there exists any path for a pair of vertices $v, t$ in a graph that contains a negative cycle in the path, then there exists no shortest path from $v$ to $t$. This is because one can build a path of arbitrary negative weight by simply looping through the cycle as many times as desired, i.e. for any claim of a shortest path between $v$ and $t$, one can show a more shorter path by looping though the negative cycle.

## Problem definition

As is evident from above, we would preferably not have a negative cycle in our weighted digraph. With this condition in place, we now present a lemma that allows us to prove that there is a shortest path available from $v$ to $t$.

**Lemma 3.** *Consider a weighted digraph $G = (V, E, W)$. If $G$ does not have any negative cycles, then there exists a cheapest path from $v$ to $t$, both $\in V$ that is simple, and has at most $|V| - 1$ edges.*

*Proof.* Consider a cheapest $v \to t$ path that uses at least $|V|$ edges. This would mean that there exists a cycle in the path, since with at least $|V|$ edges, you would be visiting at least $|V| + 1$ vertices, which would mean that at least 1 vertex is visited more than once. If we were to remove this cycle, i.e. remove the paths that loop back to any particular vertex, we would get a path that has at most $|V| - 1$ edges; having any more would indicate that a cycle is present. Also, since there are no negative cycles, removing the cycles would in fact result in a path that is cheaper than the original $\geq |V|$ edge path. Since that path was supposed to be the cheapest path, the new path would in fact be the most cheapest $v \to t$ path. □

We now define our problem in formal terms and state what our end result should be:

**Problem.** *Given a weighted directed graph $G = (V, E, W)$, where $V$ and $E$ are our vertex and edge sets respectively and $W$ is the set of weights for the edges, which can be negative, and a source vertex $s \in V$, we would like to compute the shortest paths from $s$ to all other vertices $v \in V$. If this cannot be done, then output a negative cycle.*

We will leave the output of a negative cycle for later, and for now focus on computing the shortest paths from $s$ to all vertices $v \in V$, assuming that there is no negative cycle. As with all dynamic programming problems, we will first focus on looking at the optimal solution first.

Generally, graphs do not have a sequential nature that we can use. However, what we do know that our optimal solution would be some order of vertices and edges, and our subproblems would also have a ordered solution. By intuition, we can say that for any shortest path between two vertices $s$ and $t$, all subpaths in this path would also be shortest for their respective source and destination. Given that in general a path can have any number of edges, we allow ourselves to add an extra parameter to our optimal solution, which is the number of edges that it can have. We look at the following example to illustrate this:

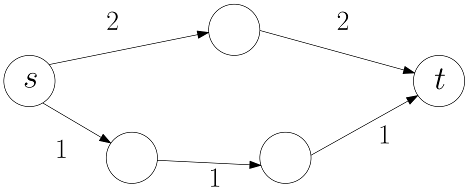


Figure 18: An example directed digraph

In Figure 18, consider that we are looking for shortest paths from $s$ to $t$. With our additional constraint of limiting the number of edges that the path can have, we can now pose questions like 'What is the shortest path from $s$ to $t$ that has $k$ or less edges?'. For this example, lets look at the shortest paths from $s$ to $t$ that have 2 or less edges. The only such path is the top path, with a path length of 4 (the sum of its edge weights). The bottom path has 3 edges, and as such is not considered in this situation. But if we were to increase our edge limit to 3, we can now choose between the two paths, the one with the shorter path length being selected. The bottom path in this case, has length 3, and as such is shorter than the upper path, and hence is selected as the shortest path with 3 edges or less. This little example illustrates an important point, which is that when we increase the number of edges permitted, we increase the size of our problem. This point gives us an opportunity to define the subproblems in our problem. We start with the lemma below:

**Lemma 4.** *Let $G = (V, E, W)$ be a directed weighted graph and a source vertex $s \in V$. $G$ might or might not have a negative cycle. For every $v \in V$, $i \leftarrow \{1, 2, 3, \ldots\}$, where $i$ is the maximum number of edges allowed, let $P$ be the optimal solution, i.e. for all paths that have at most $i$ edges and are from $s$ to $v$, $P$ is the shortest path. Cycles are permitted in this path, since there are a finite number of edges that can be used by the path. We have two possible cases in this scenario:*

1. *Either $P$ has $\leq (i-1)$ edges, i.e. it is a shortest $s \to v$ path with $\leq (i-1)$ edges, or*
2. *$P$ has $i$ edges with last hop $(x, v)$, and the rest of the path $P'$, then $P'$ is the shortest $s \to x$ path with at most $(i-1)$ edges*

*Proof.* The first case of our lemma is proven by an obvious contradiction, and is trivial. For our second case, let there be a path $Q$ that is shorter than $P'$, in that $Q$ is a path from $s$ to $x$, has $\leq (i-1)$ edges. If so, then $Q + (x, v)$, which is from $s$ to $v$, has $\leq i$ edges, has shorter path length than $P' + (x, v)$. This is obviously a contradiction, since it contradicts the optimality of $P$. Hence proved. □

With our optimal solution and its subproblems defined, we now move to define the recurrence for our problem, which will allow us to devise the required algorithm. We start as follows:

Let $OPT(i,v)$ be the minimum length of a $s \rightarrow v$ path with $\leq i$ edges. We define $OPT(i,v)$ to be $\infty$ if there are no paths from $s$ to $v$ that take at most $i$ edges. This would be true for most destination when $i$ is small. For every $v \in V, i \leftarrow \{1,2,3,\ldots\}$, we have the following recurrence.

$$OPT(i,v) = min \begin{cases} OPT(i-1,v) \\ min_{(x,v)\in E} \left\{ OPT(i-1,x) + w(x,v) \right. \end{cases}$$

The second case simply takes the minimum of all $s \rightarrow x$ paths where $x$ is one edge away from $v$ and has $v$ in its out-neighborhood plus the weight of the $(x,v)$ edge. The correctness of this recurrence follows from the optimal substructure lemma that we defined earlier.

We said earlier that we will not consider that the graph has any negative cycles, and we now explain why we did so. Suppose input graph $G$ has no negative cycles. This means that all shortest paths in such a graph will not have cycles, since all cycles would have positive weight and removing such cycles from the paths only decreases the length. As such, these shortest paths would have at most $(|V|-1)$ edges. This point gives us a upper limit on $i$ which was our edge limit, which, in absence of negative cycles, only needs to go up to $(|V|-1)$, since moving $i$ further makes no sense in this case. So, we now have a limit on the amount of subproblems to solve, which is to compute $OPT(i,v)$ for all $i \leftarrow \{0,1,2,3,\ldots,|V|-1\}$, and for all $v \in V$.

We can now write out the pseudocode for the Bellman-Ford algorithm. It is as follows:

**Algorithm 12** : Bellman-Ford algorithm

```
function BellmanFord(G,s)                                  ▷ $G = (V,E,W)$ and $s$ is source vertex
    $OPT \leftarrow [|V|-1][|V|]$                                     ▷ Two-dimensional array
    $OPT[0][s] = 0$
    for $v \in V \setminus s$ do
        $OPT[0][v] = \infty$
    for $i = 1,2,\ldots,(|V|-1)$ do
        for $v \in V$ do
            $OPT[i][v] = min[A[i-1][v], min_{(x,v)\in E}[A[i-1][x] + w(x,v)]]$
    return $OPT$                                 ▷ Final answers at $OPT[|V|-1][v]$ for all $v \in V$
```

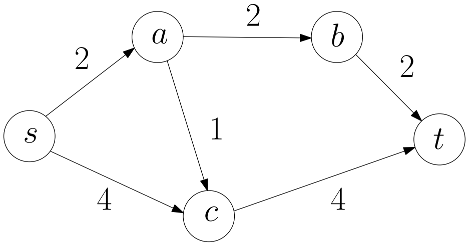
## Running example on pseudocode

We now look at an example run of the Bellman-Ford algorithm to understand how it works, as shown in Figure 19. $s$ is the source vertex. For this graph, since there are 5 vertices, $i$ would take the value of at most 4, and will iterate over the range $\{0,1,2,3,4\}$. We set up the array as in Figure **??**, with $i$ indices as the rows and $v$ indices as the columns.

For $i = 1$, we now compute $OPT[1][v]$ for all $v \in V$. $OPT[1][s]$ gets 0, and since only $a$ and $c$ are accessible from $s$ ,there values get updated. Since $OPT[0][a]$ and $OPT[0][c]$ are $\infty$, any positive weight would be better, so we will update the values for $a$ and $c$ with the weights of $(s, a)$ and $(s, c)$ respectively. The rest of the vertices, being inaccessible in one hop from $s$ , will get $\infty$.

For $i = 2$, we compute $OPT[2][v]$. $s$ gets 0, $a$ gets its previous value, since there is no 2-hop path that is shorter than the current 1 hop path. But $c$ now has two choices, either it retains the old value $OPT[1][c]$ which equals 4, or it updates based on the 2 hop path $\{s, a, c\}$, which has weight $OPT[1][a] + w(a, c) = 2 + 1 = 3$. Since the latter is lower, we would select that value. $b$ is now reachable, and so gets $OPT[1][a] + w(a, b) = 4$, and $t$ gets $OPT[1][a] + w(a, t) = 8$. Note that we did not use $a$'s updated value, but the value from the previous iteration.

We now only show the resultant arrays for the subsequent iterations, and leave it as an exercise for the reader to determine the updates that were made.



(a) The initial solution space

| $i$ | $s$ | $a$ | $b$ | $c$ | $t$ | $V$ |
|---|---|---|---|---|---|---|
| 4 | | | | | | |
| 3 | | | | | | |
| 2 | | | | | | |
| 1 | | | | | | |
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | |

(b) Iteration $i = 1$

| $i$ | $s$ | $a$ | $b$ | $c$ | $t$ | $V$ |
|---|---|---|---|---|---|---|
| 4 | | | | | | |
| 3 | | | | | | |
| 2 | | | | | | |
| 1 | 0 | 2 | $\infty$ | 4 | $\infty$ | |
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | |

(c) Iteration $i = 1$

| $i$ | $s$ | $a$ | $b$ | $c$ | $t$ | $V$ |
|---|---|---|---|---|---|---|
| 4 | | | | | | |
| 3 | | | | | | |
| 2 | 0 | 2 | 4 | 3 | 8 | |
| 1 | 0 | 2 | $\infty$ | 4 | $\infty$ | |
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | |

(d) Iteration $i = 2$

| $i$ | $s$ | $a$ | $b$ | $c$ | $t$ | $V$ |
|---|---|---|---|---|---|---|
| 4 | | | | | | |
| 3 | 0 | 2 | 4 | 3 | 6 | |
| 2 | 0 | 2 | 4 | 3 | 8 | |
| 1 | 0 | 2 | $\infty$ | 4 | $\infty$ | |
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | |

(e) Iteration $i = 3$

| $i$ | $s$ | $a$ | $b$ | $c$ | $t$ | $V$ |
|---|---|---|---|---|---|---|
| 4 | 0 | 2 | 4 | 3 | 6 | |
| 3 | 0 | 2 | 4 | 3 | 6 | |
| 2 | 0 | 2 | 4 | 3 | 8 | |
| 1 | 0 | 2 | $\infty$ | 4 | $\infty$ | |
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | |

(f) Iteration $i = 4$

Figure 19: Example Run of Bellman Ford

## Runtime

The runtime for the Bellman-Ford algorithm would at first look to be $O(n^2)$, but it is not so, but in fact the runtime comes out to be $O(|E||V|)$. This is because the total work is simply $O(|V| \times \sum_{v \in V} in - deg(v))$, since we are evaluating for each vertex, and are looking at most $in - deg(v)$ nodes when computing the optimal solutions in each iteration. Since $\sum_{v \in V} in - deg(v) = |E|$, we get $O(|V||E|)$.

## Detecting negative cycles

Recall that we had put the possibilities of there being negative cycles in our graph back in order to describe our algorithm. We now bring that possibility back in, in order to address it. Ideally, we would want that the algorithm reports back the fact that $G$ contains a negative cycle if it encounters one. In order to do so, we make a claim: by running the algorithm for one more iteration, we will be able to determine whether the graph contains a negative cycle or not. We state the claim more formally as follows:

**Claim 5.** *$G$ does not contain a negative cycle that is reachable from $s$ if, when the Bellman-Ford is run for one more iteration of $i$ i.e. at $i = |V|$, there is no change in the shortest path lengths for any vertex $v \in V$ i.e. $OPT[|V|][v] = OPT[|V| - 1][v]$ for all $v \in V$.*

As a consequence of this claim, when we run the algorithm for $|V|$ iterations and see that there is no change in the last two rows, we know that there is no negative cycle, and return the values back as the shortest path lengths. But, if there were any changes, and those changes could and would only be reductions in the value of the shortest path length, we know that there is a negative cycle that is reachable from $s$, and as consequence, we return that there is a negative cycle. But before we accept this claim, we need to prove it, and it is as follows.

*Proof.* We first look at the first part of the claim that if $G$ does not contain a negative cycle, there is no change in the values in the new iteration. This claim has been proved before by the very first lemma that defined in this section. As for the second part of the c claim, which states that if the values of $OPT$ do not change from the $|V| - 1$th to the $|V|$th iteration, then $G$ does not contain a negative cycle. Assume that the former case is true, and let $d(v)$ denote the common value of $OPT[|V| - 1][v]$ and $OPT[|V|][v]$. Recall the recurrence by which this value is calculated. We can rewrite this recurrence in terms of $d(v)$ as follows:

$$d(V) = min \begin{cases} d(v) \\ min_{(x,v) \in E} \begin{cases} d(x) + w(x, v) \end{cases} \end{cases}$$

Since $d(v)$ was chosen instead of $d(x) + w(x, v)$ we can say that the following inequality holds:

$$d(v) \leq d(x) + w(x, v). \text{ for all edges } (x, v) \in E$$

Or:

$$d(v) - d(x) \leq w(x, v). \text{ for all edges } (x, v) \in E$$

Now consider an arbitrary cycle $C$. If the previous inequality holds, then we can say the following:

$$\sum_{(x,v)\in C} w(x,v) \geq \sum_{(x,v)\in C} (d(w) - d(v))$$

All directed cycles have the property that each vertex in the cycle appears only once as the start of some arc(an order of edges), and only once as the end of an arc, so each d-value appears once as a positive value, and once as a negative value. This results in a total result of 0. □