# Contents

# 4 Dynamic Programming

*Dynamic Programming is not dynamic, nor is it about programming.*

— Professor David Kempe

**Dynamic Programming (DP)** is a useful computational technique that involves breaking complex problems into simpler sub-problems (*does this remind you of something we just learned?*), with the additional ability of storing and reusing past solutions, which turns out to be an extremely powerful optimization in certain problems. When using DP, you secretly ask yourself the following question:

$$\boxed{\text{In order to do } \textit{this}\text{, what do I need to do between the previous step and now?}}$$

## 4.1 Memoization

DP is closely related to recursion, as well as exhaustive search and backtracking. The main difference is that **it avoids unnecessary re-computation.**

Consider, for example, the famous recursion problem of computing Fibonacci numbers. We define `Fib(n)` to be such that it returns $1$ if $n = 0$ or $1$, and returns `Fib(n-1) + Fib(n-2)` otherwise. Runtime-wise, we have

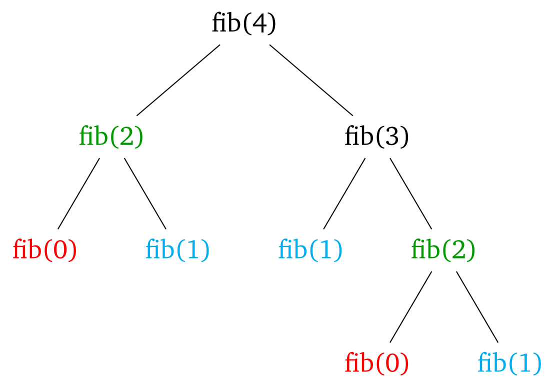
$$T(n) = T(n-1) + T(n-2) + \mathcal{O}(1) \qquad T(0) = T(1) = \mathcal{O}(1).$$

Dropping the $\mathcal{O}(1)$ work, we see $T(n) \geqslant T(n-1) + T(n-2)$, which is the Fibonacci recurrent itself. You can use an easy induction to prove that

$$\texttt{Fib(n+1)} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

which, as $n$ gets larger, approximates $(1/\sqrt{5})(((1+\sqrt{5})/2))^n \approx 1.618^n/\sqrt{5}$, since the second term vanishes because $|(1-\sqrt{5})/2| \approx 0.618 < 1$. From this, we see that **computing Fibonacci using recursion takes exponential time $\mathcal{O}(1.618^n)$**, which is... certainly really bad.

**What's the problem? Too many re-computations**. Suppose you call `Fib(4)`. What happens? Well, we see a lot of repetitions, even for an argument as small as $4$ — the lower levels in the computation tree consist of many repetitions, and they increase exponentially as the depth increases.



How do we fix this? We maintain an array of size $n+1$ ($0$-indexed, from $0$ to $n$) when computing `Fib(n)`, and we replace recursive calls with direct access to corresponding indices in the array. This technique is called **memoization**.

Obviously, the new runtime is $\mathcal{O}(n)$. So much better.

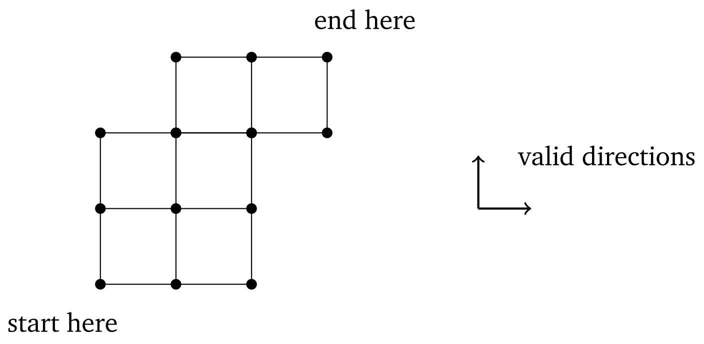
**Algorithm 8:** Fibonacci with DP

```
Function Fibonacci($n$):
    allocate integer array F[$n+1$], zero-indexed
    F[0] = F[1] = 1
    for $i = 2, 3, \cdots, n$ do
        F[$i$] = F[$i-1$] + F[$i-2$] // only array lookups, no recursive calls
    return F[$n$]
```

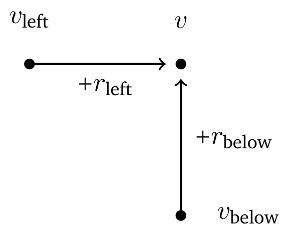
## 4.2 (Optional) Intuition Behind DP

"In order to do *this*, what do I need to do between the previous step and now?"

Let us consider a very simple problem. You have a grid, where each edge has a certain amount of reward associated with it. You start from the bottom-left vertex, and your goal is to move to the top-right vertex. You can only move up or right, not down or left. How do you maximize your reward, among all valid paths?



Well, how do we arrive at a vertex? We either walked upward or rightward to reach it. That is, we were at either the vertex to its left, or the vertex below it, at the previous step.



Let $f(\cdot)$ denote the optimal reward function. If we somehow already know the optimal reward obtainable at the two predecessor vertices, i.e. $f(v_{\text{left}})$ and $f(v_{\text{below}})$, then it becomes clear that one of them eventually plays a role in $f(v)$. What is the maximum possible reward at $v$, given that the penultimate vertex we visit is $v_{\text{left}}$? Well, the last edge has reward $r_{\text{left}}$, so the answer is $f(v_{\text{left}}) + r_{\text{left}}$. Here, we implicitly used an extremely important insight of DP: **any optimal solution to the current problem must contain optimal solutions to the sub-problems involved**. Similarly, if we arrive at $v$ from $v_{\text{below}}$, the maximum possible reward is $f(v_{\text{below}}) + r_{\text{below}}$. Since we want to maximize the reward, it naturally follows that

$$f(v) = \max\{f(v_{\text{left}}) + r_{\text{left}}, f(v_{\text{below}}) + r_{\text{below}}\}.$$

Of course, if $v$ only has one valid processor (for example, if it's on the bottom row so it can only be reached from left, not below) then one of the $\max$ terms vanish. The above equation is called the DP formula of the question. With it, starting from the bottom-left vertex, we populate a two-dimensional lookup table/array storing the values of $f$ at each vertex one at a time, all the way until we reach the top-right vertex. (One viable order is by completing the first column, then the second column, and so on.)

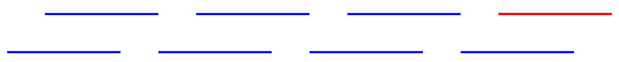
## 4.3 Weighted Interval Selection

Now we revisit the (unweighted) interval selection problem and assign weights to each interval:

> Given $n$ intervals with starting points $s(i)$, finishing points $f(i)$, and weights $w(i)$, select a set of them without overlap that achieves maximum total weight.

Motivation: instead of visiting as many events as possible, prioritize those that we consider more important.

First note that our previous "earliest finish time" greedy algorithm fails: if we have two tasks, one from $t = 1$ to $t = 100$ with weight $100$, the other from $t = 2$ to $t = 3$ with weight $1$, "earliest finish time" picks the latter.
In fact, we claim that no myopic greedy algorithm works, since in this example, short-term decisions in fact depend on long-term knowledge, something greedy algorithms do not have access to. Consider the following example:



weights: blue = 2

If the red interval has weight $1$, then the optimal solution would be selecting the bottom $4$ intervals, but if the red interval has instead weight $3$, then it is clearly more optimal to select the top $4$. There is no way for any greedy algorithm to figure out which interval to pick first!

### A DP Approach to Weighted Interval Selection

For DP, **we often analyze relatively easy** (sometimes even trivial, as we see in this example) **properties of the optimum solution, and exploit them to derive an efficient algorithm**.
In this example, what kind of trivial/apparent property can we abuse? Given a set $S$ of intervals, define $\text{OPT}(S)$ to be the total weight in the optimal solution. The key observation is that **for any interval $i \in S$, either $i$ is selected by the optimal solution or not**. No third possibility. We can derive the following based on this trivial claim:

- If $i$ is not picked by the optimal solution, then the optimal solution is identical to that of $S \setminus \{i\}$, since we wouldn't pick $i$ anyways. In other words, $\text{OPT}(S) = \text{OPT}(S\setminus\{i\})$.
- On the other hand, if $i$ is picked by the optimal solution, then the optimal solution for $S$ consists of $i$, plus the optimal solution of $S\setminus\{\text{every interval intersecting } i\}$ — apparently, if $i$ were to be chosen, anything else intersecting it cannot. In other words, $\text{OPT}(S) = w(i) + \text{OPT}(S\setminus\{\text{everything intersecting } i\})$.

### Implementing Solution in Polynomial Time

We can implement this naively using exhaustive search with backtracking, obtaining a runtime of $\mathcal{O}(2^n)$ since a set of size $n$ has $2^n$ subsets. Even with a memoization table, we still need to store info related to each subset, so still $2^n$. How do we improve?

Now it's a good time to compare our weighted interval selection against the previous lattice graph example:

- In both examples, we find two potential "previous steps" that lead to the optimal solution of the current step.
- In the lattice graph example, the previous steps are: (i) taking a step up from the vertex below, and (ii) taking a step to the right from the vertex on the left.

- In the weighted interval example, we do have two previous steps, but it is not immediately clear how to find them, in particular computing $S\backslash\{\text{everything intersecting } i\}$, unlike in the lattice graph example, where we know immediately what the two predecessor vertices are. **The notion of a clear "direction" is missing here**.

Fortunately, we do have the power to pick the interval number $i$ that we branch over and hence obtain a sense of direction. A good choice is again, the finish time, and we show by doing so we reduce runtime to $\mathcal{O}(n \log n)$ (in fact, the DP itself takes only linear time, making the sorting of finishing time the bottleneck).

We now assume the following:

- All intervals are sorted by non-decreasing finish times, so $f(i) \leqslant f(j)$ if $i < j$.
- We have an additional function $t(\cdot)$, such that $t(k)$ is the last interval $j$ that finishes before before $k$ starts. In other words, this is the last interval, index-wise, that is before $k$ and does not overlap with $k$. Define $t(k) = 0$ if all previous intervals overlap with $k$.

$$\begin{tikzcd} \mathrm{OPT}(k-1) \arrow[r, "+0"'] & \mathrm{OPT}(k) \\ & \mathrm{OPT}(t(k)) \arrow[u, "+w(i)"] \end{tikzcd}$$

With these extra assumptions, we can vastly simplify the recurrence relation. **Instead of computing the optimal solution w.r.t. a set, we compute it w.r.t. this sorted index**. Now, given a set $S$ of sorted intervals $\{1, 2, \cdots, k\}$, the last interval $k$ is either selected by $\mathrm{OPT}(k)$ or not. And there are precisely two paths to reach it:

- $k$ is not in $\mathrm{OPT}(k)$, In this case $\mathrm{OPT}(k) = \mathrm{OPT}(k-1)$ since they correspond to the same set of intervals.
- $k$ is in $\mathrm{OPT}(k)$. This means interval $k$ is selected, incurring an additional weight $w(i)$. What is the previous step if we chose this path? Well, all other intervals must have indices $\leqslant t(k)$, since they cannot overlap with $k$. And of course, we want to pick as much weight as possible from the first $t(k)$ intervals, and we know this quantity is $\mathrm{OPT}(t(k))$. Hence in this path $\mathrm{OPT}(k) = w(i) + \mathrm{OPT}(t(k))$.

Now we have all the ingredients to cook up the polynomial-time solution.

**Algorithm 9:** Weighted Interval Selection via DP

```
Inputs: $n$ intervals, with starting points $s(i)$, finishing points $f(i)$, and weights $w(i)$
------------------------------------------------------------
sort the intervals by non-decreasing finishing time
pre-compute all $t(\cdot)$ values
allocate array $\mathrm{A}[n+1]$, zero-indexed
$\mathrm{A}[0] \leftarrow 0$
for $\underline{i = 1, 2, \cdots, n}$ do
    $\mathrm{A}[i] \leftarrow \max(\mathrm{A}[i-1], w(i) + \mathrm{A}[t(i)])$
Return: $\mathrm{A}[n]$
```

Analyzing its runtime is straightforward — $\mathcal{O}(n \log n)$ for sorting, $\mathcal{O}(n \log n)$ for computing $t(\cdot)$ using $n$ binary searches ($\mathcal{O}(n)$ also possible), and $\mathcal{O}(n)$ for the DP loop. Total runtime: $\mathcal{O}(n \log n)$.

***Correctness proof.*** To prove $\mathrm{A}[n] = \mathrm{OPT}(n)$ we use strong induction and prove this holds for all $0 \leqslant i \leqslant n$. The base case is clear, since $\mathrm{OPT}(0) = 0$ as there is nothing to select, and we initialized $\mathrm{A}[0]$ to $0$.

Using strong induction, we see[25]

$$\begin{aligned} A[i] &= \max(A[i-1], w(i) + A[t(i)]) && \text{by line 7} \\ &= \max(\mathrm{OPT}(i-1), w(i) + \mathrm{OPT}(t(i))) && \text{by strong IH and } t(i) < i \\ &= \mathrm{OPT}(i) && \text{by recurrence relation.} \end{aligned}$$

□

## 4.4 Edit Distance / String Alignment

In this section we consider the following problem:

> Given two strings $x[1:n]$ and $y[1:m]$ (one-indexed, lengths $n, m$ respectively), how similar are them?

Here we choose edit distance as our metric for similarity. It cares only about single-character edits (insertions, deletions, and substitutions). Clearly, this is closely related to applications like spell checkers.
An alternate way to view these three operations is string alignment: for example, to align words *"occurrence"* and *"ocurrannce"*, we align the two words vertically, adding hyphens to both words whenever we want to make them the same length.

```
o - c u r r a n n c e
o c c u r r e - n c e
```

It follows that to transform *"ocurrannce"* into *"occurrence"*, one way is to insert a $c$, replace an "a" with "e", and remove an redundant "n." Insertion and deletion correspond to having a blank in one of the words, and replacement corresponds to having different characters at the same index.

In our course, **we will assume that insertion and deletion both cost $A$, and overwriting/replacement costs $B$.** And we will use the alignment version when constructing our DP solution. We can easily generalize this to more complicated scenarios, e.g. more specific costs, or non-constant costs even for the same operation.

### A DP Approach to String Alignment

In order to construct the DP formula for this question, we'd like to know the last position of the optimal alignment of $x[1:n]$ and $y[1:m]$. Since we only had three operations, there are three (four, to be precise) cases:

(1) *An alignment (correct or incorrect, hence two sub-cases) of $x[n]$ and $y[m]$.* In this case, $x[1:n-1]$ and $y[1:m-1]$ will be aligned with each other. To obtain optimal solution of aligning $x[1:n]$ and $y[1:m]$, the subproblem must have also been optimized, so $x[1:n-1]$ and $y[1:m-1]$ must have been aligned optimally.

- If $x[n]$ and $y[m]$ are aligned correctly (i.e. same character), no additional cost compared to the optimal solution of aligning $x[1:n-1]$ and $y[1:m-1]$.
- Otherwise, an additional cost $B$ of replacement will be incurred compared to the optimal subsolution.

(2) *An alignment of $x[n]$ with a blank.* In this case, $x[1:n-1]$ and $y[1:m]$ must have been aligned optimally. Compared to this optimal subsolution, we have an additional cost of $A$.

[25] A lot of DP correctness proofs look like this, feeling as if we were just stating the obvious. Usually a three-liner: (i) state what the algorithm does, (ii) using IH to replace the table entry into OPT, and (iii) use recurrence relation.

(3) *An alignment of a blank with $y[m]$.* Symmetric to the previous case. $x[1:n]$ and $y[1:m-1]$ must have been aligned correctly. Additional cost $A$ again.

We now translate above into formulas, denoting the optimal solution of aligning $x[1:i]$ and $y[1:j]$ by $\mathrm{OPT}(i,j)$:

$$\mathrm{OPT}(0,0)=0 \qquad \mathrm{OPT}(i,0)=iA \qquad \mathrm{OPT}(0,j)=jA \qquad \text{(base cases)}$$

$$\begin{aligned}\mathrm{OPT}(i,j)=\min(&\mathrm{OPT}(i-1,j-1)+B\cdot\mathbf{1}[x[i]\neq y[j]]^{\ddagger},\\ &\mathrm{OPT}(i-1,j)+A,\\ &\mathrm{OPT}(i,j-1)+A).\end{aligned} \qquad \text{(recurrence relation)}$$

Note that the base cases also include the situations where we align a non-empty string with an empty one (length 0), in which case the only operation we should perform is deletion. This recurrence relation might look daunting overall, but implementing it isn't. All we need is one two-dimensional array.

**Algorithm 10:** Edit Distance / String Alignment via DP

```
1  Inputs: one-indexed strings $x[1:n], y[1:m]$, last-index inclusive
   ----------------------------------------------------------------
2  allocate table $A[0:n][0:m]$, zero-indexed, last-index inclusive
3  for $i = 0, 1, \cdots, n$ do $A[i][0] \leftarrow iA$
4  for $j = 0, 1, \cdots, m$ do $A[0][j] \leftarrow jA$
5  for $i = 1, 2, \cdots, n$ do
6      for $j = 1, 2, \cdots, m$ do
7          if $x[i] = y[j]$ then
8              $A[i][j] \leftarrow \min\{A[i-1][j-1], A[i-1][j] + A, A[i][j-1] + A\}$
9          else
10             $A[i][j] \leftarrow \min\{A[i-1][j-1] + B, A[i-1][j] + A, A[i][j-1] + A\}$
11 Return: $A[n][m]$
```

It is also clear[27] that this algorithm runs in $\Theta(nm)$ time: $\Theta(m+n)$ for base cases, and $\Theta(nm)$ iterations of the double loop, each time doing $\mathcal{O}(1)$.

***Correctness proof.*** Our goal is to prove that $A[n][m]=\mathrm{OPT}(n,m)$, so we will use strong induction on $i+j$ to prove $A[i][j]=\mathrm{OPT}(i,j)$ for all $0\leqslant i\leqslant n$ and $0\leqslant j\leqslant m$.
Base case: when $i+j=0$ we must have $i=j=0$, and indeed, $A[0][0]=0$, and aligning two empty strings is free of any cost.
Using strong induction on $i+j$, we now prove the inductive step:

(1) If $i=0$, then $a[0][j]=jA$ according to the "base case" stated in the DP formula. And indeed, aligning an empty string with a string of length $j$ involves $j$ insertions whose total cost is $jA$.

(2) If $j=0$, the proof is analogous to above.

‡Indicator function: the second term is $B$ if $x[i]\neq y[j]$ and 0 otherwise.

27 One thing I love about DP is that their runtime analyses are usually extremely simple, since the recurrence relationship is basically all we need to analyze.

(3) If $x[i] = y[j]$, then

$$\begin{aligned}\mathrm{A}[i][j] &= \min\{\mathrm{A}[i-1][j-1], \mathrm{A}[i-1][j] + A, \mathrm{A}[i][j-1] + A\} && \text{(line 8)}\\ &= \min\{\mathrm{OPT}(i-1, j-1), \mathrm{OPT}(i-1, j) + A, \mathrm{OPT}(i, j-1) + A\} && \text{(strong IH on index sum)}\\ &= \mathrm{OPT}(i, j). && \text{(recurrence relation)}\end{aligned}$$

(4) Finally, if $x[i] \neq y[j]$, then

$$\begin{aligned}\mathrm{A}[i][j] &= \min\{\mathrm{A}[i-1][j-1] + B, \mathrm{A}[i-1][j] + A, \mathrm{A}[i][j-1] + A\} && \text{(line 10)}\\ &= \min\{\mathrm{OPT}(i-1, j-1) + B, \mathrm{OPT}(i-1, j) + A, \mathrm{OPT}(i, j-1) + A\} && \text{(strong IH on index sum)}\\ &= \mathrm{OPT}(i, j). && \text{(recurrence relation)}\end{aligned}$$

□

## 4.5 The Knapsack Problem

Suppose there are $n$ items, each having integer weight $w(i)$ and arbitrary value $v(i)$. You can carry a total weight of at most $W$ (assuming for convenience that $W \in \mathbb{Z}$). How to maximize the total value of the selected items?

Motivation: imagine if you were a thief and you broke into a clock shop. Each clock has its own values and weight. You could only carry a certain amount of weight. How would you make the most out of this burglary?

We can come up with easy counterexamples to show that greedy approaches fail[28]. So we resort to DP.

### A DP Approach to the Knapsack Problem

Like before, our trivial insight is that **each item $i$ is either included or not included in the optimal solution**.

As a first approach, we consider the subproblems of form $\mathrm{OPT}(i)$, where we are presented with the same Knapsack problem but with items $1, 2, \cdots, i$ only. Using our observation:

- If $\mathrm{OPT}(i)$ does not include item $i$, then it is as if item $i$ didn't exist: $\mathrm{OPT}(i) = \mathrm{OPT}(i-1)$.
- If $\mathrm{OPT}(i)$ includes item $i$, then its total value consists of $v(i)$ due to item $i$, as well as everything else from $\mathrm{OPT}(i-1)$.

Therefore, we propose the following recurrence relation:

$$\mathrm{OPT}(i) = \max(\mathrm{OPT}(i-1), v(i) + \mathrm{OPT}(i-1))$$

... which just includes every item and is clearly wrong. The problem is **we never used the assumption on total weight $W$**. But if we just stated our subproblem as "optimal solution for items $1, 2, \cdots, i$, of total weight $\leqslant W$," it would instead violate the optimality of sub-solutions: if we included item $i$, the solution for items $1, 2, \cdots, i-1$ now has a lower total weight.

[28] It fails for the integral Knapsack problem, i.e., either take an entire item or leave it, but being greedy on $v(i)/w(i)$ solves fractional Knapsack.

To address this, we introduce an additional variable: the total allowed weight, and redefine $\mathrm{OPT}(\cdot)$ as follows:

$$\mathrm{OPT}(i,w) := \text{maximum total value obtainable from items} \{1,2,\cdots,i\} \text{ with weight limit } w.$$

This time:

- If $\mathrm{OPT}(i,w)$ does not include item $i$, then like before, $\mathrm{OPT}(i,w) = \mathrm{OPT}(i-1,w)$.
- However, if $\mathrm{OPT}(i,w)$ indeed selects item $i$, then the subproblem now has a reduced weight budget $w - v(i)$. If this quantity is negative, then it means $\mathrm{OPT}(i,w)$ could not have chosen item $i$. If now, the equation is now $\mathrm{OPT}(i,w) = v(i) + \mathrm{OPT}(i-1, w - w(i))$.

Combining these observations and adding base cases, we now arrive at the correct recurrence relation:

$$\mathrm{OPT}(i,w) = 0 \qquad \mathrm{OPT}(i,w) = \begin{cases} \mathrm{OPT}(i-1,w) & w < w(i) \\ \mathrm{OPT}(i,w) = \max(\mathrm{OPT}(i-1,w), v(i) + \mathrm{OPT}(i-1, w-w(i))) & w \geqslant w(i). \end{cases}$$

Implementing this recurrence relation is also straightforward:

**Algorithm 11:** Knapsack via DP

```
Inputs: $n$ items, with weights $w(i)$ (integer) and values $v(i)$; total budget $W$ (integer)
allocate $n \times W$ array A[][]
for $w = 0, 1, \cdots, W$ do
    $A[0][w] \leftarrow 0$ // base cases
for $i = 1, 2, \cdots, n$ do
    for $w = 0, 1, \cdots, W$ do
        if $w(i) > w$ then
            $A[i][w] \leftarrow A[i-1][w]$ // DP case 1
        else
            $A[i][w] \leftarrow \max(A[i-1][w], v(i) + A[i-1][w - w(i)])$ // DP case 2
Return: $A[n][W]$
```

***Correctness proof.*** Omitted. Same trick as always. Weak induction on $i$ suffices. □

Further, we can also use backtracking to retrieve the optimal subset of items in $\mathcal{O}(n)$ time by allocating another $n \times W$ table, setting the $(i,w)$ entry to 0 if $A[i][w] = A[i-1][w]$, and 1 otherwise. Then, we backtrack from $(n, W)$, decrementing the values by $i \leftarrow i-1$ and $w \leftarrow w - w(i)$ every time we see a 1 in this table along the path.

### 4.5.1 Pseudo-Polynomial Runtime

Lines 4, 8, and 10 all take $\Theta(1)$ time, so the entire algorithm takes $\Theta(W)$ for base cases and $\Theta(nW)$ for the main DP loop, resulting in a total of $\Theta(nW)$ for the Knapsack problem. This looks polynomial, but is it? Consider $W = 2^{100}$. We can easily store it as a 100-bit binary number, but running the algorithm takes forever. The problem here is that $\mathcal{O}(nW)$, while polynomial w.r.t. $W$, is **not polynomial w.r.t. its input size, $\log_2 W$**.

On the other hand, we also want to distinguish this type of exponential runtime from $2^n$ resulted from exhaustive search, for intuitively the latter is a lot worse. The distinction is that, **if the weights $\{w_i\}, W$ is reasonably small, then DP Knapsack is good, and it only starts to grow bad if we feed it with astronomical values of $W$.**

This leads to the following definition(s):

**Definition: Polynomial & Pseud-Polynomial Algorithms**

We define an algorithm to be **pseudo-polynomial** if it runs in polynomial time when the input is given in unary encoding (e.g. $8 = 11111111$ in unary). We define an algorithm to be **polynomial**[29] if runs in polynomial time when the input is given in binary encoding (e.g. $8 = 1000$ in binary).
(*Note that every pseudo-polynomial algorithm is polynomial.*)

With this definition, it is clear that if we view $W = 2^{100}$ as truly monstrous number using unary encoding ($2^{100}$ 1's in a row), then yes, DP Knapsack is polynomial. But if we view $2^{100}$ as binary, then the runtime becomes polynomial w.r.t. $W$. Therefore, **DP Knapsack is pseudo-polynomial**. (*Of course, no one is ever insane enough to write numbers in unary. This curated definition serves the sole purpose of addressing the issue we discussed above.*)

**But why didn't we bring this up earlier**? The answer is that **it wasn't a problem until now — running times of arithmetic operations are polynomial w.r.t. binary encoding input size.** Consider a simple function that loops through an array $a[1:n]$, computing the sum of all elements. Suppose the largest number in the array is $K$, so we need $\log K$ bits[30] to represent it.

- The total input size is bounded by $[n + \log K, n \log K]$ (if all other numbers are single bit v.s. if all numbers are $K$).
- The largest possible partial sum involved is bounded by $nK$.
- Addition of two numbers $a, b$ takes $\mathcal{O}(\max(\log a, \log b)) = \mathcal{O}(\log a + \log b)$, so each addition takes at most $\mathcal{O}(\log(nK)) = \mathcal{O}(\log n + \log K)$.
- At most $n$ additions, so the total running time is at most $\mathcal{O}(n \log n + n \log K)$.

This space is intentionally left blank

[29] To further break it down, this is actually called *weakly polynomial*. There is yet a stronger category, *strongly polynomial*, that requires the algorithm to run in polynomial time on the number of input items, regardless of input values.

[30] I drop the base 2 here since they are equivalent in big-$\mathcal{O}$ notation.

## 4.6 Shortest Paths, Revisited

Back in CS104/170 we we introduced to the shortest path problem and Dijkstra's algorithm, but we always assumed that edge lengths are nonnegative/positive. Here we relax this assumption and consider the following, more general question:

> Given a directed graph $G = (V, E)$ with edge costs $c(e) \in \mathbb{R}$ (possibly negative), a start node $s$, and a finish node $t$, find a shortest path (minimum sum of edge costs) from $s$ to $t$.

First, we note that with these weaker assumptions, the original Dijkstra immediately fails: consider a graph with $3$ nodes, $s, u, t$. There is one direct edge from $s \to t$ with cost 1. Another path, $s \to u \to t$, has edge costs $c(s,u) = 2$ and $c(u,t) = -2$. Dijkstra would have picked $s \to t$ directly, but in reality the cost of $s \to u \to t$ is $0$.

**For simplicity, we also assume that $G$ contains no negative cycles** (cycles with negative weight sums that are reachable from $s$ and reach $t$): otherwise we can loop through this cycle multiple times and obtain even "cheaper" paths, making our definition of shortest path ill-defined.

### A DP Approach to the Generalized Shortest Path Problem

We first try to define $\mathrm{OPT}(v)$ as the minimum cost of any path from $v$ to $t$. This leads to the trivial base case $\mathrm{OPT}(t) = 0$. For other nodes, the subproblem is decided by the next hop of the path, observing that **the optimal path from $v$ to $t$ has to take one of the edges $(v, u)$ first, then follow the optimal path from $u$ to $t$**. Hence, the general recurrence relation is given by

$$\mathrm{OPT}(v) = \min_{\text{outgoing edges from } v} (c(v,u) + \mathrm{OPT}(u)).$$

This is called the **Bellman equation**. It is correct, but it is not clear how to convert it to a tabular/recursive algorithm. Once again, we are lacking the notion of order here. Hence, like Knapsack, we introduce an auxiliary variable $k$ limiting the total number of hops

$$\mathrm{OPT}(v,k) := \text{minimal cost of } v \to t \text{ paths, with} \leqslant k \text{ hops.}$$

It follows that

$$\begin{cases} \mathrm{OPT}(t,k) = 0 & \text{(base cases) for all } k \\ \mathrm{OPT}(v,0) = \infty & \text{(base cases) for all } v \neq t \\ \mathrm{OPT}(v,k+1) = \min\limits_{u:(v,u) \text{ is an edge}} c(v,u) + \mathrm{OPT}(u,k). & \text{(recurrence)} \end{cases}$$

This is called the **Bellman-Ford algorithm**. Before we implement the algorithm, observe that **the shortest path has $\leqslant n$ nodes** (so $\leqslant n-1$ hops). Otherwise, by pigeonhole principle at least one node would repeat, thus creating a cycle, and by the no-negative-cycle assumption, this cycle has non-negative weight. So we can just remove it and make the path cheaper (or the same, in which case why not?).

With this in mind, we know precisely the size of the tabular dimension DP requires: $n \times n$. Implementation should be a breeze now:

**Algorithm 12:** Generalized Shortest Path via DP; Bellman-Ford

```
1 Inputs: directed graph $G = (V, E)$ with edge costs $c(e)$; start/end nodes $s, t \in V$
2 allocate table of dimension $n \times n$; first argument = node[31], second = path length upper bound
3 $\mathrm{A}[t][0] \leftarrow 0$
4 for all $v \in V$ do $\mathrm{A}[v][0] \leftarrow \infty$
5 for $k = 1, 2, \cdots, n-1$ do
6     for all nodes $v \neq t$ do
7         set $\mathrm{A}[v][k] \leftarrow \min_{u:(v,u) \text{ is an edge}} c(v,u) + \mathrm{A}[u][k-1]$
8 Return: $\mathrm{A}[s][n-1]$[32]
```

***Correctness proof.*** Omitted. Use induction on $k$ and prove that for all $k$, $\mathrm{A}[v][k] = \mathrm{OPT}(v,k)$ for all nodes $v$. Standard three-step proof as always. ☐

Viewing lines 6 and 7 as a whole, we basically inspected each edge once, and so the runtime is $\Theta(m)$, where $m = |E|$. Hence the outer loop has runtime $\Theta(mn)$. This dominates all other work (initialization, base cases), so this implementation of Bellman-Ford runs in $\Theta(mn)$. Slightly worse than Dijkstra's $\Theta(m \log n)$.

**A final question: What if there were negative cycles? How do we detect and find them?** The solution is simple: we run one more iteration (now totaling $n+1$). We showed previously that, if there were no negative cycles, then all results would be final. Therefore, if we obtain different values after running one additional iteration, we know it must be caused by a negative cycle. Put more formally: if $\mathrm{OPT}(u, n-1) \neq \mathrm{OPT}(u, n)$ for some $u$, then there exists a negative cycle along the longer path consisting of $n$ hops and $n+1$ nodes, by our previous pigeonhole argument.

### 4.6.1 (Optional) Adding Asynchronicity: Distance Vector Protocols

In our previous implementation of Bellman-Ford, we adopted a "pull-based" approach, in the sense that each node *actively* contacts its neighbor, asking for information, even when the neighbor often times has no new information to provide.

This pull-based approach is somewhat analogous to an obnoxious person waiting in line at a restaurant, shouting, "Is it my turn yet? Is it my turn yet?" This is both annoying and inefficient, so instead we can consider a "push-based" approach, such that we patiently wait until the reception brings news to us.

Converting this idea back to the shortest path problem, we have an alternate approach to constructing the table: (i) if a node $v$ figures out a better path from itself to $t$, notify all incoming neighbors (nodes $u$ with an $u \to v$ edge), and (ii) otherwise stay put, and patiently wait for updates from its outgoing neighbors to do what is described in (i), so that the only way it figures out a better path exists is if one of its outgoing neighbors told it so.

Think of it as ripple effect, or even BFS. At first, $t$ is the only node knowing how to get to $t$ (trivial case also). Every other node thought $t$ is unreachable, represented by distance $\infty$. Node $t$ then tells all of its incoming neighbors that "hey, you can actually reach me," and then each incoming neighbor tells its own incoming neighbors that "hey, you can actually reach $t$," and so on and so forth.

There is one more thing we'd like to add: **asynchronicity**. The motivation is simple and practical: routers and

[31] You of course need some additional implementation to convert nodes into node IDs. Not included here.

[32] Professor Kempe's notes said we return $\mathrm{A}[s][n]$, but I believe $n-1$ is correct: we want a path of at most $n$ nodes, and hence $\leqslant n-1$ hops.

network. When we have multiple machines, how do we efficiently send a message from one machine to another? Further, we have no control over the performance of each individual machine — some routers may be slower at reporting updates, some may need more time to process the data before doing computations, and so on. Therefore, the BFS-alike, push-based algorithm needs some optimization. Instead of propagating updates in rounds/iterations, we say a node becomes *active* if it needs to spread some update, and once it is done notifying all its incoming neighbors, the state reverts to *inactive* again. The following algorithm implements this idea. Compared to our previous versions, it has the advantage of having no (or little) constraint on the ordering of updates.

**Algorithm 13:** Asynchronous Shortest Path (source: KT §6.9)

```
Inputs: directed graph $G = (V, E)$ with edge costs $c(e)$; start/end nodes $s, t \in V$
allocate node-indexed array $M$ of size $n = |V|$
initialize $M(t) \leftarrow 0$ and $M(v) \leftarrow \infty$ for all $v \neq t$
declare $t$ to be active (and all other nodes inactive)
while there exists an active node do
    $w \leftarrow$ any active node
    y for all edges $v \rightarrow w$ do
        $M(v) \leftarrow \min(M(v), c(v, w) + M(w))$
        if the value of $M(v)$ changed then
            set $v$'s next hop to be $w$ // for backtracing, if needed
            declare $v$ to be active
    declare $w$ to be inactive
Return: backtracing results, and/or $M(s)$
```