Chapter 4

Dynamic programming

By Sariel Har-Peled, December 2, 2021[1]

> The events of 8 September prompted Foch to draft the later legendary signal: "My centre is giving way, my right is in retreat, situation excellent. I attack." It was probably never sent.
>
> – – The first world war, John Keegan..

Version: 0.1

4.1. Basic Idea - Partition Number

Definition 4.1.1. For a positive integer n, the ***partition number*** of n, denoted by $p(n)$, is the number of different ways to represent n as a decreasing sum of positive integers.

The different number of partitions of 6 are shown on the right.

6 = 6		
6=5+1		
6=4+2	6=4+1+1	
6 = 3 + 3	6 = 3 + 2 + 1	6+3+1+1+1
6=2+2+2	6=2+2+1+1	6=2+1+1+1+1
6=1+1+1+1+1+1		

It is natural to ask how to compute $p(n)$. The "trick" is to think about a recursive solution and observe that once we decide what is the leading number d, we can solve the problem recursively on the remaining budget $n - d$ under the constraint that no number exceeds d.

The resulting recursive formula is

$$f(n,d) = \begin{cases} 1 & n \leq 1 \text{ or } d = 1 \\ \sum_{i=\min(d,n)}^{1} f(n-i,i) & \text{otherwise.} \end{cases}$$

One can readily turn this recursive formula into a recursive algorithm, as depicted in Figure 4.1.

Suggestion 4.1.2. Recursive algorithms are one of the main tools in developing algorithms (and writing programs). If you do not feel comfortable with recursive algorithms you should spend time playing with recursive algorithms till you feel comfortable using them. Without the ability to think recursively, this class would be a long and painful torture to you. Speak with me if you need guidance on this topic. TIP

We are interested in analyzing the running time of this algorithm. To this end, draw the recursion tree of Partitions and observe that the amount of work spend at each node, is proportional to the number of children it has. Thus, the overall time spend by the algorithm is proportional to the size of the recurrence tree, which is proportional (since every node is either a leaf or has at least two children) to the number of leafs in the tree, which is $\Theta(p(n))$. See Figure 4.2 for an example.

This is not very exciting, since it is easy verify that $3^{\sqrt{n}/4} \leq p(n) \leq n^n$.

```
PartitionsI(num, d)      //d-max digit
    if (num ≤ 1) or (d = 1)
        return 1
    if d > num
        d ← num
    res ← 0
    for i ← d down to 1
        res = res + PartitionsI(num − i, i)
    return res

Partitions(n)
    return PartitionsI(n, n)
```

Figure 4.1: Recursive algorithm for computing partition number.

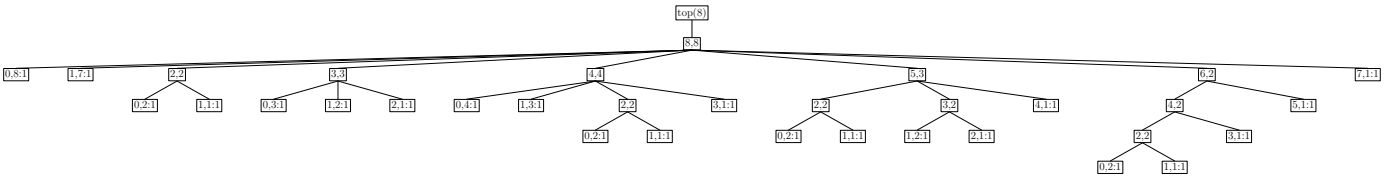


Figure 4.2: The recursion tree of Partitions(8).

Exercise 4.1.3. *Prove the above bounds on $p(n)$ (or better bounds).*

Suggestion 4.1.4. Exercises in the class notes are a natural easy questions for inclusions in exams. You probably want to spend time doing them. $\mathcal{TIP}$

Hardy and Ramanujan (in 1918) showed that $p(n) \approx \dfrac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$ (which I am sure was your first guess).

It is natural to ask, if there is a faster algorithm. Or more specifically, why is the algorithm Partitions so slowwwwwwwwwwwwwwwww? The answer is that during the computation of Partitions(n) the function PartitionsI(num, max_digit) is called a lot of times with the *same* parameters.

An easy way to overcome this problem is cache the results of PartitionsI using a hash table.[2] Whenever PartitionsI is being called, it checks in a cache table if it already computed the value of the function for this parameters, and if so it returns the result. Otherwise, it computes the value of the function and before returning the value, it stores it in the cache. This simple (but powerful) idea is known as ***memoization***.

```
PartitionsI_C(num, max_digit)
    if (num ≤ 1) or (max_digit = 1)
        return 1
    if max_digit > num
        d ← num
    if ⟨num, max_digit⟩ in cache
        return cache(⟨num, max_digit⟩)
    res ← 0
    for i ← max_digit down to 1 do
        res += PartitionsI_C(num − i,i)
    cache(⟨num, max_digit⟩) ← res
    return res

PartitionS_C(n)
    return PartitionsI_C(n, n)
```

What is the running time of PartitionS_C? Analyzing recursive algorithm that have been transformed by memoization are usually analyzed as follows: (i) bound the number of values stored in the hash table, and (ii) bound the amount of work involved in storing one value into the hash table (ignoring recursive calls).

Here is the argument in this case:

(A) If a call to PartitionsI_C takes (by itself) more than constant time, then this call performs a store in the cache.

(B) Number of store operations in the cache is $O(n^2)$, since this is the number of different entries stored in the cache. Indeed, for PartitionsI_C(num, max_digit), the parameters num and max_digit are both integers in the range $1, \ldots, n$.

(C) We charge the work in the loop to the resulting store. The work in the loop is at most $O(n)$ time (since $max\_digit \leq n$).

(D) As such, the overall running time of PartitionS_C(n) is $O(n^2) \times O(n) = O(n^3)$.

Note, that this analysis is naive but it would be sufficient for our purposes (verify that the bound of $O(n^3)$ on the running time is tight in this case).

4.1.1. A Short sermon on memoization

This idea of memoization is generic and nevertheless very useful. To recap, it works by taking a recursive function and caching the results as the computations goes on. Before trying to compute a value, check if it was already computed and if it is already stored in the cache. If so, return result from the cache. If it is not in the cache, compute it and store it in the cache (for the time being, you can think about the cache as being a hash table).

- **When does it work:** There is a lot of inefficiency in the computation of the recursive function because the same call is being performed repeatedly.
- **When it does NOT work:**
  (A) The number of different recursive function calls (i.e., the different values of the parameters in the recursive call) is "large".
  (B) When the function has side effects.

[2] Throughout the course, we will assume that a hash table operation can be done in constant time. This is a reasonable assumption using randomization and perfect hashing. tidbit

Tidbit 4.1.5. Some functional programming languages allow one to take a recursive function $f(\cdot)$ that you already implemented and give you a memorized version $f'(\cdot)$ of this function without the programmer doing any extra work. For a nice description of how to implement it in Scheme see [ASS96].

It is natural to ask if we can do better than just using caching? As usual in life – more pain, more gain. Indeed, in a lot of cases we can analyze the recursive calls, and store them directly in an (sometime multi-dimensional) array. This gets rid of the recursion (which used to be an important thing long time ago when memory, used by the stack, was a truly limited resource, but it is less important nowadays) which usually yields a slight improvement in performance in the real world.

This technique is known as ***dynamic programming***[3]. We can sometime save space and improve running time in dynamic programming over memoization.

Dynamic programming made easy:
(A) Solve the problem using recursion - easy (?).
(B) Modify the recursive program so that it caches the results.
(C) Dynamic programming: Modify the cache into an array.

4.2. Example – Fibonacci numbers

Let us revisit the classical problem of computing Fibonacci numbers.

4.2.1. Why, where, and when?

To remind the reader, in the Fibonacci sequence, the first two numbers $F_0 = 0$ and $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$, for $i > 1$. This sequence was discovered independently in several places and times. From Wikipedia:

> "The Fibonacci sequence appears in Indian mathematics, in connection with Sanskrit prosody. In the Sanskrit oral tradition, there was much emphasis on how long (L) syllables mix with the short (S), and counting the different patterns of L and S within a given fixed length results in the Fibonacci numbers; the number of patterns that are m short syllables long is the Fibonacci number F_{m+1}."

(To see that, imagine that a long syllable is equivalent in length to two short syllables.) Surprisingly, the credit for this formalization goes back more than 2000 years (!)

Fibonacci was a decent mathematician (1170—1250 AD), and his most significant and lasting contribution was spreading the Hindu-Arabic numerical system (i.e., zero) in Europe. He was the son of a rich merchant that spend much time growing up in Algiers, where he learned the decimal notation system. He traveled throughout the Mediterranean world to study mathematics. When he came back to Italy he published a sequence of books (the first one "Liber Abaci" contained the description of the decimal notations system). In this book, he also posed the following problem:

> Consider a rabbit population, assuming that: A newly born pair of rabbits, one male, one female, are put in a field; rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits; rabbits never die

[3] As usual in life, it is not dynamic, it is not programming, and its hardly a technique. To overcome this, most texts find creative ways to present this topic in the most opaque way possible.

```
FibDP(n)
    if n ≤ 1
        return 1
    if F[n] initialized
        return F[n]
    F[n] ⟸FibDP(n − 1)+FibDP(n − 2)
    return F[n]
```
Figure 4.3

and a mating pair always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci posed was: how many pairs will there be in one year?

(The above is largely based on Wikipedia.)

4.2.2. Computing Fibonacci numbers

The recursive function for computing Fibonacci numbers is depicted on the right. As before, the running time of FibR(n) is proportional to $O(F_n)$, where F_n is the nth Fibonacci number. It is known that

```
FibR(n)
    if n = 0
        return 1
    if n = 1
        return 1
    return FibR(n − 1) + FibR(n − 2)
```

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \Theta(\phi^n),$$

where $\phi = \frac{1+\sqrt{5}}{2}$.

We can now use memoization, and with a bit of care, it is easy enough to come up with the dynamic programming version of this procedure, see FibDP in Figure 4.3. Clearly, the running time of FibDP(n) is linear (i.e., $O(n)$).

A careful inspection of FibDP exposes the fact that it fills the array $F[...]$ from left to right. In particular, it only requires the last two numbers in the array.

As such, we can get rid of the array all together, and reduce space needed to $O(1)$: This is a phenomena that is quite common in dynamic programming: By carefully inspecting the way the array/table is being filled, sometime one can save space by being careful about the implementation.

```
FibI(n)
    prev ← 0, curr ← 1
    for i = 1 to n do
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

The running time of FibI is identical to the running time of FibDP. Can we do better?

Surprisingly, the answer is yes, to this end observe that

$$\begin{pmatrix} y \\ x+y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}.$$

As such,

$$\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2\begin{pmatrix} F_{n-3} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3}\begin{pmatrix} F_2 \\ F_1 \end{pmatrix}.$$

Thus, computing the nth Fibonacci number can be done by computing $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3}$.

How to this quickly? Well, we know that $a*b*c = (a*b)*c = a*(b*c)$[4], as such one can compute a^n by repeated squaring, see pseudo-code on the right. The running time of FastExp is $O(\log n)$ as can be easily verified. Thus, we can compute in F_n in $O(\log n)$ time.

```
FastExp(a, n)
    if n = 0 then
        return 1
    if n = 1 then
        return a
    if n is even then
        return (FastExp(a, n/2))^2
    else
        return a * (FastExp(a, (n-1)/2))^2
```

But, something is very strange. Observe that F_n has $\approx \log_{10} 1.68...^n = \Theta(n)$ digits. How can we compute a number that is that large in logarithmic time? Well, we assumed that the time to handle a number is $O(1)$ independent of its size. This is not true in practice if the numbers are large. Naturally, one has to be very careful with such assumptions.

4.3. Dynamic programming – a quick introduction

Consider some optimization problem. For specificity consider the problem of given a directed graph G with positive weights on the edges, and you would like to compute the shortest walk from s to t in G, that visits some k special vertices $u_1, \ldots, u_k$. Let refer to this as the Visit k vertices problem.

Remark 4.3.1 (Motivation). Think about being a traveler that would like to visit some k sites of interest while minimizing your travel cost. This problem is as hard as TSP (so we have to continue with caution), but it should be solvable easily for k small, right?

How to solve this problem? The naive solution is to try and use the shortest path algorithm, but it is easy to see that this is not going to work, because the optimal solution might be quite long, and might involve revisiting some vertices many times (i.e. hubs). We need to come up with a subproblem that is well defined by few parameters, and if we can solve the subproblem then we can solve the original problem. Coming up with the right subproblem to solve for, is the main challenge and could be quite hard. Usually, we look for a property of the optimal solution that any part of it must have.

For example, consider the optimal solution π. It visit u_1 at some point. So lets break it into two parts – the part that goes from s to u_1 (denoted by π_1), and the part that goes from u_1 to t. Let U_1 be set of all the sites of interest that π_1 visits (in its interior), and let U_2 be the set of all sites that π_2 visit. Clearly, π_1 must be the shortest path from s to u_1 that visits in its interior all the sites of U_1. Similarly, π_2 must be the shortest path between u_1 and t that visits all the sites of U_2 in its interior. Namely, we found a way to break up the optimal solution into two subproblems that are smaller. Formally, let $U = \{u_1, \ldots, u_k\}$ and observe that $|U_1| < |U|$ and $|U_2| < |U|$. In particular, let $f(X, u, v)$ be the price of the shortest path between u and v that visits all the sites of X on the way. Let $d(u, v)$ denote the shortest path between u and v. We have the following recursive formula

$$f(X,u,v) = \begin{cases} d(u,v) & |X| = \emptyset \\ \min_{x \in X} \min_{Y \subseteq X \setminus \{x,u,v\}} \Big(f(Y,u,x) + f\big(X \setminus (Y \cup \{x,u,v\}), x, v\big) \Big) & \text{otherwise.} \end{cases}$$

[4] Associativity of multiplication...

This formula makes sense, but we need to make sure that it indeed makes progress. To this end, observe that if we compute $f(X, u, v)$ using this formula, and $t = |X|$, then in the recursive calls, we have

$$\forall Y \subseteq X \setminus \{x, u, v\} : \qquad |Y| < t \qquad \text{and} \qquad |X \setminus (Y \cup \{x, u, v\}| < t.$$

Thus, we indeed make progress in this recursive formula. To spell things even further, consider each recursive call as a configuration defined by its parameters. Clearly, we have $2^k n^2$ different configurations. We put a directed edge between a configuration (X, u, v) and (X', u', v') if computing $f(X, u, v)$ might directly require computing $f(X', u', v')$. This is a dependency graph telling us what we have to compute first. Since $|X'| < |X|$, it must be that the dependency graph is a DAG.

The next observation is that when we compute the value of $f(X, u, v)$ we can just remember this value for future usage, if this specific call is ever made again. This can be done using some kind of table – either an appropriate array or a hash table. This simple idea is crucial as it immediately speeds up things – it is known as ***memoization***.

Computing $f(U, s, t)$ recursively with memoization is a *dynamic programming* solution to the problem. The running time analysis is somewhat counter intuitive – we compute how many distinct recursive calls are there (because we will explicitly compute each of them exactly once), and for each one of them, we compute how much time we spend directly on this call. There are $2^k n^2$ distinct recursive calls, and each one of them performs at most $2 \cdot k2^{k-1}$ recursive calls, which takes $O(2k2^{k-1})$ to compute. We conclude that the overall running time is

$$O\left(2^k n^2 \cdot 2k2^{k-1}\right) = O\left(2^{2k} n^2\right).$$

We also have to add the preprocessing time We precompute $d(u, v)$ for all vertices in the graph. This can be done naively by running Dijkstra n times (here $n = |\mathsf{V}(\mathsf{G})|$ and $m = |\mathsf{E}(\mathsf{G})|$). This takes $O(n(n \log n + m))$ time. Overall, the running time of the algorithm is

$$O\left(2^{2k} n^2 + n^2 \log n + nm\right).$$

There are a lot of low level details that are yet to be resolved:

(A) How exactly do we store the values of $f(X, u, v)$? Especially if want to avoid hashing?

(B) How one can avoid the recursion, and turn this into iterative algorithm? In most cases iterative algorithms are more efficient in practice.

(C) Can one use less space for the memoization.

When you apply the above optimizations you would get the "classical" dynamic programming solution for the optimization problem.

Remark 4.3.2. The above solution is not even remotely the most efficient way to solve this specific problem. Think about how to improve the running time of the above algorithm.

4.4. Edit Distance

We are given two strings A and B, and we want to know how close the two strings are too each other. Namely, how many edit operations one has to make to turn the string A into B?

We allow the following operations: (i) insert a character, (ii) delete a character, and (iii) replace a character by a different character. Price of each operation is one unit.

For example, consider the strings A ="har-peled" and B ="sharp eyed". Their ***edit distance*** is 4, as can be easily seen.

	h	a	r	–	p		e	l	e	d
s	h	a	r		p	<space>	e	y	e	d
1	0	0	0	1	0	1	0	1	0	0

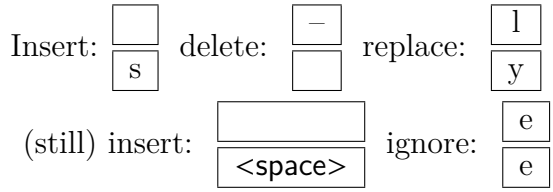


Figure 4.4: Interpreting edit-distance as a alignment task. Aligning identical characters to each other is free of cost. The price in the above example is 4. There are other ways to get the same edit-distance in this case.

But how do we compute the edit-distance (min # of edit operations needed)?

The idea is to list the edit operations from left to right. Then edit distance turns into a an alignment problem. See Figure 4.4.

In particular, the idea of the recursive algorithm is to inspect the last character and decide which of the categories it falls into: insert, delete or ignore. See pseudo-code on the right.

```
ed(A[1..m], B[1..n])
    if m = 0 return n
    if n = 0 return m
    p_insert = ed(A[1..m], B[1..(n − 1)]) + 1
    p_delete = ed(A[1..(m − 1)], B[1..n]) + 1
    p_r/i = ed(A[1..(m − 1)], B[1..(n − 1)] )
                + [A[m] ≠ B[n]]
    return min(p_insert, p_delete, p_replace/ignore)
```

The running time of ed(...)? Clearly exponential, and roughly 2^{n+m}, where $n+m$ is the size of the input.

So how many **different** recursive calls ed performs? Only:$O(m*n)$ different calls, since the only parameters that matter are n and m.

So the natural thing is to introduce memoization. The resulting algorithm edM is depicted on the right. The running time of edM(n,m) when executed on two strings of length n and m respective is $O(nm)$, since there are $O(nm)$ store operations in the cache, and each store requires $O(1)$ time (by charging one for each recursive call). Looking on the entry $T[i,j]$ in the table, we realize that it depends only on $T[i-1,j]$, $T[i,j-1]$ and $T[i-1,j-1]$. Thus, instead of recursive algorithm, we can fill the table T row by row, from left to right.

```
edM(A[1..m], B[1..n])
  if m = 0 return n
  if n = 0 return m
  if T[m, n] is initialized then return T[m, n]
  p_insert = edM(A[1..m], B[1..(n − 1)]) + 1
  p_delete = edM(A[1..(m − 1)], B[1..n]) + 1
  p_r/i = edM(A[1..(m − 1)], B[1..(n − 1)]) + [A[m] ≠ B[n]]
  T[m, n] ← min(p_insert, p_delete, p_replace/ignore)
  return T[m, n]
```

			A		L		G		O		R		I		T		H		M
	0	←	1	←	2	←	3	←	4	←	5	←	6	←	7	←	8	←	9
	↑	⇖																	
A	1		0	←	1	←	2	←	3	←	4	←	5	←	6	←	7	←	8
	↑		↑	⇖															
L	2		1		0	←	1	←	2	←	3	←	4	←	5	←	6	←	7
	↑		↑		↑	⇖								↖					
T	3		2		1		1	←	2	←	3	←	4		4	←	5	←	6
	↑		↑		↑		↑	⇖		↖									
R	4		3		2		2		2		2	←	3	←	4	←	5	←	6
	↑		↑		↑	↖		↖		⇖		↖							
U	5		4		3		3		3		3		3	←	4	←	5	←	6
	↑		↑		↑	↖		↖		↖		⇖							
I	6		5		4		4		4		4		3	←	4	←	5	←	6
	↑		↑		↑		↑		↑		↑		⇑						
S	7		6		5		5		5		5		4	←	4	←	5	←	6
	↑		↑		↑		↑		↑		↑		↑	⇖					
T	8		7		6		6		6		6		5		4	←	5	←	6
	↑		↑		↑		↑		↑		↑		↑		↑	⇖			
I	9		8		7		7		7		7		6		5		5	←	6
	↑		↑		↑		↑		↑		↑		↑		↑		↑	⇖	
C	10		9		8		8		8		8		7		6		6		6

Figure 4.5: Extracting the edit operations from the table.

```
edDP(A[1..m], B[1..n])
  for i = 1 to m do T[i, 0] ← i
  for j = 1 to n do T[0, j] ← j
  for i ← 1 to m do
      for j ← 1 to n do
          p_insert = T[i, j − 1] + 1
          p_delete = T[i − 1, j] + 1
          p_r/ignore = T[i − 1.j − 1] + [A[i] ≠ B[j]]
          T[i, j] ← min(p_insert, p_delete, p_r/ignore)
  return T[m, n]
```

The dynamic programming version that uses a two dimensional array is pretty simple now to derive and is depicted on the left. Clearly, it requires $O(nm)$ time, and $O(nm)$ space. See the pseudo-code of the resulting algorithm edDP on the left.

It is enlightening to think about the algorithm as computing for each $T[i, j]$ the cell it got the value from. What you get is a tree encoded in the table. See Figure 4.5. It is now easy to extract from the table the sequence of edit operations that realizes the minimum edit distance between A and B. Indeed, we start a walk on this graph from the node corresponding to $T[n, m]$. Every time we walk left, it corresponds to a deletion, every time we go up, it corresponds to an insertion, and going sideways corresponds to either replace/ignore.

Note, that when computing the ith row of $T[i, j]$, we only need to know the value of the cell to the left of the current cell, and two cells in the row above the current cell. It is thus easy to verify that the algorithm needs only the remember the current and previous row to compute the edit distance. We conclude:

Theorem 4.4.1. *Given two strings A and B of length n and m, respectively, one can compute their edit distance in $O(nm)$. This uses $O(nm)$ space if we want to extract the sequence of edit operations, and*

$O(n+m)$ *space if we only want to output the price of the edit distance.*

Exercise 4.4.2. *Show how to compute the sequence of edit-distance operations realizing the edit distance using only $O(n+m)$ space and $O(nm)$ running time. (Hint: Use a recursive algorithm, and argue that the recursive call is always on a matrix which is of size, roughly, half of the input matrix.)*

4.4.1. Shortest path in a DAG and dynamic programming

Given a dynamic programming problem and its associated recursive program, one can consider all the different possible recursive calls, as ***configurations***. We can create graph, every configuration is a node, and an edge is introduced between two configurations if one configuration is computed from another configuration, and we put the additional price that might be involved in moving between the two configurations on the edge connecting them. As such, for the edit distance, we have directed edges from the vertex (i, j) to $(i, j-1)$ and $(i-1, j)$ both with weight 1 on them. Also, we have an edge between (i, j) to $(i-1, j-1)$ which is of weight 0 if $A[i] = B[j]$ and 1 otherwise. Clearly, in the resulting graph, we are asking for the shortest path between (n, m) and $(0, 0)$.

And here are where things gets interesting. The resulting graph G is a DAG (***directed acyclic graph***[5]). DAG can be interpreted as a partial ordering of the vertices, and by topological sort on the graph (which takes linear time), one can get a full ordering of the vertices which agrees with the DAG. Using this ordering, one can compute the shortest path in a DAG in linear time (in the size of the DAG). For edit-distance the DAG size is $O(nm)$, and as such this algorithm takes $O(nm)$ time.

This interpretation of dynamic programming as a shortest path problem in a DAG is a useful way of thinking about it, and works for many dynamic programming problems.

More surprisingly, one can also compute the longest path in a DAG in linear time. Even for negative weighted edges. This is also sometime a problem that solving it is equivalent to dynamic programming.

4.5. Shortest paths in a graph

4.5.1. Bellman-Ford

You are given a graph G with weights on its edges (potentially with negative weights), and a start vertex s. For an edge $e = (u, v)$, let $\ell(e) = \ell(u, v)$ denote the weight of the edge. We would like to compute the shortest path from s to all the vertices of G. If there is a walk from s to any vertex x, that involves a negative cycle, then the algorithm should stop and report that the input instance is ill defined and contains a negative cycle.

To this end, let $g(v, i)$ be the length of the shortest walk from s to v that uses at most i edges. We have the following recursive definition for g:

$$g(v,i) = \begin{cases} 0 & i = 0 \text{ and } v = s \\ \infty & i = 0 \text{ and } v \neq s \\ \min \begin{cases} \min_{e=(u,v)\in \mathsf{E}(\mathsf{G})} \big(g(u, i-1) + \ell(e)\big) \\ g(v, i-1) \end{cases} & \text{otherwise.} \end{cases}$$

Clearly, this recursive formula makes progress as the second parameter i decreases in the recursive calls. We are going to compute $g(v, n+1)$, for all $v \in \mathsf{V}(\mathsf{G})$. The recursive function can be turned into a

[5] No cycles in the graph – its a miracle!

```
BellmanFordR(v, i):
  if i = 0 and v = s then return 0
  if i = 0 then return +∞
  α ← BellmanFordR(v, i − 1)

  for e = (u, v) ∈ E(G) do
      α ← min(α, BellmanFordR(u, i − 1) + ℓ(e))

  return α
```

Figure 4.6: The extremely inefficient recursive version of Bellman-Ford.

```
bfM(v, i):
  if C[v, i] is defined then
    return C[v, i]
  if i = 0 and v = s then
    C[v, i] ← 0
    return 0
  if i = 0 then
    C[v, i] ← +∞
    return +∞
  α ← bfM(v, i − 1)
  for e = (u, v) ∈ E(G) do
      α ← min(α, bfM(u, i − 1) + ℓ(e))

  C[v, i] ← α
  return α
```

```
BellmanFordM(G, s):
  C ← empty hash table
  // I.e., C is an associative map.
  Init array d[] indexed by the vertices of G
  for v ∈ V(G) do
    d[v] ← bfM(v, n)
  return d[···]
```

Figure 4.7: The memoized version of Bellman-Ford.

recursive code in a pretty straight fashion way, as demonstrated in Figure 4.6. It is not hard to see that the running time of this function in practice is going to be exponential, since we are going to repeated compute the same values again and again.

To get a faster algorithm, we are going to use memoization, so we do not repeat computation. We also preprocess the graph, so that each vertex has the list of incoming edges. This preprocessing takes $O(n+m)$ time. With preprocessing, one can compute $g(v,i)$, for all $v \in \mathsf{V}(\mathsf{G})$, in $O(n+m)$ time, assuming that $g(v,i-i)$ is given to us for all $v \in \mathsf{V}(\mathsf{G})$. Namely, using memoization, one can compute $g(v,n+1)$, for all $v \in \mathsf{V}(\mathsf{G})$, in $O(n(n+m))$ time. This memoized version is depicted in Figure 4.7.

We will next show how to convert it into an iterative code, but first, let us worry about negative cycles.

4.5.1.1. Negative cycles reachable from s

Lemma 4.5.1. *There is a negative cycle reachable from s in G $\iff$ there exists a vertex $v \in \mathsf{V}(\mathsf{G})$, such that $g(v,n) < g(v,n-1)$.*

```
Bellman-Ford(G, s):
 for each u ∈ V do
  d(u) ← ∞
 d(s) ← 0

 for k = 1 to n − 1 do
   for each v ∈ V do
    for each edge (u, v) ∈ in(v) do
     d(v) = min{d(v), d(u) + ℓ(u, v)}

 (* One more iteration to check if distances change *)
 for each v ∈ V do
  for each edge (u, v) ∈ in(v) do      // in(v): All edges coming into v
   if (d(v) > d(u) + ℓ(u, v))
    Output "Negative Cycle"

 for each v ∈ V do
   d(s, v) ← d(v)
```

Figure 4.8

Proof: Observe that $g(v,i)$ is the length of the shortest walk from s to v using at most i edges. As such, if $g(v,n) < g(v,n-1)$ then there must be a walk using exactly n edges, from s to v, that is shorter from any walk using fewer edges (indeed, a change in distances happens in iteration i only if it corresponds to a walk that uses exactly i edges). But such a walk must include a cycle, and this cycle must be negative as otherwise, it can be removed and result in a shorter walk.

As for the other direction, let $C = u_0, u_2, \ldots, u_{k-1}$ be a negative cycle reachable from s, and observe that

$$d(u_i, n) \le d(u_{i-1}, n-1) + \ell(u_{i-1}, u_i),$$

where $u_{-1} = u_{k-1}$. Adding up all these inequalities for all i, we have

$$\sum_{i=0}^{k-1} d(u_i, n) \le \sum_{i=0}^{k-1} d(u_{i-1}, n-1) + \sum_{i=0}^{k-1} \ell(u_{i-1}, u_i),$$

Rearranging, we have

$$\sum_{i=0}^{k-1} \Big(d(u_i, n) - d(u_i, n-1) \Big) = \sum_{i=0}^{k-1} \Big(d(u_i, n) - d(u_{i-1}, n-1) \Big) \le \ell(C) < 0.$$

Namely, there must be an index i, such that $d(u_i, n) < d(u_i, n-1)$, as claimed. ∎

4.5.1.2. The iterative version

Using a two dimensional array, and turning this into an iterative algorithm, we get the algorithm depicted in Figure 4.8.

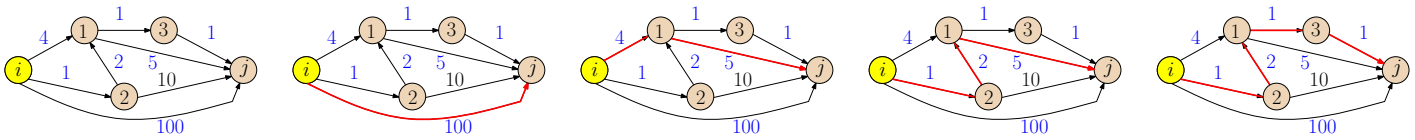


Figure 4.9: The intermediate shortest paths: $\mathsf{dist}(i,j,0) = 100$, $\mathsf{dist}(i,j,1) = 9$, $\mathsf{dist}(i,j,2) = 8$, $\mathsf{dist}(i,j,3) = 5$.

4.5.1.3. The result

Theorem 4.5.2 (Bellman-Ford algorithm). *Let G be a directed graph with n vertices and m edges, with weights (potentially negative) on the edges, and let s be a vertex in G. One can compute in $O(nm)$ time the shortest path from s to all the vertices of G, or alternative, output that there is a negative length cycle reachable from s.*

Remark 4.5.3. A nice consequence of the above algorithm is that one can modify it, so that it detects if there is any negative cycle in G in $O(n+m)$ time.

4.5.2. Floyd-Warshall: All-Pairs Shortest Paths

Definition 4.5.4. The input is (undirected or directed) graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with edge lengths. For edge $e = (u,v)$, let $\ell(e) = \ell(u,v)$ denote its length. The task at hand is compute the shortest paths for all pairs of nodes in G.

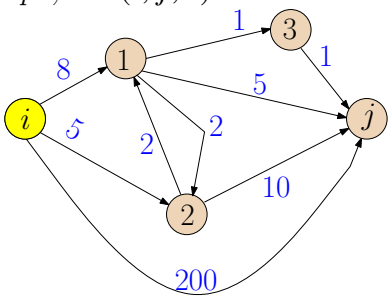
The easy solution is to apply single-source algorithms n times, once for each vertex. If there are no negative edges, then this can be done in $O(nm + n^2 \log n)$ time using Dijkstra. If there are negative length edges, then the running time is $O(n^2 m)$, which is $\Theta(n^4)$ if $m = \Omega(n^2)$ (by using Bellman-Ford.

A better recursive solution comes from guessing the intermediate vertex. To this end, number the vertices of G arbitrarily as $v_1, v_2, \ldots, v_n$. Let

$$\mathsf{dist}(i,j,k)$$

denote the length of shortest walk from v_i to v_j among all walks in which the largest index of an *intermediate node* is at most k (could be $-\infty$ if there is a negative length cycle).

Exercise 4.5.5. *For the following graph,* $\mathsf{dist}(i,j,2)$ *is:*



The two possibilities of $\mathsf{dist}(i,j,k)$ are depicted in Figure 4.10, and we get the following recursive formula:

$$\mathsf{dist}(i,j,k) = \begin{cases} \ell(i,j) & k = 0 \text{ and } (i,j) \in \mathsf{E}(\mathsf{G}) \\ +\infty & k = 0 \text{ and } (i,j) \notin \mathsf{E}(\mathsf{G}) \\ \min \begin{cases} \mathsf{dist}(i,j,k-1) \\ \mathsf{dist}(i,k,k-1) + \mathsf{dist}(k,j,k-1). \end{cases} & k > 0. \end{cases}$$

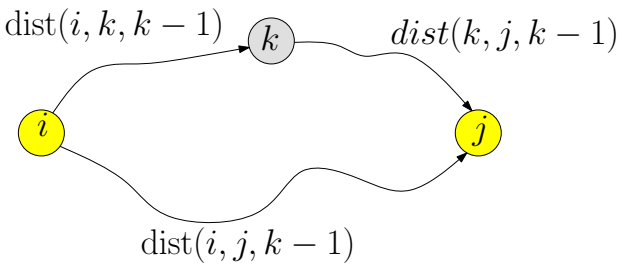


Figure 4.10

```
Floyd-Warshall:
  for i = 1 to n do
    for j = 1 to n do
      dist(i, j, 0) = ℓ(i, j)      // ℓ(i, j) = ∞ if (i, j) ∉ E(G), and 0 if i = j
      next(i, j) = −1
  for k = 1 to n do
    for i = 1 to n do
      for j = 1 to n do
        if (dist(i, j, k − 1) > dist(i, k, k − 1) + dist(k, j, k − 1)) then
            dist(i, j, k) = dist(i, k, k − 1) + dist(k, j, k − 1)
            next(i, j) = k
  for i = 1 to n do
    if (dist(i, i, n) < 0) then
      Output that there is a negative length cycle in G
```

Figure 4.11: Floyd-Warshall Algorithm

The correctness of the above recursive formula follows, since if shortest walk from i to j goes through k, then k occurs only once on the path — otherwise there is a negative length cycle. If $\mathsf{dist}(k, k, k-1) < 0$, then the graph G contains a negative length cycle (that contains k).

The dynamic programming version of the resulting algorithm is depicted in Figure 4.11.

The resulting running time is $\Theta(n^3)$, and the space used is $\Theta(n^3)$.

The correctness follows via induction on the recursive definition. We omit the tedious details here.

Floyd-Warshall Algorithm: Finding the Paths. A natural question is how to compute the paths in addition to the distances. The idea is as follows:

- Create a $n \times n$ array `next` that stores the next vertex on shortest path for each pair of vertices
- Modify the algorithm so that it computes the intermediate vertex that was used during the computation, and run this algorithm. Figure 4.11 depicts this modified version.
- With the two-dimensional array `next`, for any pair of given vertices i, j can compute a shortest path in $O(n)$ time.

Exercise 4.5.6. *Given* `next` *array and any two vertices* i, j *describe an* $O(n)$ *algorithm to find a* i*-*j *shortest path.*

Single source

	name/technique	running time
No negative edges	Dijkstra	$O(n \log n + m)$
Edge lengths can be negative	Bellman Ford	$O(nm)$

All pair shortest path

No negative edges	n * Dijkstra	$O\left(n^2 \log n + nm\right)$
No negative cycles	n * Bellman Ford	$O\left(n^2 m\right) = O\left(n^4\right)$
No negative cycles (*)	BF + n * Dijkstra	$O\left(nm + n^2 \log n\right)$
No negative cycles	Floyd-Warshall	$O\left(n^3\right)$
Unweighted	Matrix multiplication	$O(n^{2.38})$, $O(n^{2.58})$

Table 4.1: Summary of known results on shortest paths.

Bibliography

[ASS96] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs.* MIT Press, 1996.