Chapter 4

Dynamic programming

By Sariel Har-Peled, December 7, 2009[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.

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		

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

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..

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

The resulting algorithm is depicted on the right. We are interested in analyzing its running time. 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 $\Theta(p(n))$.

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)
```

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.

In fact, Hardy and Ramanujan (in 1918) showed that $p(n) \approx \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$ (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 idea is known as ***memoization***.

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:

```
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)
```

[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.

1. If a call to **PartitionsI_C** takes (by itself) more than constant time, then we perform a store in the cache.

2. Number of store operations in the cache is $O(n^2)$.

3. We charge the work in the loop to the resulting store. The work in the loop is $O(n)$.

4. Running time of **PartitionS_C**(n) is $O(n^3)$.

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

4.1.1 Memoization:

This idea of memoization is very generic and very useful. To recap, it just says to take a recursive function and cache the results as the computations goes on. Before trying to compute a value, check if it was already computed and if it is already 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 (i.e., hash table).

- **When does it work:** When there is a lot of inefficiency in the computation of the recursive function because we perform the same call again and again.

- **When it does NOT work:**

  1. When the number of different recursive function calls (i.e., the different values of the parameters in the recursive call) is "large".
  2. When the function has side effects.

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]. tidbit

It is natural to ask if we can do better than just than 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 (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.

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

Dynamic programing made easy.

1. Solve the problem using recursion - easy (?).

2. Modify the recursive program so that it caches the results.

3. Dynamic programming: Modify the cache into an array.

4.2 Fibonacci numbers

Let us revisit the classical problem of computing Fibonacci numbers. The recursive call to do so 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 ≤ 1
    return 1
  return FibR(n − 1)+FibR(n − 2)
```

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

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** on the right. Clearly, the running time of **FibDP**(n) is linear (i.e., $O(n)$).

```
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]
```

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
    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, if 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 $a*b*c = (a*b)*c = a*(b*c)$, 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? Inherently, we assumed that the time to handle a number is $O(1)$. This is not true in practice if the numbers are large. Be careful with such assumptions.

4.3 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 ="harpeled" and B ="sharp eyed". Their ***edit distance*** is 4, as can be easily seen.

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 4.1.

	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

Insert: $\begin{array}{|c|}\hline \\ \hline \text{s} \\ \hline \end{array}$ delete: $\begin{array}{|c|}\hline – \\ \hline \\ \hline \end{array}$ replace: $\begin{array}{|c|}\hline \text{l} \\ \hline \text{y} \\ \hline \end{array}$

insert: $\begin{array}{|c|}\hline \\ \hline \text{<space>} \\ \hline \end{array}$ ignore: $\begin{array}{|c|}\hline \text{e} \\ \hline \text{e} \\ \hline \end{array}$

Figure 4.1: 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.

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]
```

```
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 4.2. 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 4.3.1 *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 4.3.2 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.)

			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 4.2: Extracting the edit operations from the table.

Bibliography

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