Introduction to Dynamic Programming

André Ryser

November 4, 2018

Swiss Olympiad in Informatics

Table of Contents

What is dynamic programming?

Dynamic programming is...

What is dynamic programming?

Dynamic programming is...

- **not** an algorithm

What is dynamic programming?

Dynamic programming is...

- **not** an algorithm
- a technique

What is dynamic programming?

Dynamic programming is...

- **not** an algorithm
- a technique for solving problems (in particular optimization problems)

What is dynamic programming?

Dynamic programming is...

- **not** an algorithm
- a technique for solving problems (in particular optimization problems) more efficiently.

What is dynamic programming

DP is a technique that you may use when you can divide a problem into subproblems and build the full solution using the partial solutions, but the subproblems overlap and you end up solving the same subproblems over and over again.

What is dynamic programming

DP is a technique that you may use when you can divide a problem into subproblems and build the full solution using the partial solutions, but the subproblems overlap and you end up solving the same subproblems over and over again.
A dynamic program avoids this problem by **remembering** what it has already done and not computing it again.

A simple example: Binomial coefficient

Binomial coefficient

Binomial coefficient intuition: How many ways to pick k elements out of a set with n elements.

Binomial coefficient

Binomial coefficient intuition: How many ways to pick k elements out of a set with n elements. Definition:

$$Binom(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!} \forall 0 \leq k \leq n$$

What's the problem?

Binomial coefficient

Binomial coefficient intuition: How many ways to pick k elements out of a set with n elements. Definition:

$$Binom(n, k) = \binom{n}{k} = \frac{n!}{k!(n-k)!} \forall 0 \leq k \leq n$$

What's the problem?

Arithmetic Overflow, 20! already doesn't fit into a 64bit integer.

Recursive Definition

To calculate the binomial coefficient we can also use

$$\begin{cases} \binom{n}{0} = \binom{n}{n} = 1 \forall n \geq 0 \\ \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \forall 1 \leq k \leq n-1 \end{cases}$$

Binomial coefficient: Intuitive algorithm

An intuitive way of computing the binomial coefficient woud be:

```
int Binom(int n, int k) {
  if(k==0 || k == n) return 1;
  return Binom(n-1, k-1) + Binom(n-1, k);
}
```

Binomial coefficiente: The problem

If you try to run this code to compute $\binom{100}{10}$, you'd have to be **very** patient to get an answer.

Binomial coefficiente: The problem

If you try to run this code to compute $\binom{100}{10}$, you'd have to be **very** patient to get an answer.

Why?

Binomial coefficiente: The problem

Our program computes the same values over and over.

Demo at whiteboard of $\binom{4}{2}$

Binomial coefficiente: The problem

This example shows us that the number of operations roughly doubles with n.

Binomial coefficiente: The problem

This example shows us that the number of operations roughly doubles with n.
We have an exponential running time...

Binomial coefficiente: The solution

We can do (much) better.

Binomial coefficiente: The solution

We can do (much) better. We don't need to compute anything twice:

Binomial coefficiente: The solution

We can do (much) better. We don't need to compute anything twice:
Just remember the previous values!

Binomial coefficiente: The solution

We can do (much) better. We don't need to compute anything twice:
Just remember the previous values! We can first compute lower values and then combine them to get the next one.

Binomial coefficiente: The solution

We can do (much) better. We don't need to compute anything twice:
Just remember the previous values! We can first compute lower values and then combine them to get the next one. Demo at whiteboard

Pascal's triangle

A nicer a way to present it.

					1					
				1		1				
			1		2		1			
		1		3		3		1		
	1		4		6		4		1	
1		5		10		10		5		1

Binomial coefficient: The solution

We compute all values (once) from $\binom{0}{0}$ up to $\binom{n}{k}$:

```
int binom(int n, int k) {
  vector<vector<int> > b(n+1, vector<int>(k+1));

  for(int i = 0; i <=n) b[i][0] = 1;
  for(int j = 0; j <=k) b[j][j] = 1;

  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= i; j++)
      b[i][j] = b[i-1][j-1] + b[i-1][j];
  return b[n][k];
}
```

Binomial coefficient: The solution

We compute all values (once) from $\binom{0}{0}$ up to $\binom{n}{k}$:

```
int binom(int n, int k) {
  vector<vector<int> > b(n+1, vector<int>(k+1));

  for(int i = 0; i <=n) b[i][0] = 1;
  for(int j = 0; j <=k) b[j][j] = 1;

  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= i; j++)
      b[i][j] = b[i-1][j-1] + b[i-1][j];
  return b[n][k];
}
```

Our running time is now down to...

Binomial coefficient: The solution

We compute all values (once) from $\binom{0}{0}$ up to $\binom{n}{k}$:

```
int binom(int n, int k) {
  vector<vector<int> > b(n+1, vector<int>(k+1));

  for(int i = 0; i <=n) b[i][0] = 1;
  for(int j = 0; j <=k) b[j][j] = 1;

  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= i; j++)
      b[i][j] = b[i-1][j-1] + b[i-1][j];
  return b[n][k];
}
```

Our running time is now down to $\mathcal{O}(n^2)$.

Binomial coefficient: bonus solution

Bonus solution: you don't need $\mathcal{O}(n^2)$ space.

```
int binom(int n, int k) {
  vector<int> b(n+1, 1);
  for(int i = 1; i <= n; i++) {
    for(int j = i-1; j > 0; j--) {
      b[j] +=b[j-1];
    }
  }
  return b[k];
}
```

The recipe for creating a good DP solution

The recipe for creating a good DP solution

This was a simple example, but the same schemata apply to much more complicated problems. We shall now generalize what we've learned from Binomial coefficient and apply it to other problems.

DP's four steps

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.
2. Find a general recurrence formula to solve a subproblem using the solution to other subproblems.

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.
2. Find a general recurrence formula to solve a subproblem using the solution to other subproblems.
3. Find base case(s).

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.
2. Find a general recurrence formula to solve a subproblem using the solution to other subproblems.
3. Find base case(s).
4. Which is the relevant subproblem?

DP's four steps and Binomial coefficient

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems:

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems: $\binom{i}{j}$.

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems: $\binom{i}{j}$.
2. General formula:

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems: $\binom{i}{j}$.
2. General formula: $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j-1}$.

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems: $\binom{i}{j}$.
2. General formula: $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j-1}$.
3. Base cases:

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems: $\binom{i}{j}$.
2. General formula: $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j-1}$.
3. Base cases: $\binom{i}{0} = 1, \binom{i}{i} = 1$.

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems: $\binom{i}{j}$.
2. General formula: $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j-1}$.
3. Base cases: $\binom{i}{0} = 1, \binom{i}{i} = 1$.
4. Relevant suproblem:

DP's four steps and Binomial coefficient

How does our solution for Binomial coefficient match our four steps?

1. Suproblems: $\binom{i}{j}$.
2. General formula: $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j-1}$.
3. Base cases: $\binom{i}{0} = 1, \binom{i}{i} = 1$.
4. Relevant suproblem: $\binom{n}{k}$.

How to implement a DP solution

Subproblem ordering

— Okay, I've followed your four steps. How do I use this to code a solution now?

Subproblem ordering

Most subproblems can be solved only using other subproblems.

Subproblem ordering

Most subproblems can be solved only using other subproblems.

In what order can we compute subproblems?

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.
3. Continue computing further subproblems which are now solvable.

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.
3. Continue computing further subproblems which are now solvable.
4. When the relevant subproblem is found, return the result!

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.
3. Continue computing further subproblems which are now solvable.
4. When the relevant subproblem is found, return the result!

This is the hardest part of most difficult dynamic programming problems. Sometimes, a viable ordering is obvious, sometimes it is not; the best way to get used to it is to solve a lot of this kind of problems.

Another example: Rod cutting

Rod cutting: Task statement

The problem ist the following:

Rod cutting: Task statement

The problem ist the following:

- Serling Enterprises buys long stell rods and cuts them into shorter rods, which it then sells.

Rod cutting: Task statement

The problem ist the following:

- Serling Enterprises buys long stell rods and cuts them into shorter rods, which it then sells.
- They want to know how to cut them to make the most profit.

Rod cutting: Task statement

The problem ist the following:

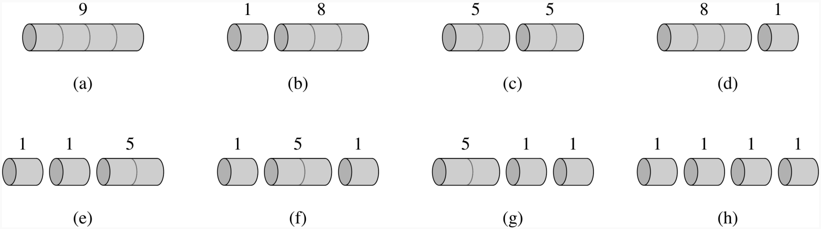
- Serling Enterprises buys long stell rods and cuts them into shorter rods, which it then sells.
- They want to know how to cut them to make the most profit.
- They get delivered rods of length n.

Rod cutting: Task statement

The problem ist the following:

- Serling Enterprises buys long stell rods and cuts them into shorter rods, which it then sells.
- They want to know how to cut them to make the most profit.
- They get delivered rods of length n.
- Given are for $i = 1, 2, \ldots, n$ the price p_i they can charge for a rod of length $i$cm.

Rod cutting: Example



length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	10

Rod Cutting: DP's four steps

How do we modelize this problem using the four steps ?

Rod Cutting: Finding Subproblems

Suppose we know how to optimally cut rods of length $1, 2, \ldots, k-1$: $r_1, r_2, \ldots, r_{k-1}$. We have the following possibilites to cut a rod:

Rod Cutting: Finding Subproblems

Suppose we know how to optimally cut rods of length $1, 2, \ldots, k-1$: $r_1, r_2, \ldots, r_{k-1}$. We have the following possibilites to cut a rod:

1. Don't cut just charge p_k

Rod Cutting: Finding Subproblems

Suppose we know how to optimally cut rods of length $1, 2, \ldots, k-1$: $r_1, r_2, \ldots, r_{k-1}$. We have the following possibilites to cut a rod:

1. Don't cut just charge p_k
2. Cut off a piece of length 1 at the end: $p_1 + r_{k-1}$.

Rod Cutting: Finding Subproblems

Suppose we know how to optimally cut rods of length $1, 2, \ldots, k-1$: $r_1, r_2, \ldots, r_{k-1}$. We have the following possibilites to cut a rod:

1. Don't cut just charge p_k
2. Cut off a piece of length 1 at the end: $p_1 + r_{k-1}$.
3. Cut off a pice of length 2 at the end: $p_2 + r_{k-2}$.

Rod Cutting: Finding Subproblems

Suppose we know how to optimally cut rods of length $1, 2, \ldots, k-1$: $r_1, r_2, \ldots, r_{k-1}$. We have the following possibilites to cut a rod:

1. Don't cut just charge p_k
2. Cut off a piece of length 1 at the end: $p_1 + r_{k-1}$.
3. Cut off a pice of length 2 at the end: $p_2 + r_{k-2}$.
4. ...

Rod Cutting: Finding Subproblems

Suppose we know how to optimally cut rods of length $1, 2, \ldots, k-1$: $r_1, r_2, \ldots, r_{k-1}$. We have the following possibilites to cut a rod:

1. Don't cut just charge p_k
2. Cut off a piece of length 1 at the end: $p_1 + r_{k-1}$.
3. Cut off a pice of length 2 at the end: $p_2 + r_{k-2}$.
4. ...
5. Cut off a pice of length k-1 at the end: $p_{k-1} + r_1$

Rod Cutting: Finding Subproblems

Suppose we know how to optimally cut rods of length $1, 2, \ldots, k-1$: $r_1, r_2, \ldots, r_{k-1}$. We have the following possibilites to cut a rod:

1. Don't cut just charge p_k
2. Cut off a piece of length 1 at the end: $p_1 + r_{k-1}$.
3. Cut off a pice of length 2 at the end: $p_2 + r_{k-2}$.
4. ...
5. Cut off a pice of length k-1 at the end: $p_{k-1} + r_1$

r_k should be maximised, take the max of all possibilities.

$$r_k = \max(p_k, p_1 + r_{k-1}, p_2 + r_{k-2}, \ldots, p_{k-1} + r_1)$$

Rod cutting: Base case

The problem is trivial for $k = 0$: $r_0 = 0$

An argument coud be made that no base case is necessairy. The previous formula r_1 does not need any other r_i.

However with r_0 we can rewrite it:

$$r_k = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

.

Rod cutting: Slow Solution

```
cut_rod(vector<int> &p, n) {
  if(n == 0)
    return 0;
  r = -1;
  for(int i = 1; i <= n; i++) {
    r = max(r, p[i] + cut_rod(p, n-i))
  }
  return r;
}
```

Rod cutting: Computation

We don't want to compute subproblems twice. We just compute them in the order in which they're needed and store them.

Rod cutting: Computation

We don't want to compute subproblems twice. We just compute them in the order in which they're needed and store them.
The order in which they're needed, like for Binomial Coefficient, is pretty obvious. Every subproblem relies on earlier subproblems only, so we solve them in increasing order.

Rod cutting: Computation

We don't want to compute subproblems twice. We just compute them in the order in which they're needed and store them.

The order in which they're needed, like for Binomial Coefficient, is pretty obvious. Every subproblem relies on earlier subproblems only, so we solve them in increasing order.

In fact it relies on all previous subproblems.

Rod cutting: DP Solution

```
// p[1..n], no price for rod with length 0
cut_rod(vector<int> &p, n) {
  vector<int> r(n+1, 0);
  for(int i = 1; i <= n; i++) {
    for(int k = 1; k <= i; i++) {
      r[i] = max(r[i], p[k] + r[i-k]);
    }
  }
  return r[n];
}
```

Runtime analysis

How fast des this solution run?

Runtime analysis

How fast des this solution run? $\mathcal{O}(n^2)$

Conclusion

When is DP useful?

- When you can divide a problem into subproblems.

When is DP useful?

- When you can divide a problem into subproblems.
- When the subproblems overlap.

When is DP useful?

- When you can divide a problem into subproblems.
- When the subproblems overlap.
- For example, it enables you to compute some recursive functions faster, for example Binomial coefficient.

When is DP useful?

- When you can divide a problem into subproblems.
- When the subproblems overlap.
- For example, it enables you to compute some recursive functions faster, for example Binomial coefficient.
- A lot of optimization problems require a dynamic programming solution.

Some remarks about recursion

It is also possible to keep the recursive function and store already stored values, for example in a `map`.

```
map<pair<int,int>,int> m;
int Binom(int n, int k) {
  if(k==0 || k == n) return 1;

  pair<int, int> p = make_pair(n, k);
  if(m[p])
    return m[p];
  return m[p] = Binom(n-1, k-1) + Binom(n-1, k);
}
```

Some remarks about recursion

In cases such as Binomial coefficient it's not necessary to store all previous values. Recursion can also cause further problems (stack limit exceeded). The approach we used, building up the solutions in order, is called "bottom-up", and it is good to get used to it.

How to be good at DP

- DP is hard

How to be good at DP

- DP is hard for most people.

How to be good at DP

- DP is hard for most people.
- The concept is simple, but...

How to be good at DP

- DP is hard for most people.
- The concept is simple, but applying it to a problem and implementing the solution is difficult.

How to be good at DP

- DP is hard for most people.
- The concept is simple, but applying it to a problem and implementing the solution is difficult.
- Always think before you code!

How to be good at DP

- DP is hard for most people.
- The concept is simple, but applying it to a problem and implementing the solution is difficult.
- Always think before you code!
- Most important of all: solve, solve, solve!

What's next:

Solve DP tasks on the grader.