Dynamic Programming

Advanced Algorithms and Data Structures - Lecture 5

Venanzio Capretta
Thursday 29 October 2020

School of Computer Science, University of Nottingham
Repeated Subproblems

- **Divide-and-Conquer:**
  Split the problem into smaller subproblems - solve them recursively

![Diagram of repeated subproblems]
Repeated Subproblems

- **Divide-and-Conquer:**
  Split the problem into smaller subproblems - solve them recursively

![Diagram of repeated subproblems]

- We may hit the same subproblem in different branches.
Repeated Subproblems

- **Divide-and-Conquer:**
  Split the problem into smaller subproblems - solve them recursively

- We may hit the *same subproblem* in different branches
- Divide-and-Conquer would recompute $P_8$ four times
Repeated Subproblems

- **Divide-and-Conquer:**
  Split the problem into smaller subproblems - solve them recursively

  ![Diagram of subproblems](image)

- We may hit the **same subproblem** in different branches
- **Dynamic programming:**
  Remember the solution of $P_8$ after the first time

Divide-and-Conquer would recompute $P_8$ four times
Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

```
<table>
<thead>
<tr>
<th>Problem</th>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The table is a global variable
Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

The table is a global variable
Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

The table is a global variable
Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

The table is a global variable
Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

The table is a global variable
Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

The table is a global variable
Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

The table is a global variable
Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

The table is a global variable
The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price
The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price
We have a table of prices for pieces of different length
You must cut the rod to maximize the total price of the pieces

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>
The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price
We have a table of prices for pieces of different length
You must cut the rod to maximize the total price of the pieces

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>
The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price
We have a table of prices for pieces of different length
You must cut the rod to maximize the total price of the pieces

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

If we split it in $9 = 2 + 4 + 3$, price: $5 + 9 + 8 = 22$
If we don’t split it, price: 24
The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price
We have a table of prices for pieces of different length
You must cut the rod to maximize the total price of the pieces

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>
The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price

We have a table of prices for pieces of different length

You must cut the rod to maximize the total price of the pieces

If we split it in $9 = 2 + 4 + 3$, price: $5 + 9 + 8 = 22$

If we don’t split it, price: 24

If we split it in $9 = 6 + 3$, price: $17 + 8 = 25$ (maximum)

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>
We can adopt a divide-and-conquer strategy:

- First do one cut, you get two smaller rods
- Then apply the algorithm recursively to them
We can adopt a divide-and-conquer strategy:

- First do one cut, you get two smaller rods
- Then apply the algorithm recursively to them

Equations expressing the price $r_n$ of an optimal cut of a rod of length $n$:

\[
\begin{align*}
r_1 &= p_1 \\
r_n &= \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}
\end{align*}
\]

We cut the rod of length $n$ into two rods of length $i$ and $n - i$ in all possible ways (explicitly consider the uncut price $p_n$)
We can improve the algorithm by taking the first cut to be definitive: The first half will not be further cut, so we don’t need a recursive call for it:

\[ r_0 = 0 \]
\[ r_n = \max_{i=1 \ldots n} (p_i + r_{n-i}) \]

This takes care also of

- \( r_1 \) (it automatically gives \( p_1 \))
- the uncut option when \( i = n \)

**Observation:**
Possible improvement: assume that the first cut is the largest:
Cutting 9 = 3 + 6 is equivalent to 9 = 6 + 3
Order of cuts is unimportant: only consider the second one

But we don’t follow this path (exercise: try)
We’ll look at a better algorithm using Dynamic Programming
Naive Algorithm in Haskell

\[
\text{maxCut} :: \mathbb{[Int] \to Int \to Int}
\]
\[
\text{maxCut \ pr \ 0 = 0}
\]
\[
\text{maxCut \ pr \ n = maximum \ [pr!!k + maxCut \ pr \ (n-k) | k \leftarrow [1..n]]}
\]
Naive Algorithm in Haskell

```
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

- **Type**: `maxCut` takes two inputs:
  - `pr :: [Int]`, the list of prices
    - `(pr!!n` is the price of a rod of length `n`)
  - `n :: Int`, the length of the rod we want to cut
Naive Algorithm in Haskell

```
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

- **Type**: `maxCut` takes two inputs:
  - `pr :: [Int]`, the list of prices
    (`pr!!n` is the price of a rod of length `n`)
  - `n :: Int`, the length of the rod we want to cut
- **A rod of length 0** cannot be cut and gives a total price of 0
Naive Algorithm in Haskell

maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]

- **Type**: maxCut takes two inputs:
  - `pr :: [Int]`, the list of prices
    (pr!!n is the price of a rod of length n)
  - `n :: Int`, the length of the rod we want to cut

- A rod of length 0 cannot be cut and gives a total price of 0
- For longer rods, we create the list of all possible prices
  - Make one cut and recursively compute the price of the optimal cut of the remaining part
  - Take the maximum of all possibilities
Naive Algorithm in Haskell

```
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

- **Type**: `maxCut` takes two inputs:
  - `pr :: [Int]`, the list of prices
    - `pr!!n` is the price of a rod of length `n`
  - `n :: Int`, the length of the rod we want to cut

- A rod of **length 0** cannot be cut and gives a total price of 0
- For **longer rods**, we create the **list of all possible prices**
  - Make one cut and recursively compute the price of the optimal cut of the remaining part
  - Take the maximum of all possibilities

**Exercise**: Modify it so it returns the cuts (the list of `ks`)
The Complexity is Exponential: \( T(n) = O(2^n) \)
(See IA for the formal derivation)

**Problem:**
We recompute several times optimal cuts for the same length
Eg when computing \( \text{maxCut pr 9} \), among the possibilities we have
\( 9 = 5 + 4, 9 = 3 + 2 + 4, 9 = 4 + 1 + 4 \) etc
The optimal solution for a rod of length 4 is recomputed each time.

Idea: keep a table with the optimal prices already computed and look up
in it before recomputing.
We construct a global array/table $\text{bestCut}$ that contains the optimal cut for every length:

$$\text{bestCut}[:i] = \text{total price of optimal cut for a rod of length } i$$
We construct a global array/table `bestCut` that contains the optimal cut for every length:

\[ \text{bestCut}[i] = \text{total price of optimal cut for a rod of length } i \]

Two ways of constructing the table:

- **Top-Dow Memoization**: Initialize all entries in the table with a default value \((-\infty)\); implement like Divide-and-Conquer, but always check if the result is already in the table; if it is not, compute it and put it in the table.
DP Solution: Imperative

We construct a global array/table \texttt{bestCut} that contains the optimal cut for every length:

\[ \texttt{bestCut}[i] = \text{total price of optimal cut for a rod of length } i \]

Two ways of constructing the table:

- **Top-Dow Memoization**: Initialize all entries in the table with a default value \((-\infty)\); implement like Divide-and-Conquer, but always check if the result is already in the table; if it is not, compute it and put it in the table

- **Bottom-Up Method**: Systematically compute all the values in the table in order: \texttt{bestCut}[0], \texttt{bestCut}[1], ..., \texttt{bestCut}[n]; when computing \texttt{bestCut}[i], we already know all the previous values are in the table

Bottom-Up is efficient if we know in advance that we need to compute all the values in the table
In Functional Programming:

- **Declarative Style**: We can just define the table of values, without worrying about the order in which it is computed and when values will be available.

- **Lazy Evaluation**: Entries of the table will be computed when needed and they persist for further calls.

```haskell
maxCutD :: [Int] -> Int -> Int
maxCutD pr n = last bestCut
  where bestCut = 0:[ maximum [ pr!!k + bestCut!!(m-k) |
                                k <- [1..m] ] |
                         m <- [1..n] ]
```
Ingredients for Dynamic Programming
Ingredients for Dynamic Programming

- **Optimal Substructure**
  The optimal solution to an instance of the problem (e.g., cutting a rod of length $n$) contains optimal solutions of some subproblems (cutting rods of shorter length).
Ingredients for Dynamic Programming

- **Optimal Substructure**
  The optimal solution to an instance of the problem (e.g., cutting a rod of length \( n \)) contains optimal solutions of some subproblems (cutting rods of shorter length).

- **Overlapping Subproblems**
  Different branches of the computation of an optimal solution require to compute the same subproblem several times.