Julius-Maximilians-UNIVERSITÄT WÜRZBURG

CAIDAS

WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Dynamic Programming

Prof. Dr. Goran Glavaš

30.11.2023

Content

- Divide-and-Conquer
- Dynamic Programming
  - Recursive DP
  - Iterative (Bottom-Up) DP

What are algorithms built from?

- **Building blocks** of algorithms
  - Elementary operations
  - Sequential processing (**one** processing lines)
  - Parallel processing (**multiple** processing lines)
  - Conditions (conditioned execution)
  - Loops (repetition)
  - Subprograms (modular construction of an algorithm)
  - Recursion

Recursion

- Recursive algorithms solve **recursive problems**:
  - **Divisible** into subproblems of the same type as the original problem
  - **Solution to the subproblem** is part of the solution to the whole problem

Factorial (Fakultät) problem

Input: Natural number *n*

(Desired) Output: Factorial $n! = 1 * 2 * \ldots * n$

Iterative solution

```
prod <- 1
for x in [2, 3, ..., n] do
  prod <- prod * x
```

But n! is a recursive problem

```
n! = n * (n-1)!
   = n * (n-1) * (n-2)!
   = ...
   = n * (n-1) * (n-2) * ... * 3 * 2 * 1
```

Dynamic Programming

- **Dynamic programming** solves **problems** with following properties:
  - **Divisible** into subproblems of the same type as the original problem
  - **Solution to the subproblem** is part of the solution to the whole problem

+

  - **Subproblems repeat a lot**: storing solutions of solved subproblems crucial

- Commonly applied to **(discrete) optimization problems**
  - Problems that have many possible solutions
  - Solutions have values or costs associated to them
  - We want to find the **optimal solution** – one with max value or min cost
    - There can be more than one optimal solution!

Example DP Problem: **Rod cutting**

Rod-cutting problem

A company buys **long steel rods** and **cuts** them into **shorter rods** which it then sells. For each rod of length *i* (in inches), we know the corresponding price p_i the company sells it for. Given a rod of length *n* inches and a table of prices p_i for $i = 1, 2, \ldots, n$ determine the **maximal revenue** r_n the company can have from that rod and find a cutting **s** that achieves that maximal revenue.

Length *i*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Price p_i	1	5	8	9	10	17	17	20	24	26

- Note that, in principle, if p_n is large enough, the optimal revenue may be achieved without any cutting at all.

Example DP Problem: **Rod cutting**

- Let's have a toy example: short rod – n = 4
- Even for such a short rod, we have already **8** different potential solutions (with no cutting as one of them)

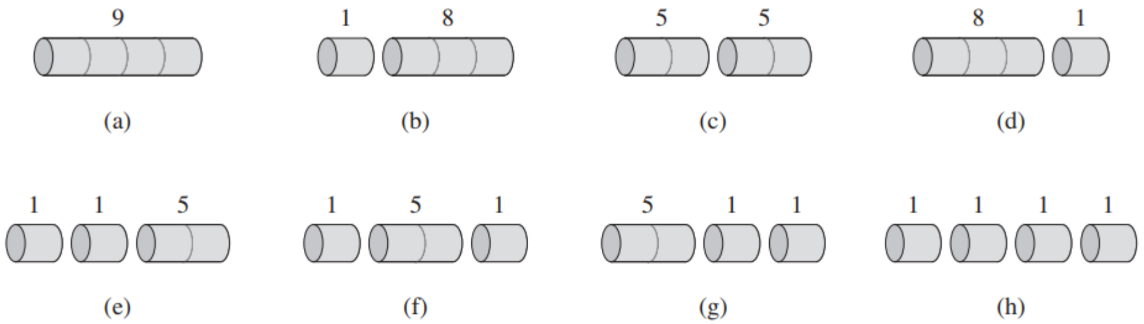


Image from Cormen et al.

- We will denote solution with ordinary additive notation:
  - 4 = 1 + 1 + 2: means that we cut the rod of length 4 inches (left to right) into segments of 1 inch, 1 inch, and 2 inches
  - But 1+1+2, 1+2+1, 2+1+1 all clearly have the same value – if we computed the price for one, we don't need to for the other (**repeating subproblems**)

Example DP Problem: **Rod cutting**

- Assume that an **optimal solution** cuts the rod into *k* pieces
- **General notation** is then:

$n = i_1 + i_2 + i_3 + \ldots + i_k$ (rod pieces of lengths $i_1, \ldots, i_k$ inches)

$r_n = p_{i1} + p_{i2} + \ldots + p_{ik}$ (prices $p_{i1}, \ldots, p_{ik}$ euros for rods of lengths $i_1, \ldots, i_k$ inches)

- **Divide-and-conquer**: the problem naturally decomposable into the subproblems of the same type („definable recursively”)

$$r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2}, \ldots, r_{n-2}+r_2, r_{n-1}+r_1)$$

  - Note that $r_m + r_{n-m} = r_{n-m} + r_m$
  - Note that $r_1 = p_1$ as the 1-inch rod cannot be cut further

Example DP Problem: **Rod cutting**

- We can further **simplify** the formulation of the problem
  - Assume we're making the first cut
  - There are only n posibilities for the first cut: after 1, 2, ..., n inches of length
    - Note that „cutting after n inches" means not cutting the rod at all

  - If the first cut is after *i* inches then the cost $r_i = p_i + r_{n-1}$
    - Assume $r_0 = 0$

  - **Reformulate** the problem as

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Example DP Problem: **Rod cutting**

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

- **Recursive** (divide-and-conquer) solution:

```
cut_rod(p, n) # p is the array with prices
   if n == 0 # termination criterion of recursion
     return 0
    r = -inf
    for i in 1 to n
      q = p[i] + cut_rod(p, n-i) # recursive call
      if q > r
        r = q
    return r
```

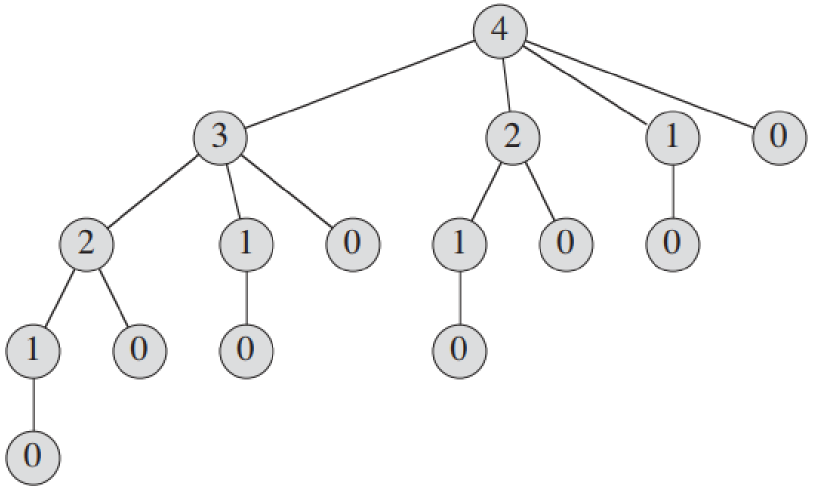
Example DP Problem: **Rod cutting**

- **Q:** What is potentially problematic here?
- **Q:** How long will this run for some large n?
  - How many calls of cut-rod?
- **Repeating subproblems!**
  - cut_rod is called again and again with the same length arguments!

```
cut_rod(p, n)
  if n == 0
    return 0
  r = -inf
  for i in 1 to n
    q = p[i] + cut_rod(p, n-i)
    if q > r
      r = q
  return r
```

Example DP Problem: **Rod cutting**

- Let's plot the **tree of recursive calls** (for n = 4):



- `cut_rod(p, 2)` called **2** times,
- `cut_rod(p, 1)` called **4** times
- `cut_rod(p, 0)` called **8** times

```
cut_rod(p, n)
  if n == 0
    return 0
  r = -inf
  for i in 1 to n
    q = p[i] + cut_rod(p, n-i)
    if q > r
      r = q
  return r
```

- **Q:** runtime T(n)?
  - No. nodes in the tree: 16 = 2^4
  - Can be computed recursively
    $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$
  - Exponential complexity: $O(2^n)$

Content

- Divide-and-Conquer
- Dynamic Programming
  - Recursive (Top-Down) DP
  - Iterative (Bottom-Up) DP

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
   - **Top-down** manner (recursively)
   - **Bottom-up** manner (without recursion)
   - Memorize the solutions to solved subproblems!!!
4. **Construct** an optimal solution
   - Not enough just to know what the optimal (max or min) value is
   - **Rod-cutting**: need to know where to cut the rod ☺! (and not just the maximal price we can achieve for the rod of given length)

Dynamic programming: **solution memorization**

- Avoid the exponential complexity of the recursive solution **$O(2^n)$**
- As usual**,** the **solution** is to **trade some space for time**
- **Memorization**
  - Once the **solution for a subproblem** is computed the first time, **store it**
  - When you encounter **the same subproblem**, simply **retrieve its solution**
  - Obviously, this only helps in case of **repeating subproblems**

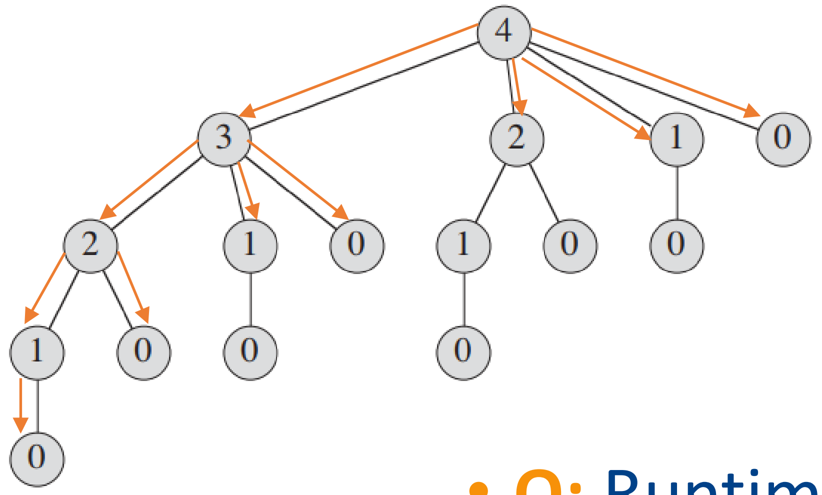
```
cut_rod(p, n)
   r = {}  # empty solutions hashtable
   return cut_rod_rec(p,n,r)

cut_rod_rec(p, n, r)
   if n in r # lookup key in hashtable
     return r[n]

   q = -inf
   if n == 0
     q = 0
   else
     for i in 1 to n
       t = p[i] + cut_rod_rec(p, n-i,r)
       if t > q
         q = t
   r[n] = q # store when computed 1st time
   return r[n]
```

Dynamic programming: **solution memorization**

- **Memorization**
  - Once the **solution for a subproblem** is computed the first time, **store it**
  - When you encounter **the same subproblem**, simply **retrieve its solution**



- **Q:** Runtime?

```
cut_rod(p, n)
   r = {}  # empty solution values hashtable
   return cut_rod_rec(p,n,r)

cut_rod_rec(p, n, r)
   if n in r # lookup key in hashtable
     return r[n]
   q = -inf
   if n == 0
     q = 0
   else
     for i in 1 to n
       t = p[i] + cut_rod_rec(p, n-i, r)
       if t > q
         q = t
   r[n] = q # store when computed 1st time
   return r[n]
```

Dynamic programming: **solution memorization**

- **Memorization**
  - Once the **solution for a subproblem** is computed the first time, **store it**
  - When you encounter **the same subproblem**, simply **retrieve its solution**

- **Top-down DP solution**
  - Recursion + memorization
  - But we can also „**build**” the solution **bottom up** („real” DP)

```
cut_rod(p, n)
    r = {}  # empty solutions hashtable
    return cut_rod_rec(p,n,r)

cut_rod_rec(p, n, r)
    if n in r # lookup key in hashtable
      return r[n]

    q = -inf
    if n == 0
      q = 0
    else
      for i in 1 to n
        t = p[i] + cut_rod_rec(p, n-i)
        if t > q
          q = t
    r[n] = q # store when computed 1st time
    return r
```

Content

- Divide-and-Conquer
- Dynamic Programming
  - Recursive DP
  - Iterative (Bottom-Up) DP

Dynamic programming: **bottom up**

- **Bottom-up DP** – the idea is that of **iterative induction**:
  - We know the solution of the **smallest** subproblem
  - For rod-cutting, this is when $n = 1$
  - If we know r_{k-i} – the solution for **all** $i < k$, then we know how to compute the solution for r_k – the solution for k

Length *i*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Price p_i	1	5	8	9	10	17	17	20	24	26

- $r_1 = p_1 = \mathbf{1}$
- $r_2 = \max(p_1 + r_1, p_2) = \max(1+1, 5) = \mathbf{5}$
- $r_3 = \max(p_1 + r_2, p_2 + r_1, p_3) = \max(1+5, 5+1, 8) = \mathbf{8}$
- $r_4 = \max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4) = \max(1+8, 5+5, 8+1, 9) = \mathbf{10}$
- …

Dynamic programming: **bottom up**

- **Bottom-up DP** – the idea is that of **iterative induction**:
  - We know the solution of the **smallest** subproblem
  - For rod-cutting, this is when $n = 1$
  - If we know r_{k-i} – the solution for **all** $i < k$, then we know how to compute the solution for r_k – the solution for k

```
cut_rod_iter(p, n)
  r = {}
  r[0] = 0
  r[1] = p[1]

  for i in 2 to n
    r[i] = -inf

    # we're looking for the max
    for j in 1 to i
      q = p[j] + r[i-j]
      if q > r[i]
        r[i] = q

  return r[n]
```

- **Q: Runtime** of bottom-up approach?
  - **O(n^2)**

Dynamic programming: **reconstructing solution**

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
   - **Top-down** manner (recursively)
   - **Bottom-up** manner (without recursion)
   - Memorize the solutions to solved subproblems!!!
4. **Construct** an optimal solution
   - Not enough just to know what the optimal (max or min) value is
   - Example – **rod-cutting**: need to know where to cut the rod ☺! (and not just the maximal price we can achieve for the rod of given length)

Dynamic programming: **reconstructing solution**

- So far, we only computed the optimal value, but not the **solution** itself
- **Rod-cutting**: we know the max. price, but not where to cut!
- Both for iterative (bottom-up) and recursive (top-down) approach
  - Need to add additional information for **reconstructing the actual solution**
  - **RC**: store **from which subproblem** (*j*) the maximum for the problem (*i*) came

Top-down (recursive)

```
cut_rod(p, n)
  r = {}
  return cut_rod_rec(p,n,r)


cut_rod_rec(p, n, r)
 if n in r
   return r[n]

 q = -inf
 if n == 0
   t = 0
 else
   for i in 1 to n
     t = p[i] + cut_rod(p, n-i, r)
     if t > q
       q = t
 r[n] = q
 return r
```

Bottom-up (iterative)

```
cut_rod_iter(p, n)
  r = {}
  r[0] = 0
  r[1] = p[1]

  for i in 2 to n
    r[i] = -inf

    # we're looking for the max
    for j in 1 to i
      q = p[j] + s[i-j]
      if q > r[i]
        r[i] = q

  return r
```

Dynamic programming: **reconstructing solution**

- So far, we only computed the optimal value, but not the **solution itself**
- **Rod-cutting**: we know the max. price, but not where to cut!

- Both for iterative (bottom-up) and recursive (top-down) approach
  - Need to add additional information for **reconstructing the actual solution**
  - **RC**: store **from which subproblem** (*j*) the maximum for the problem (*i*) came

Bottom-up (iterative)

```
cut_rod_iter(p, n)
  r = {}
  r[0] = 0
  r[1] = p[1]

  s = {}

  for i in 2 to n
    r[i] = -inf
    for j in 1 to i
      q = p[j] + r[i-j]
      if q > r[i]
        r[i] = q
        s[i] = j

  return r
```

Rod-cutting: complete 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	26

- $r_1 = p_1 = 1$
- $r_2 = \max(p_1 + r_1, p_2) = \max(1+1, 5) = 5$, $s_2 = 2$
- $r_3 = \max(p_1 + r_2, p_2 + r_1, p_3) = \max(1+5, 5+1, 8) = 8$, $s_3 = 3$
- $r_4 = \max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4) = \max(1+8, 5+5, 8+1, 9) = 10$, $s_4 = 2$
- $r_5 = \max(1+10, 5+8, 8+5, 9+1, 10) = 13$, $s_5 = 2$
- $r_6 = \max(1+13, 5+10, 8+8, 9+5, 10+1, 17) = 17$, $s_6 = 6$
- $r_7 = \max(1+16, 5+5, 8+10, 9+8, 10+5, 17+1, 17) = 18$, $s_7 = 3$
- $r_8 = \max(1+18, 5+17, 8+13, 9+10, 10+8, 17+5, 18+1, 20) = 22$, $s_8 = 6$
- $r_9 = \max(1+22, 5+18, 8+17, 9+13, 10+10, 17+8, 17+5, 22+1, 24) = 25$, $s_9 = 3$
- $r_{10} = \max(1+25, 5+22, 8+18, 9+17, 10+13, 17+10, 17+8, 20+5, 24+1, 26) = 27$, $s_{10} = 6$

```
cut_rod_iter(p, n)
  r = {}
  r[0] = 0
  r[1] = p[1]

  s = {}

  for i in 2 to n
    r[i] = -inf
    for j in 1 to i
      q = p[j] + r[i-j]
      if q > r[i]
        r[i] = q
        s[i] = j

  return r, s
```

- **Reconstruction** of the solution: $s_{10} = 6 \rightarrow s_{4(=10-6)} = 2 \rightarrow s_{2(=4-2)} = 2 \rightarrow s_{0(=2-2)}$ **end!**
  - Optimal solution is to cut **twice**, we sell three rods, 6, 2, and 2 inches long.

Rod-cutting: complete 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	26

- $r_1 = p_1 = \mathbf{1}$
- $r_2 = \max(p_1 + r_1, p_2) = \max(1+1, 5) = \mathbf{5}, s_2 = 2$
- $r_3 = \max(p_1 + r_2, p_2 + r_1, p_3) = \max(1+5, 5+1, 8) = \mathbf{8}, s_3 = 3$
- $r_4 = \max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4) = \max(1+8, 5+5, 8+1, 9) = \mathbf{10}, s_4 = 2$
- $r_5 = \max(1+10,5+8,8+5,9+1,10) = \mathbf{13}, s_5 = 2$
- $r_6 = \max(1+13,5+10,8+8,9+5,10+1, 17) = \mathbf{17}, s_6 = 6$
- $r_7 = \max(1+16,5+5,8+10,9+8,10+5,17+1,17) = \mathbf{18}, s_7 = 3$
- $r_8 = \max(1+18,5+17,8+13,9+10,10+8,17+5,18+1,20) = \mathbf{22}, s_8 = 6$
- $r_9 = \max(1+22,5+18,8+17,9+13,10+10,17+8,17+5,22+1,24) = \mathbf{25}, s_9 = 3$
- $r_{10} = \max(1+25,5+22,8+18,9+17,10+13,17+10,17+8,20+5,24+1,26) = \mathbf{27}, s_{10} = 6$

```
get_solution_rod_iter(p, n)
  r, s = cut_rod_iter(p, n)
  print(r[n]) # max price
  i = n
  while i > 0
    print(s[i]) # cuts / lengths
    i = i-s[i]
```

- **Reconstruction** of the solution: $s_{10} = 6 \rightarrow s_{4(=10-6)} = 2 \rightarrow s_{2(=4-2)} = 2 \rightarrow s_{0(=2-2)}$ **end**!
  - Optimal solution is to cut **twice**, we sell three rods, 6, 2, and 2 inches long.

Dynamic programming & AI

- Dynamic programming is used **often** in AI & DS applications

- This is why next time we will solve a couple more problems with dynamic programming (bottom-up)
  - Knapsack Problem
  - Minimal Edit Distance

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟