Lecture 4.
Dynamic programming.

Algorithms and Data Structures
Ivan Solomatin
MIPT

Outline

- Basic principles of DP.
  - Example: Fibonacci numbers
  - Example: Paid stairs
  - Principles of DP
  - Example: Turtle
- Longest Increasing Subsequence (LIS)
  - Problem statement
  - $O(N^2)$ algorithm
  - $O(N\ log\ N)$ algorithm

Basic principles of DP.

- **Example: Fibonacci numbers**
- Example: Paid stairs
- Principles of DP
- Example: Turtle

Example: Fibonacci numbers

Statement

$$F_0 = 0, \qquad F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$$

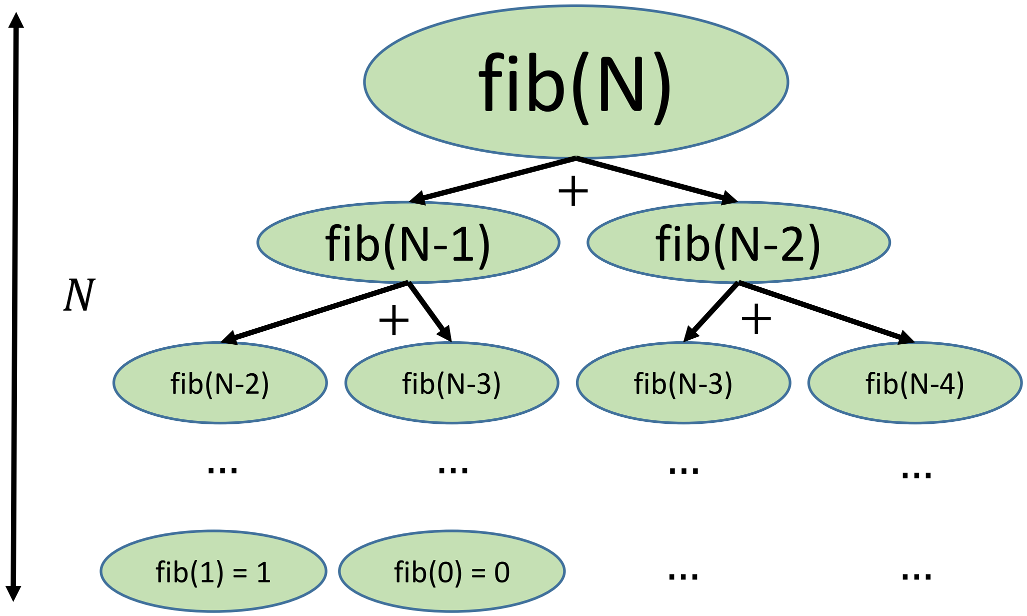
Problem Fibonacci numbers:

Given N, find $F_N \; mod \; 10^9 = ?$

Not to deal with huge numbers, let's calculate $F_N \; mod \; 10^9$ instead of F_N.

Example: Fibonacci numbers

Naïve solution: $O(2^N)$




```python
def fib(n):
    if n < 2:
        return n
    return (fib(n - 1) +
            fib(n - 2)) % 10**9
```

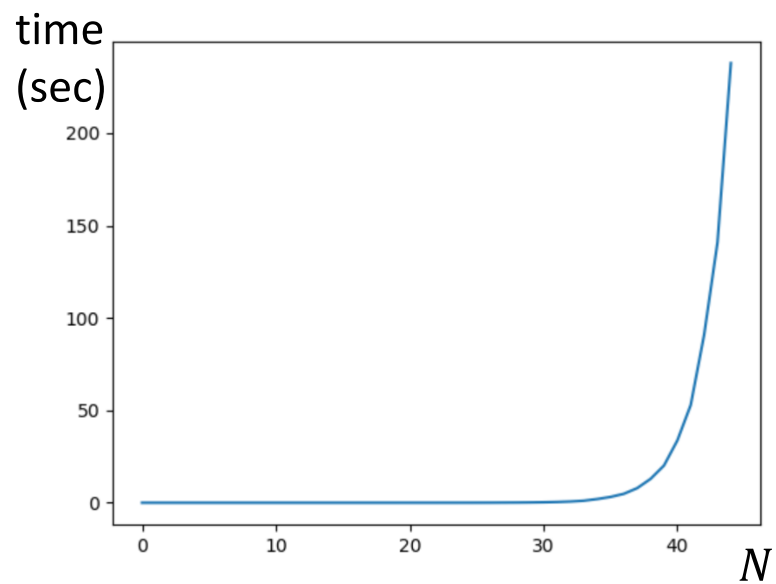
$$O(2^N)$$

$N = 16 : 0.0005$ sec

$N = 32 : 0.67$ sec

$N = 42 : 90$ sec

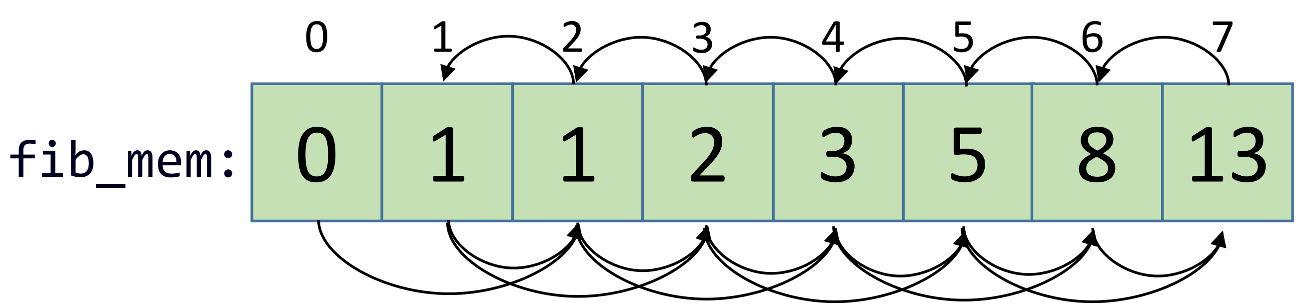
$N = 44 : 237$ sec



Example: Fibonacci numbers

Memoization (top-down): $O(N)$

```
fib_mem = {0: 0, 1: 1}
def fib(n):
    if n not in fib_mem:
        fib_mem[n] = (fib(n - 1) +
                      fib(n - 2)) % 10**9
    return fib_mem[n]
```



$O(N)$

$N = 1000$: 0.0005 sec

$N = 2000$: 0.001 sec

$N = 3000$: 0.0025 sec

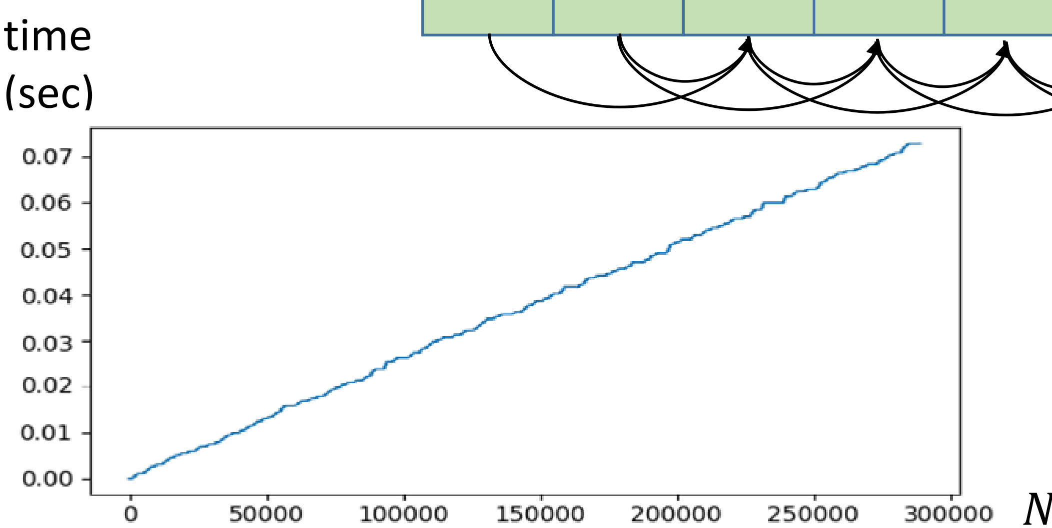
Example: Fibonacci numbers

Tabulation (bottom-up): $O(N)$

```python
def fib(n):
    fib_tab = [None] * max((n + 1), 2)
    fib_tab[0] = 0
    fib_tab[1] = 1
    for i in range(2, n + 1):
        fib_tab[i] = (fib_tab[i - 1] +
                      fib_tab[i - 2]) % 10**9
    return res[n]
```

res:	0	1	1	2	3	5	8	13

$O(N)$



$N = 10000$: 0.002 sec
$N = 100000$: 0.024 sec
$N = 200000$: 0.048 sec
$N = 300000$: 0.073 sec

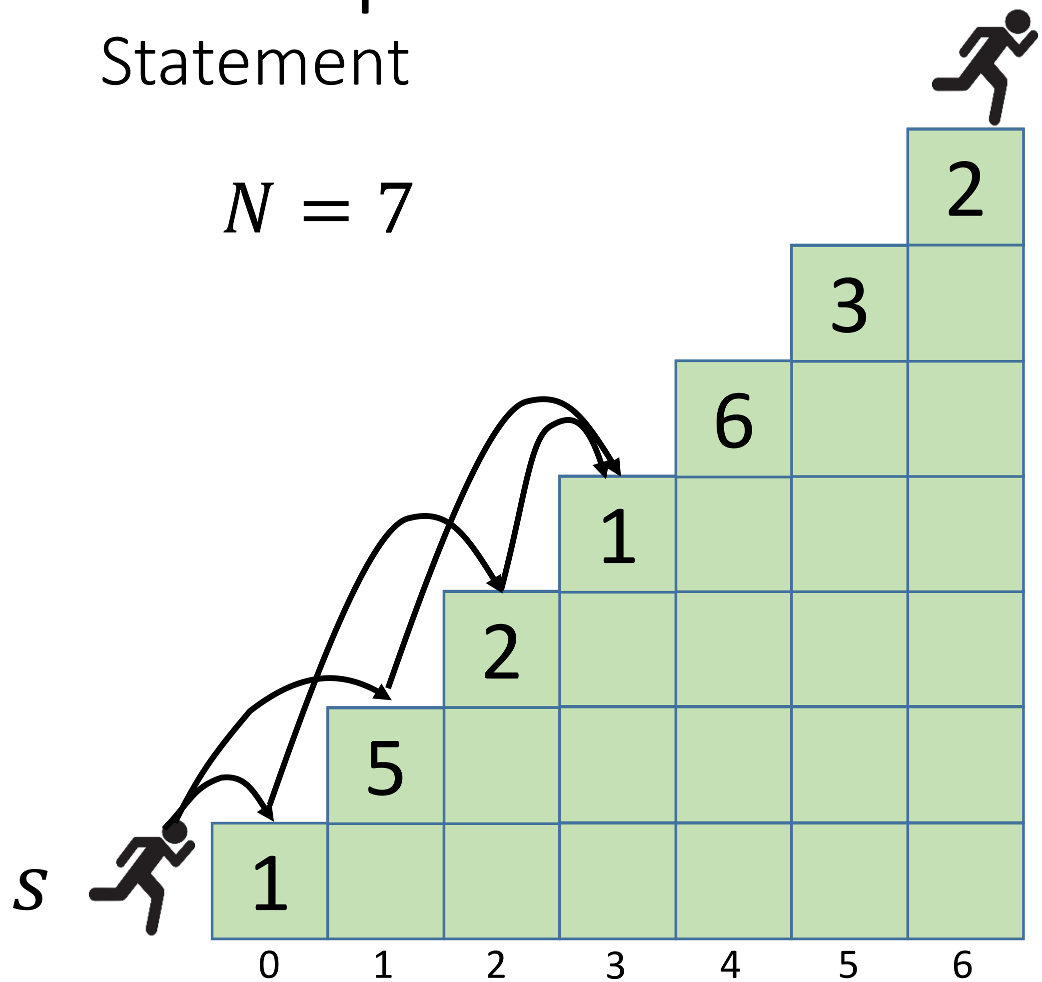
Basic principles of DP.

- Example: Fibonacci numbers
- **Example: Paid stairs**
- Principles of DP
- Example: Turtle

Example: Paid stairs

Statement

$N = 7$



Problem paid stairs:
Given stairs with N steps. To use i-th step you need to pay $s[i]$ coins.

From i-th step you can move to i+1-th, or to i+2-th (skip one step)

$$i \rightarrow \left[\begin{array}{l} i+1 \\ i+2 \end{array}\right.$$

Initially you stay before 0-th step. You need to get to (N-1)-th step and minimize number of spent coins.

Example: Paid stairs

Naïve solution

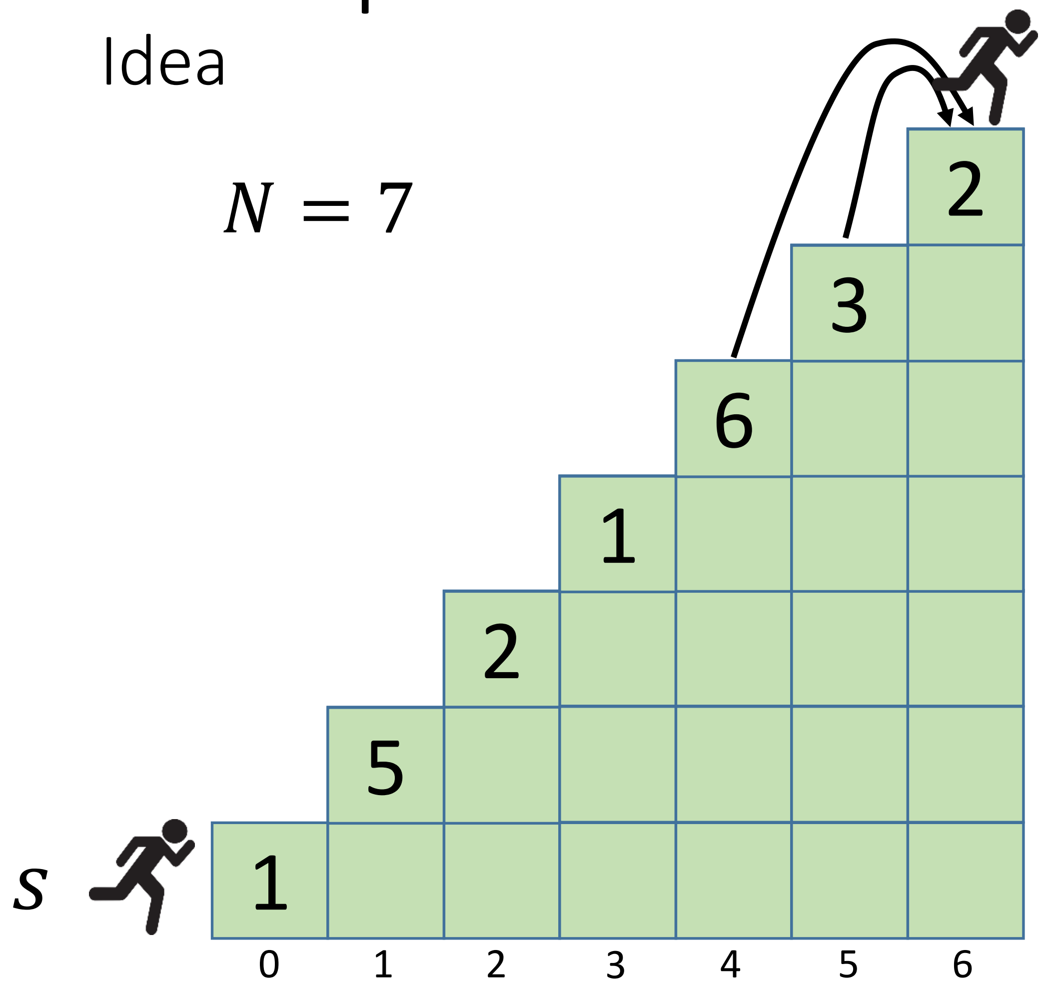
```python
def stairs_rec(s):
    if len(s) <= 2:
        return s[-1]
    d1 = s[0] + stairs_rec(s[1:])
    d2 = s[1] + stairs_rec(s[2:])
    return min(d1, d2)
```

$$O(2^N)$$

Example: Paid stairs

Idea

$N = 7$



$$i \to \begin{bmatrix} i+1 \\ i+2 \end{bmatrix}$$

$$d(N-1) = \begin{bmatrix} d(N-2) + s[N-1] \\ d(N-3) + s[N-1] \end{bmatrix}$$

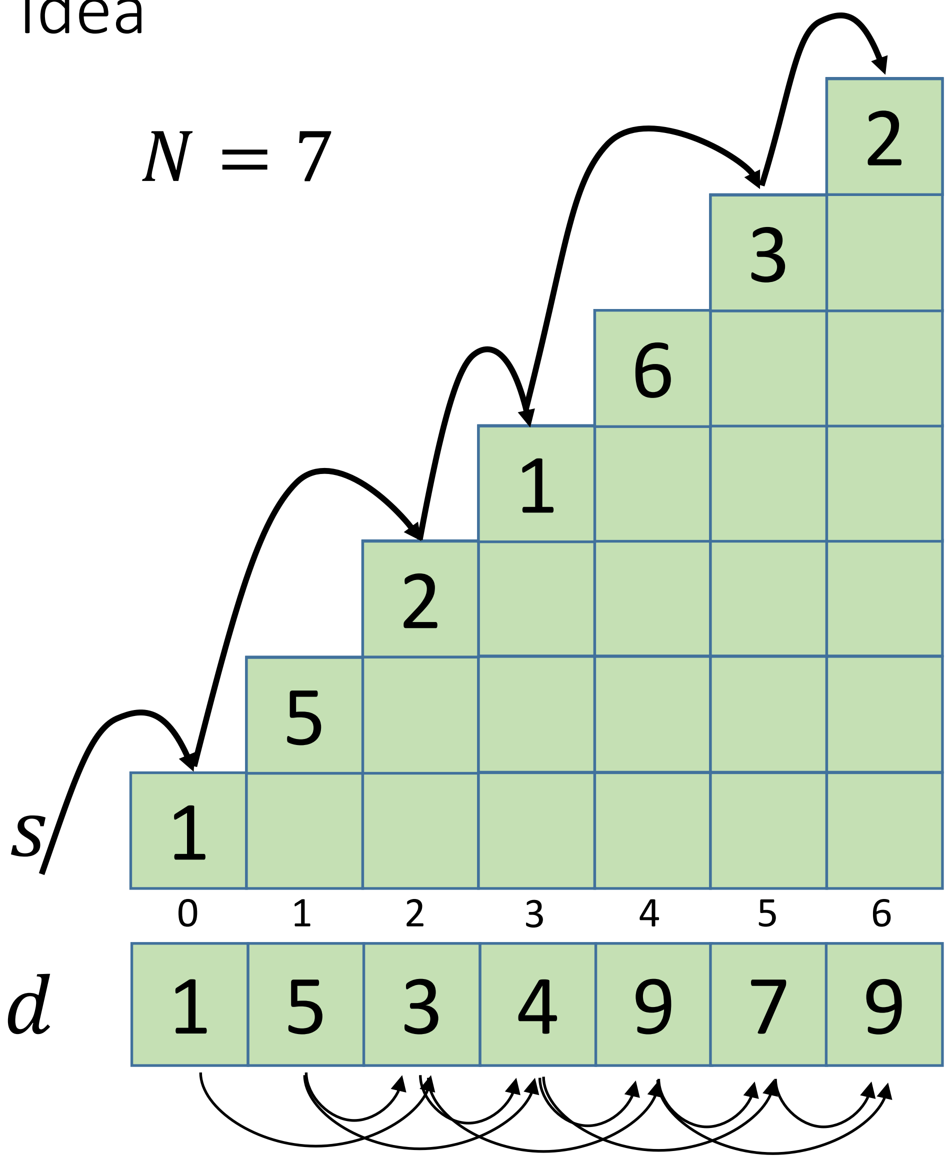
$$d(i) = \min\big(d(i-1), d(i-2)\big) + s[i]$$

$$d(0) = s[0], \qquad d(1) = s[1]$$

Example: Paid stairs

Idea

$N = 7$



$$d[0] = s[0], \qquad d[1] = s[1]$$

$$d[i] = \min\begin{pmatrix} d[i-1], \\ d[i-2] \end{pmatrix} + s[i]$$

$d[N-1]$ – desired answer

Example: Paid stairs

Implementation

```
def solve_stairs(s):
    N = len(s)
    d = [None] * N
    d[0] = s[0]
    d[1] = s[1]
    for i in range(2, N):
        d[i] = min(d[i - 1], d[i - 2]) + s[i]
    return d[N - 1]
```

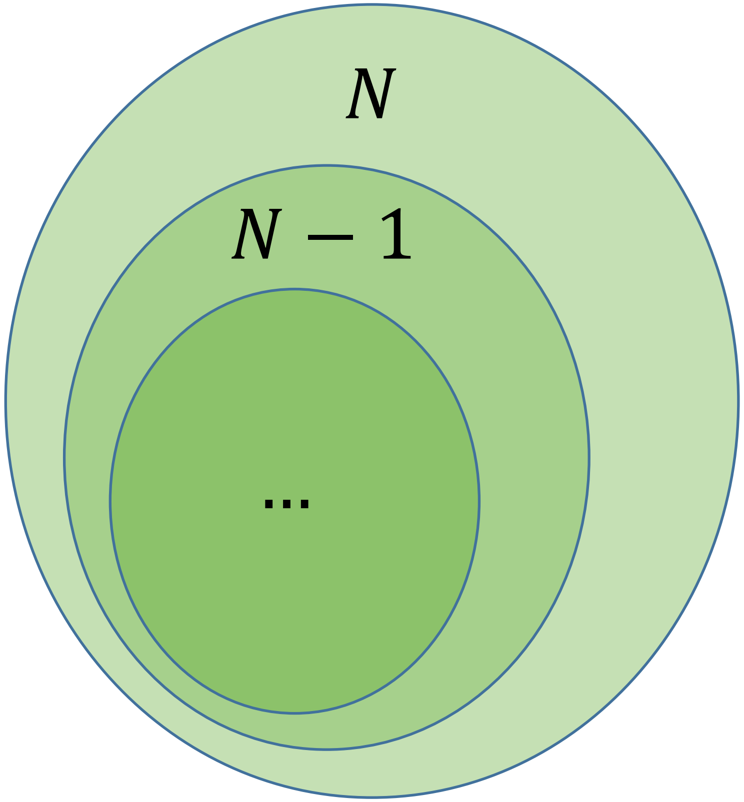
$$O(N)$$

Basic principles of DP.

- Example: Fibonacci numbers
- Example: Paid stairs
- **Principles of DP**
- Example: Turtle

Principles of DP

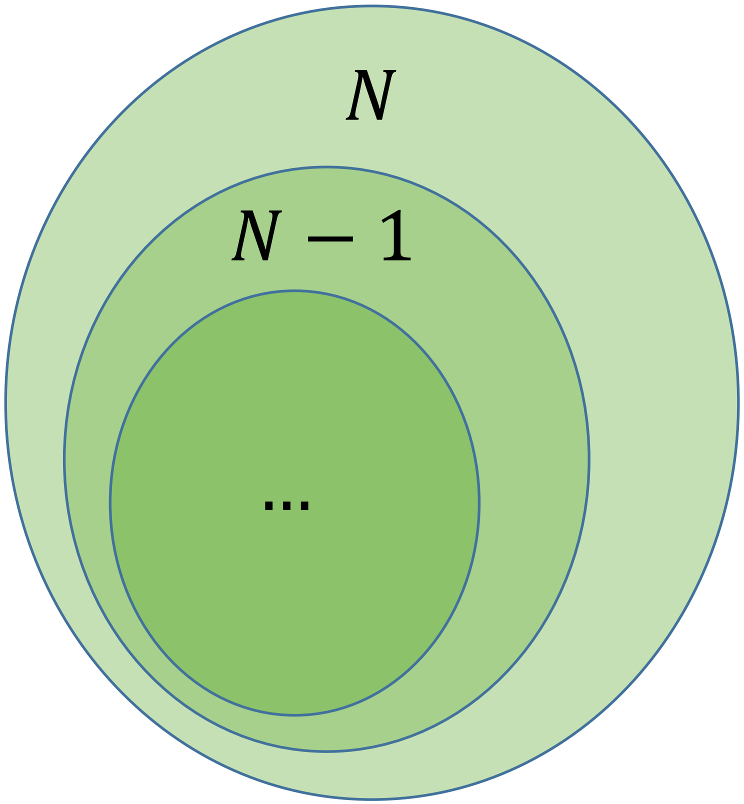
Subproblems and optimal structure



$$F(N-1) \rightarrow$$
$$F(N-2) \rightarrow$$
$$F(N)$$

Principles of DP

Subproblems and optimal structure



1. Overlapping subproblems

Our problem should have overlapping subproblems such that solution of subproblem may be used as subsolution of an initial problem (may be constructed using subsolution).

2. Optimal substructure:

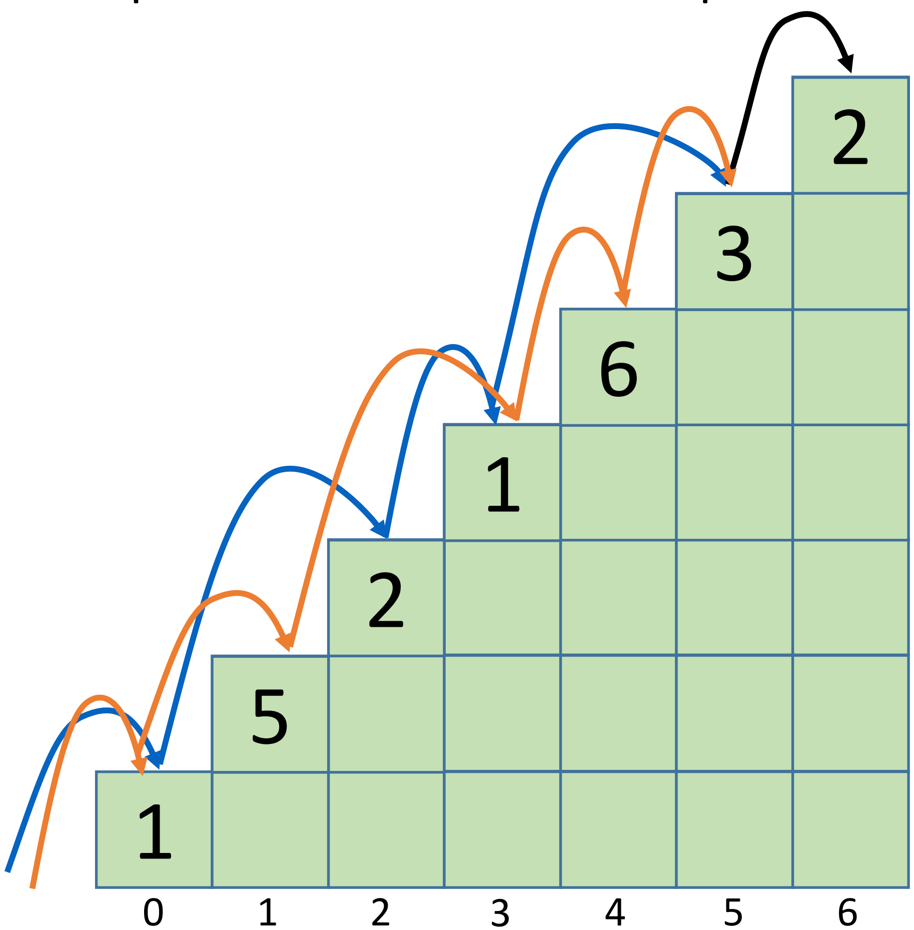
A problem is said to have **optimal substructure** if an optimal solution can be constructed from optimal solutions of its subproblems.

This usually can be proven by contradiction. If solution $F(N)$ is constructed using subproblem solution $F(K)$ (as a subsolution), $F(K)$ must also be optimal.

Let's prove that for paid stairs example

Principles of DP

Optimal structure in paid stairs example



In paid stairs example: Let's suppose that we construct optimal solution $\boldsymbol{F_1(N)}$ from solution $\boldsymbol{F_1(N-1)}$.
Also, let's suppose that there's better solution $\boldsymbol{F_2(N-1)} < \boldsymbol{F_1(N-1)}$, so $\boldsymbol{F_1(N-1)}$ is not optimal.
But this means, we can replace path to $N-1$-th step with $\boldsymbol{F_2(N-1)}$ and obtain solution $\boldsymbol{F_2(N)}$ which is better than $\boldsymbol{F_1(N)}$: $\boldsymbol{F_2(N)} < \boldsymbol{F_1(N)}$.
This means, initial solution $\boldsymbol{F_1(N)}$ was not optimal. Contradiction.

So, this means, if we use solution of subproblem A to construct optimal solution B for it's superproblem, solution A must also be an optimal solution.
This means that this problem has optimal substructure and can be solved using DP.

Principles of DP

Basic steps for DP based solution

1. **Split your problem to subproblems and parametrize these subproblems.**
E.g.: Solutions for parametrized subtasks: $d(i)$. Solution for your task may be obtained from this function: $d(N)$.

2. **Define basis of DP**: solutions for small subproblems which can't be splited deeper, e.g.:

$$d(0) = x_0, \qquad d(1) = x_1$$

3. Prove that problem has optimal substructure and define how we can construct solution from solutions of subproblems (**Inductive step**), e.g::

$$d(i) = F(d(i-1), d(i-2), \dots d(0))$$

4. **Implement algorithm**: Define what values we need to get solution of initial problem. Calculate desired values of d using top-down or bottom-up approach. (Using data structure (list, dict) for storing solutions within whole parametric space of subproblems).

Principles of DP

Basic steps for DP based solution

So, in case of paid stairs:

1. Subproblems:

   $d[i]$ – minimum cost we need to get to i-th step.

2. Basis:

   $$d[0] = s[0], \qquad d[1] = s[1]$$

3. Inductive step:

   $$d[i] = \min(d[i-1], d[i-2]) + s[i]$$

4. Answer for initial problem:

   $$d[N-1]$$

```
d[0] = s[0]
d[1] = s[1]
for i in range(2, N):
    d[i] = min(d[i - 1], d[i - 2]) + s[i]
return d[N - 1]
```

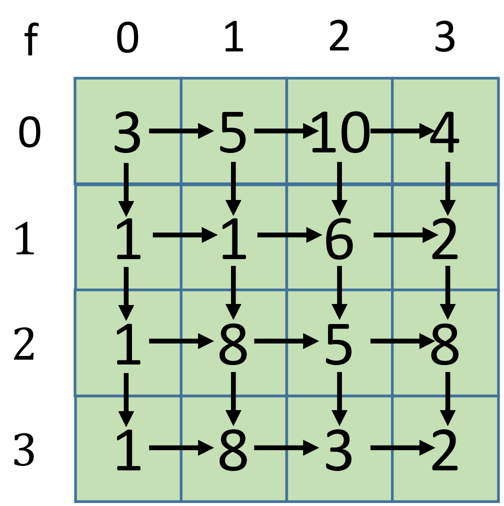
Basic DP problems

- Example: Fibonacci numbers
- Example: Paid stairs
- Principles of DP
- **Example: Turtle**

Example: turtle

Statement

Problem turtle. Turtle stands in the left top corner of a rectangular field $N \times M$. Each cell of field contains given amount of food: $f(i,j)$. Turtle can move to adjacent cell only down or right. Find maximum total amount of food turtle can get on a path from left top corner of a field to the bottom right corner.



Naïve solution:

Check all possible paths.

$$C_{N+M-2}^{N-1} = \frac{(N+M-2)!}{(N-1)!\,(M-1)!}$$

Too long.

$> 10^{10}$ paths for $N = M = 20$

Example: turtle

Solution

f	0	1	2	3
0	3	5	10	4
1	1	1	6	2
2	1	8	5	8
3	1	8	3	2

d	0	1	2	3
0	**3**	**8**	**18**	22
1	4	9	**24**	26
2	5	17	**29**	**37**
3	6	25	32	**39**

1. Subproblems:
   $d[i,j]$ – maximum amount of food turtle can get on path from (0, 0) to (i, j)

2. Basis:

$$d[0,0] = f[0,0]$$
$$d[0,j] = d[0,j-1] + f[0,j]$$
$$d[i,0] = d[i-1,0] + f[i,0]$$

3. Inductive step:

$$d[i,j] = \max(d[i-1,j], d[i,j-1]) + f[i,j]$$

4. Answer for initial problem:

$$d[N-1, M-1]$$

Complexity: $O(NM)$

Longest Increasing Subsequence
(LIS)

- **Problem statement**
- $O(N^2)$ algorithm
- $O(N\ log\ N)$ algorithm

LIS: Problem statement.

Statement

Problem: Longest Increasing Subsequence (LIS).

Given sequence of N elements:

$$x_0, x_1, \dots, x_{N-1}$$

Subsequence is a sequence that can be obtained from initial one by removing elements:

$$x_{i_0}, x_{i_1}, \dots, x_{i_{K-1}}: \quad 0 \le i_0 < i_1 < \ \dots < i_{K-1} < N$$

Increasing subsequence is a subsequence that satisfies inequality of sorted array (strict ascending order):

$$x_{i_0} < x_{i_1} < \cdots < x_{i_{K-1}}$$

Task is to find increasing subsequence of maximum length: $K \to max$.

LIS: Problem statement.

Example

	0	1	2	3	4	5	6	7
x:	3	5	10	1	6	8	9	8
$K = 3$	3	5	10					
$K = 2$				1			9	
$K = 4$	3	5			6			8
$K = 5$	3	5			6	8	9	

Longest Increasing Subsequence
(LIS)

- Problem statement
- $\boldsymbol{O(N^2)}$ **algorithm**
- $O(N\ log\ N)$ algorithm

LIS: $O(N^2)$ algorithm
Idea

Let's use DP:

1. Subproblems:

   $d[i]$ – length of LIS which ends in $x[i]$.

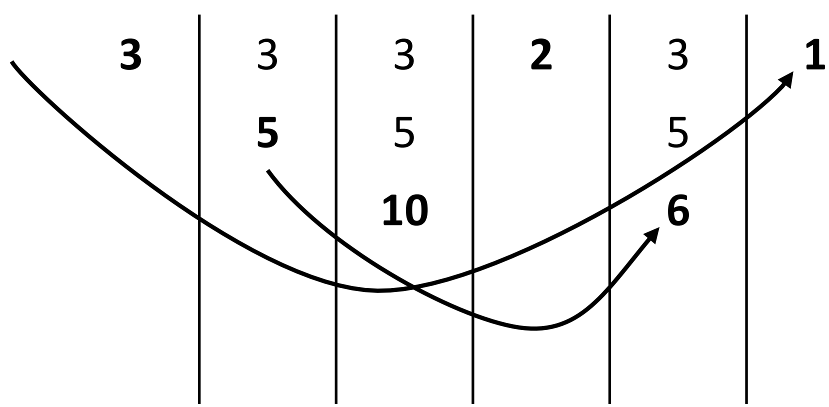
2. Basis:

   $$d[0] = 1$$

LIS: $O(N^2)$ algorithm

Idea

	0	1	2	3	4	5	6	7
x:	3	5	10	2	6	1	9	8
d:	1	2	3	1	3	1		



LIS: $O(N^2)$ algorithm

Idea

Let's use DP:

1. Subproblems:

   $d[i]$ – length of LIS which ends in $x[i]$.

2. Basis:

   $$d[0] = 1$$

3. Inductive step:

   $$d[i] = \begin{cases} d[j] + 1, \mathit{where}\ j = \underset{\substack{0 \le j < i \\ x[j] < x[i]}}{\operatorname{argmax}} d[j], \mathit{if}\ \exists j \\ 1, \qquad \mathit{otherwise} \end{cases}$$

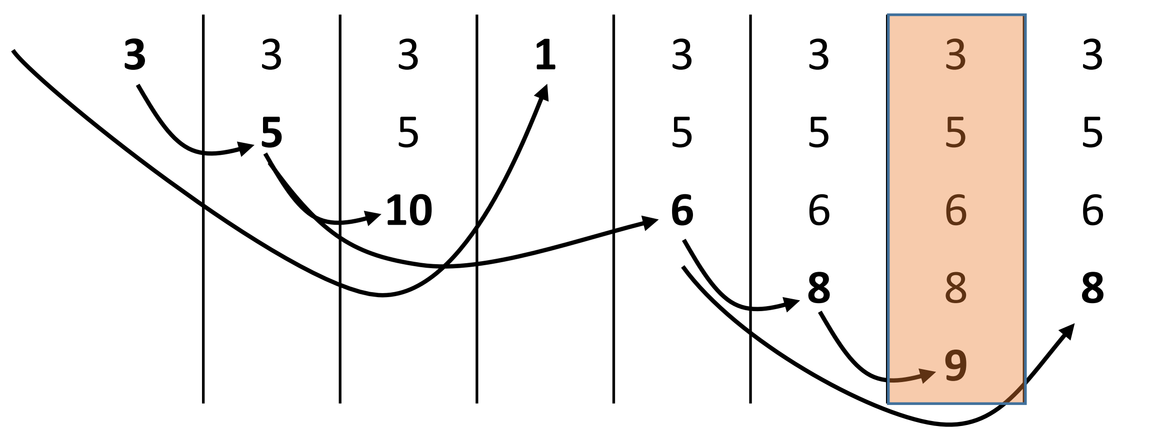
4. Answer for initial problem:

   $$\max_{i: 0 \le i < N} d[i]$$

LIS: $O(N^2)$ algorithm

Example

	0	1	2	3	4	5	6	7
x:	3	5	10	1	6	8	9	8
d:	1	2	3	1	3	4	5	4



LIS: $O(N^2)$ algorithm

Implementation

```
def lis_n2(x):
    N = len(x)
    d = [0] * N
    d[0] = 1
    for i in range(1, N):
        # don't need to find j
        # need just to calculate max_d:
        max_d = 0
        for j in range(i):
            if x[j] < x[i] and d[j] > max_d:
                max_d = d[j]
        d[i] = max_d + 1
    return max(d)
```

$$O(N^2)$$

Longest Increasing Subsequence
(LIS)

- Problem statement
- $O(N^2)$ algorithm
- **$\boldsymbol{O(N\ log\ N)}$ algorithm**

LIS: $O(N\ log\ N)$ algorithm
Idea

1. Subproblems: $d[l][i]$ – minimum last value of increasing subsequence among all increasing subsequences of length l for first i elements of x.

   Set of all increasing subsequences of length l on $x_0, x_1, \dots, x_{i-1}$:

$$J_{i,l} = \{(\mathrm{j}_0, \mathrm{j}_1, \dots \mathrm{j}_{\mathrm{l}-1}):\ 0 \le j_0 < j_1 \dots < j_{l-1} < i, ;\ \ x_{j_0} < x_{j_1} < \cdots < x_{j_{l-1}}\}.$$

$$x_{j_{l-1}} \to \min$$

$$d[l][i] = \begin{cases} \min\limits_{J_{i,l}} x_{j_{l-1}}, & if\ J_{i,l} \neq \emptyset \\ \infty, & if\ J_{i,l} = \emptyset \end{cases}$$

2. Basis:

$$d[0][0] = -\infty, \qquad d[1:][0] = \infty$$

LIS: $O(N \; log \; N)$ algorithm

Idea

	0	1	2	3	4	5	6	7
x:	3	5	10	1	6	0	9	8

$l = 0$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$l = 1$	**3**	3	3	1	1	**0**	0
$l = 2$		5	5	5	5	5	5
$l = 3$			10	10	**6**	6	6
$l = 4$							**9**
$l = 5$							

LIS: $O(N\ log\ N)$ algorithm

Idea

1. Subproblems: $d[l][i]$ – minimum last value of increasing subsequence among all increasing subsequences of length l for first i elements of x.
   Set of all increasing subsequences of length l on $x_0, x_1, \dots, x_{i-1}$:
$$J_{i,l} = \{(\mathrm{j}_0, \mathrm{j}_1, \dots \mathrm{j}_{\mathrm{l}-1}):\ 0 \le j_0 < j_1 \dots < j_{l-1} < i, ;\ \ x_{j_0} < x_{j_1} < \dots < x_{j_{l-1}}\}.$$
$$x_{j_{l-1}} \to \min$$
$$d[l][i] = \begin{cases} \min\limits_{J_{i,l}} x_{j_{l-1}}, & if\ J_{i,l} \ne \emptyset \\ \infty, & if\ J_{i,l} = \emptyset \end{cases}$$
2. Basis:
$$d[0][0] = -\infty, \qquad d[1:][0] = \infty$$
3. Inductive step:
   $(l^* - 1)\ -$ length of longest subsequence which can be extended by $x[i]$:
$$l^* = \min\{l: d[l][i-1] \ge x[i]\}$$
$$d[l][i] = \begin{cases} d[l][i-1], & if\ l \ne l^*, \\ x[i], & if\ l = l^* \end{cases}$$
4. Answer for initial problem:
$$\max_{d[l][N] < \infty} l$$

LIS: $O(N\ log\ N)$ algorithm

Example

	0	1	2	3	4	5	6	7
x:	3	5	10	1	6	8	9	7

$l = 0$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$l = 1$		**3**	3	3	**1**	1	1	0	1
$l = 2$			**5**	5	5	5	5	5	5
$l = 3$				10	10	**6**	6	6	6
$l = 4$							**8**	8	**7**
$l = 5$								**9**	9
	0	1	2	3	4	5	6	7	8

LIS: $O(N\ log\ N)$ algorithm

Remarks

$$d[l][i] = \begin{cases} d[l][i-1], & if\ l \neq l^* + 1, \\ x[i], & if\ l = l^* + 1 \end{cases}$$

$d[l][i]$ differs from $d[l][i-1]$ only in l^*. Also, on i-th step we need only results for $i-1$-th step. So, we can use 1D array $d[l]$ and update it iteratively so that on i-th step $d[l]$ is indeed $d[l][i]$.

$$l^* = \max\{l : d[l][i-1] < x[i]\}$$

Let's notice that $d[l]$ is always monotonous, because new value $x[i]$ is inserted after all values $< x[i]$. So, we can use binary search to find l^*.

LIS: $O(N\ log\ N)$ algorithm

Implementation (pseudocode)

```
def lis(x):
    N ← len(x)
    # value that is > than each value in x:
    d ← [∞] * (N + 1)
    # value that is < than each value in x:
    d[0] ← -∞
    for i in range(N):
        # use binary search (bisect)
        # to find l* here:
        l* ← min{l: d[l] ≥ x[i]}
        d[l*] ← x[i]
    return max{l: d[l] < ∞}
```

$O(N \log N)$

Conclusion

Thank you for watching!