59534809

Artificial Intelligence and Machine Learning (AIML)

2023–24

What is DP?

- Wikipedia definition: “method for solving complex problems by breaking them down into simpler subproblems”
- This definition will make sense once we see some examples
  - Actually, we’ll only see problem solving examples today

Overview of Dynamic Programming

- *Dynamic programming* (DP) is used to solve a wide variety of discrete optimization problems such as scheduling, string-editing, packaging, and inventory management.
- Break problems into subproblems and combine their solutions into solutions to larger problems.
- In contrast to divide-and-conquer, there may be relationships across subproblems.

Steps for Solving DP Problems

1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

- Each step is very important!

Mathematical background: recurrence relations

- **Recurrence relation/recursion**: equational relation for how one value in an indexed collection, is related to any other value(s) in the collection, e.g.:

$$x_{i+1} = 2x_i + 3$$

Mathematical background: recurrence relations

- **Recurrence relation/recursion**: equational relation for how one value in an indexed collection, is related to any other value(s) in the collection, e.g.:

  $$x_{i+1} = 2x_i + 3$$

  with $x_1 = 1$, for all $i \in \mathbb{N}$, refers to collection of equations:

  $x_1 = 1$, $x_2 = 2x_1 + 3$, $x_3 = 2x_2 + 3$, $x_4 = 2x_3 + 3$, and so on

59534809

Mathematical background: recurrence relations

- **Recurrence relation/recursion**: equational relation for how one value in an indexed collection, is related to any other value(s) in the collection, e.g.:

  $$x_{i+1} = 2x_i + 3$$

  with $x_1 = 1$, for all $i \in \mathbb{N}$, refers to collection of equations:

  $x_1 = 1,\ x_2 = 2x_1 + 3,\ x_3 = 2x_2 + 3,\ x_4 = 2x_3 + 3,$ and so on

- Evaluating gets **sequence** $x = 1, 5, 13, 29, \ldots$

Maths background: linear and nonlinear functions

- **Linear function**: special form **preserving** algebraic rules of addition and multiplication, satisfies the following:

(1) $f(x + y) = f(x) + f(y)$

(2) $f(c\,x) = c\,f(x)$

for any two variables x,y and constant c.

Maths background: linear and nonlinear functions

- **Linear function**: special form **preserving** algebraic rules of addition and multiplication, satisfies the following:

  (1) $f(x + y) = f(x) + f(y)$

  (2) $f(c\, x) = c\, f(x)$

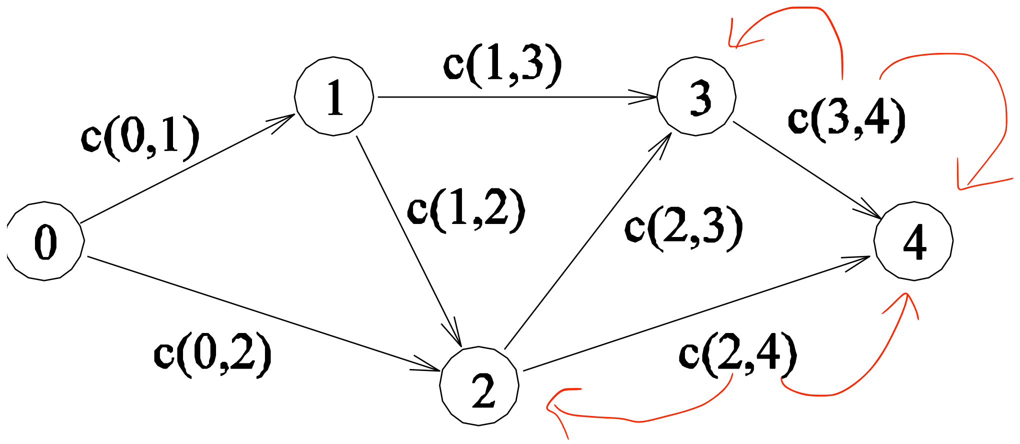
  for any two variables x,y and constant c.

- **Nonlinear function**: does not satisfy both (1) and (2) above

Greedy Algorithms

- Greedy algorithms focus on making the best local choice at each decision point.
- For example, a natural way to compute a shortest path from x to y might be to walk out of x, repeatedly following the cheapest edge until we get to y. WRONG!
- In the absence of a correctness proof greedy algorithms are very likely to fail.

Example



- A graph for which the shortest path between nodes 0 and 4 is to be computed.

$$f(4) = \min\{f(3) + c(3,4), f(2) + c(2,4)\}.$$

Problem:

Let's consider the calculation of **Fibonacci** numbers:

$F(n) = F(n-2) + F(n-1)$

with seed values $F(1) = 1, F(2) = 1$

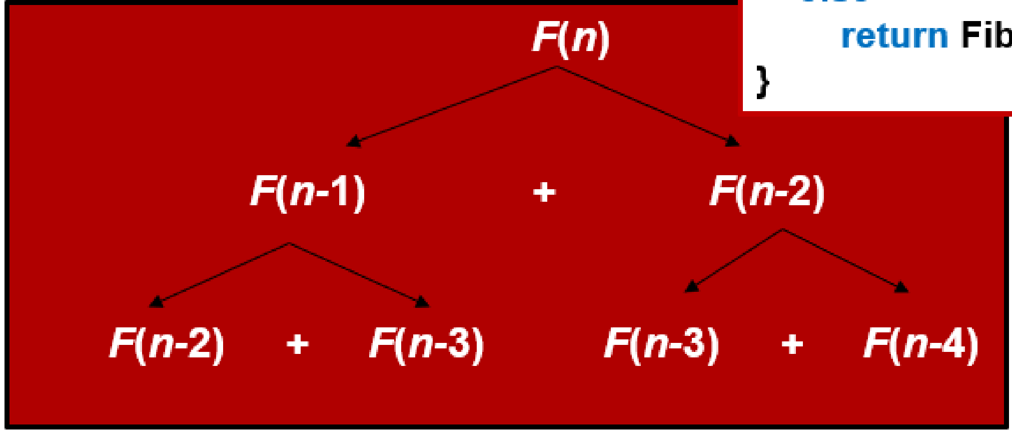
or $F(0) = 0, F(1) = 1$

What would a series look like:

$0, 1, 1, 2, 3, 4, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

- Computing the nth Fibonacci number recursively:
  - F(n) = F(n-1) + F(n-2)
  - F(0) = 0
  - F(1) = 1
  - Top-down approach

```
int Fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return Fib(n - 1) + Fib(n - 2);
}
```



Recursive Algorithm:

```
Fib(n)
{
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

   Return Fib(n-1)+Fib(n-2)
}
```

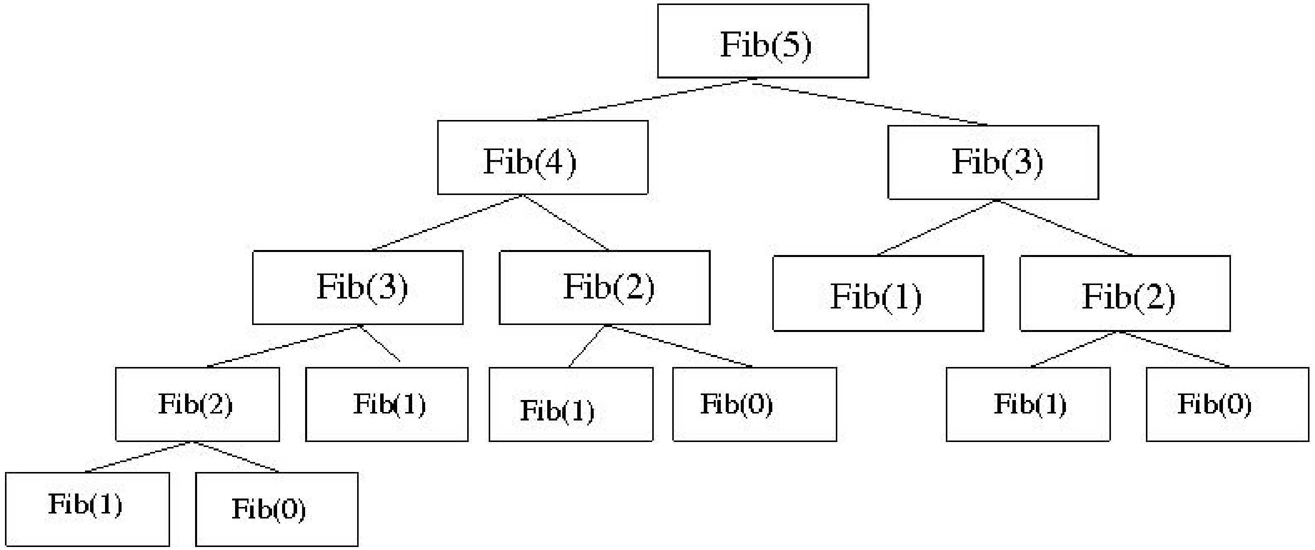
Recursive Algorithm:

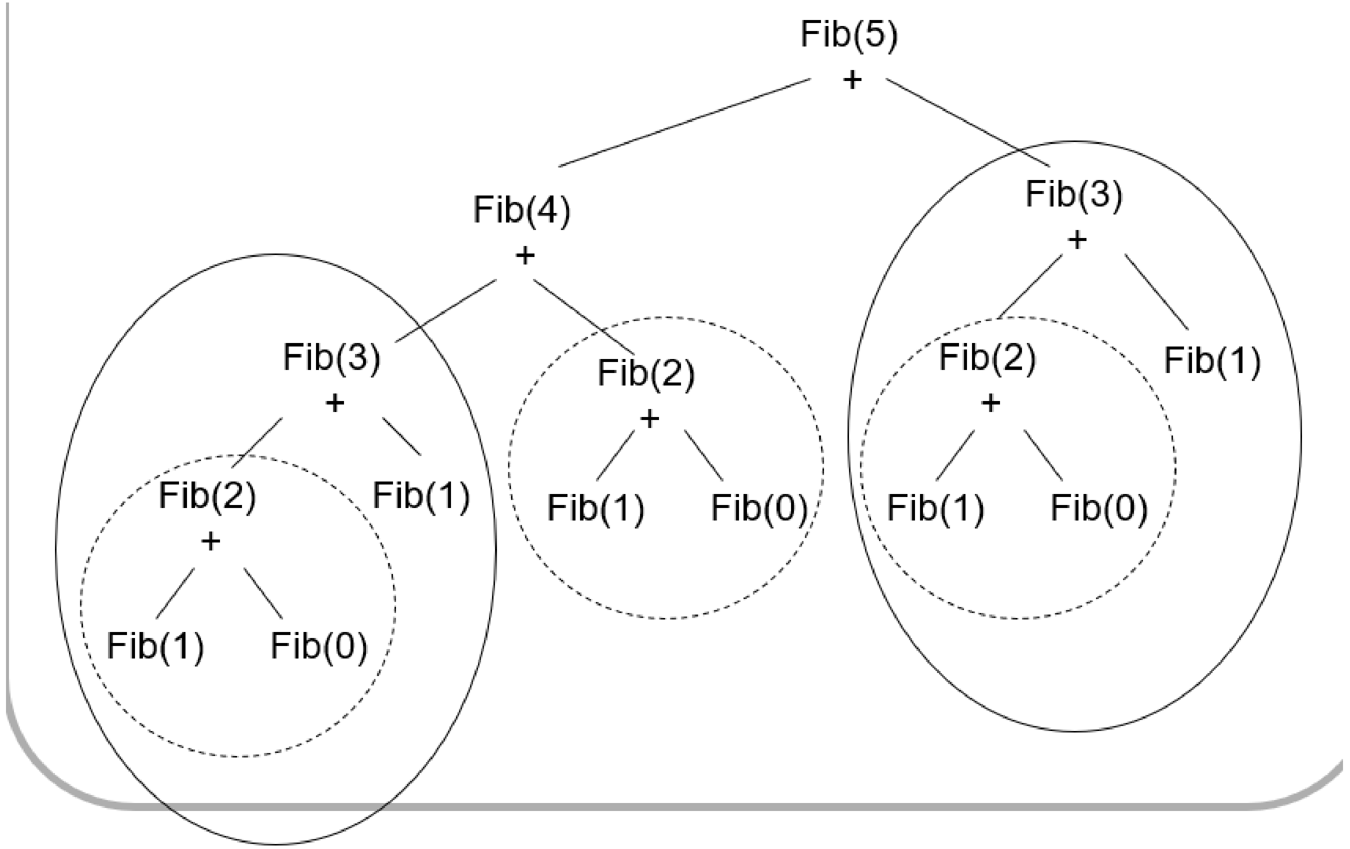
```
Fib(n)
{
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

   Return Fib(n-1)+Fib(n-2)
}
```

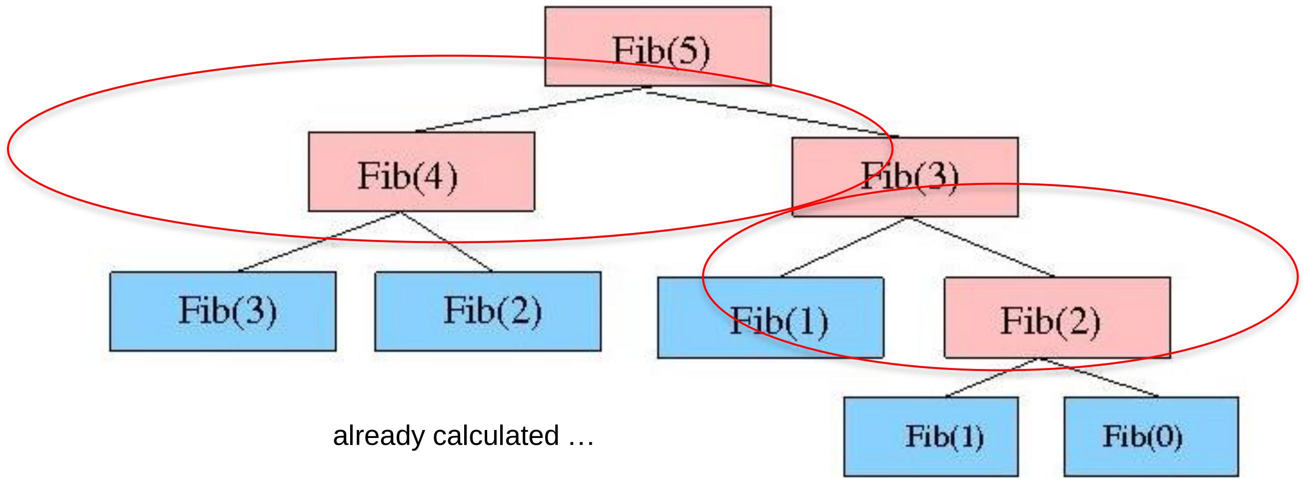
It has a serious issue!

Recursion tree

What's the problem?




Fib(5)
+
Fib(4)
+
Fib(3)
+
Fib(2)
+
Fib(1)
Fib(0)
Fib(1)
Fib(2)
+
Fib(1)
Fib(0)
Fib(3)
+
Fib(2)
+
Fib(1)
Fib(0)
Fib(1)



already calculated …

Memoization:

```
Fib(n)
{
    if (n == 0)
        return M[0];

    if (n == 1)
        return M[1];

    if (Fib(n-2) is not already calculated)
        call Fib(n-2);

    if(Fib(n-1) is not already calculated)
        call Fib(n-1);

    //Store the ${n}^{th}$ Fibonacci no. in memory & use previous results.
    M[n] = M[n-1] + M[n-2]

    Return M[n];
}
```

Dynamic programming

- Main approach: recursive, holds answers to a sub problem in a table, can be used without recomputing.

- Can be formulated both via recursion and saving results in a table (*memoization*). Typically, we first formulate the recursive solution and then turn it into recursion plus dynamic programming via *memoization* or bottom-up.

-”*programming*” as in tabular not programming code

1-dimensional DP Problem

- Problem: given n, find the number of different ways to write n as the sum of 1, 3, 4
- Example: for $n = 5$, the answer is 6

$$\begin{aligned} 5 &= 1+1+1+1+1 \\ &= 1+1+3 \\ &= 1+3+1 \\ &= 3+1+1 \\ &= 1+4 \\ &= 4+1 \end{aligned}$$

1-dimensional DP Problem

59534809

- Define subproblems
  - Let D_n be the number of ways to write n as the sum of 1, 3, 4
- Find the recurrence
  - Consider one possible solution $n = x_1 + x_2 + \cdots + x_m$
  - If $x_m = 1$, the rest of the terms must sum to $n - 1$
  - Thus, the number of sums that end with $x_m = 1$ is equal to D_{n-1}
  - Take other cases into account ($x_m = 3$, $x_m = 4$)

1-dimensional DP Problem

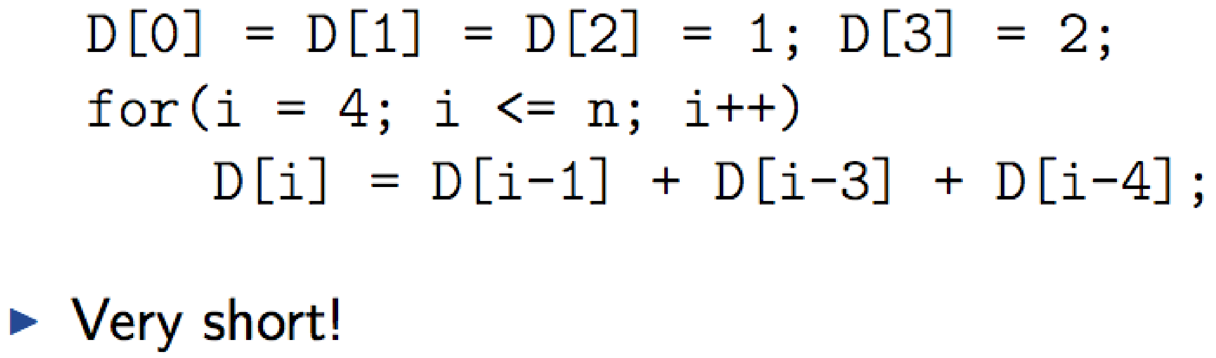
- Recurrence is then

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- Solve the base cases
  - $D_0 = 1$
  - $D_n = 0$ for all negative n
  - Alternatively, can set: $D_0 = D_1 = D_2 = 1$, and $D_3 = 2$

- We're basically done!

1-dimensional DP Problem



```
D[0] = D[1] = D[2] = 1; D[3] = 2;
for(i = 4; i <= n; i++)
    D[i] = D[i-1] + D[i-3] + D[i-4];
```

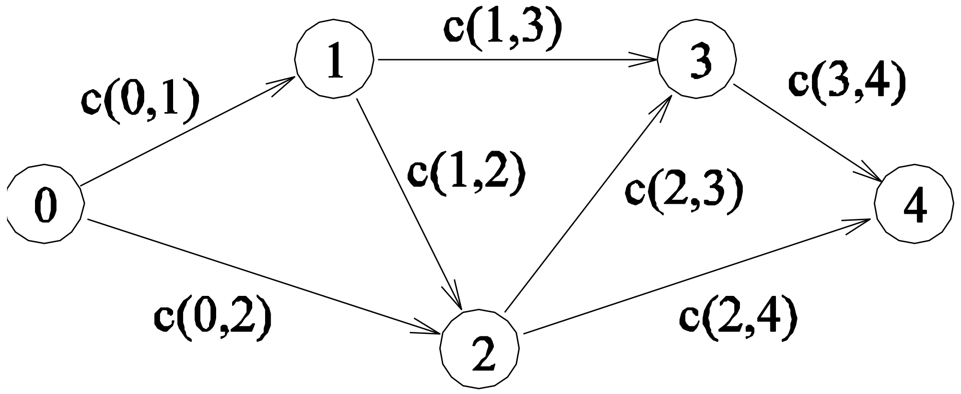
- Very short!

Dynamic Programming: Example

- Consider the problem of finding a shortest path between a pair of vertices in an acyclic graph.
- An edge connecting node *i* to node *j* has cost *c(i,j)*.
- The graph contains *n* nodes numbered *0,1,…, n-1*, and has an edge from node *i* to node *j* only if *i < j*. Node 0 is source and node *n-1* is the destination.
- Let *f(x)* be the cost of the shortest path from node 0 to node *x*.

$$f(x) = \begin{cases} 0 & x = 0 \\ \min\limits_{0 \le j < x} \{f(j) + c(j,x)\} & 1 \le x \le n-1 \end{cases}$$

Dynamic Programming: Example



- A graph for which the shortest path between nodes 0 and 4 is to be computed.

$$f(4) = \min\{f(3) + c(3,4), f(2) + c(2,4)\}.$$

Dynamic Programming

- The solution to a DP problem is typically expressed as a minimum (or maximum) of possible alternate solutions.

- If *r* represents the cost of a solution composed of subproblems $x_1, x_2, \ldots, x_l$, then *r* can be written as

  $$r = g(f(x_1), f(x_2), \ldots, f(x_l)).$$

  Here, *g* is the *composition function*.

- If the optimal solution to each problem is determined by composing optimal solutions to the subproblems and selecting the minimum (or maximum), the formulation is said to be a DP formulation.

Dynamic Programming

- The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution.
- In dynamic programming we usually reduce time by increasing the amount of space
- We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually).
- The table is then used for finding the optimal solution to larger problems.
- Time is saved since each sub-problem is solved only once.

SDP exact: dynamic programming

- Define efficient **Bellman recursion** which solves problem for all possible configurations: uses **SDP factorization**

SDP exact: dynamic programming

- Define efficient **Bellman recursion** which solves problem for all possible configurations: uses **SDP factorization**
- **Principle of optimality**: consequence of **distributivity** e.g. $\min(a + b, a + c) = a + \min(b, c)$; **does not require explicit computation of configurations** (implicitly retains only one optimal configuration)

AIML

SDP exact: dynamic programming

- Define efficient **Bellman recursion** which solves problem for all possible configurations: uses **SDP factorization**
- **Principle of optimality**: consequence of **distributivity** e.g. $\min(a+b, a+c) = a + \min(b, c)$; **does not require explicit computation of configurations** (implicitly retains only one optimal configuration)
- **Efficiency**: reduction to one optimal configuration avoids need to compute all configurations

SDP exact: dynamic programming

- Define efficient **Bellman recursion** which solves problem for all possible configurations: uses **SDP factorization**
- **Principle of optimality**: consequence of **distributivity** e.g. $\min(a+b, a+c) = a + \min(b, c)$; **does not require explicit computation of configurations** (implicitly retains only one optimal configuration)
- **Efficiency**: reduction to one optimal configuration avoids need to compute all configurations
- **Complexity**: typically $O(Nk)$, $O(N^k)$

SDP exact: dynamic programming

- **Applicability**: SDP factorization and principle of optimality required, not necessarily easy to determine when this holds

SDP exact: dynamic programming

- **Applicability**: SDP factorization and principle of optimality required, not necessarily easy to determine when this holds
- **Examples**: bin-packing, piecewise regression, gene sequence alignment, optimal policy iteration (RL), many others …

DP: Bellman equation

- **Optimal objective** value in current stage, in terms of **previous stage's optimal configuration** for n=1,2,...,N:

$$F\left(X_n^{\star}\right) = \min_{X' \in S_{n-1}} F\left(X'\right)$$

DP: Bellman equation

- **Optimal objective** value in current stage, in terms of **previous stage's optimal configuration** for $n=1,2,\ldots,N$:

$$F\left(X_n^\star\right) = \min_{X' \in S_{n-1}} F\left(X'\right)$$

- $S_{n\text{-}1}$ contains **extensions of optimal configuration** in previous stage, $X^*_{n\text{-}1}$.

DP: Bellman equation

- **Optimal objective** value in current stage, in terms of **previous stage's optimal configuration** for $n=1,2,\ldots,N$:

$$F\left(X_n^\star\right) = \min_{X' \in S_{n-1}} F\left(X'\right)$$

- $S_{n\text{-}1}$ contains **extensions of optimal configuration** in previous stage, $X^*_{n\text{-}1}$.
- Recursion for **optimal configurations** themselves:

$$X_n^\star = \arg\min_{X' \in S_{n-1}} F\left(X'\right)$$

DP: Bellman equation

- **Optimal objective** value in current stage, in terms of **previous stage's optimal configuration** for $n=1,2,\ldots,N$:

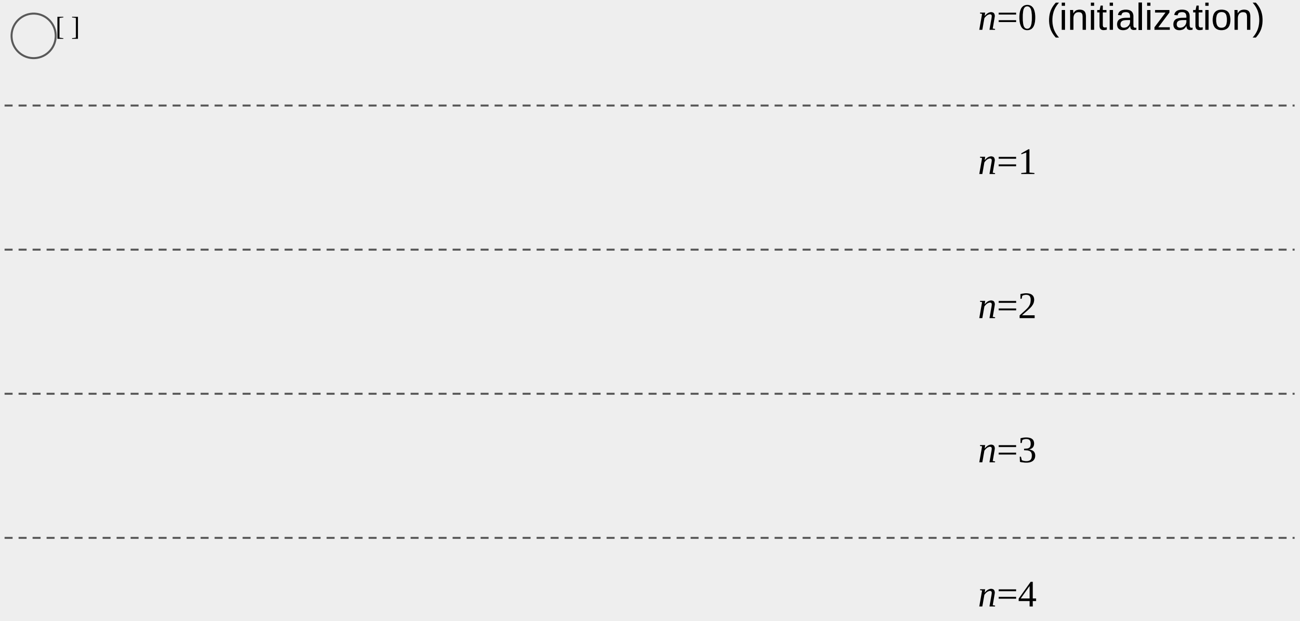
$$F\left(X_n^\star\right) = \min_{X' \in S_{n-1}} F\left(X'\right)$$

- $S_{n\text{-}1}$ contains **extensions of optimal configuration** in previous stage, $X^*_{n\text{-}1}$.
- Recursion for **optimal configurations** themselves:

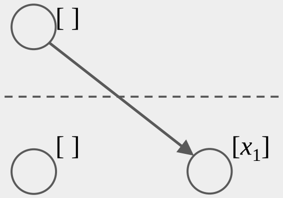
$$X_n^\star = \underset{X' \in S_{n-1}}{\arg\min}\, F\left(X'\right)$$

- Need to **initialize** X^*_0 to start the recursion at $n=0$ (**problem-specific**).

SDP: exhaustive tail subsequences



SDP: exhaustive tail subsequences



n=0 (initialization)

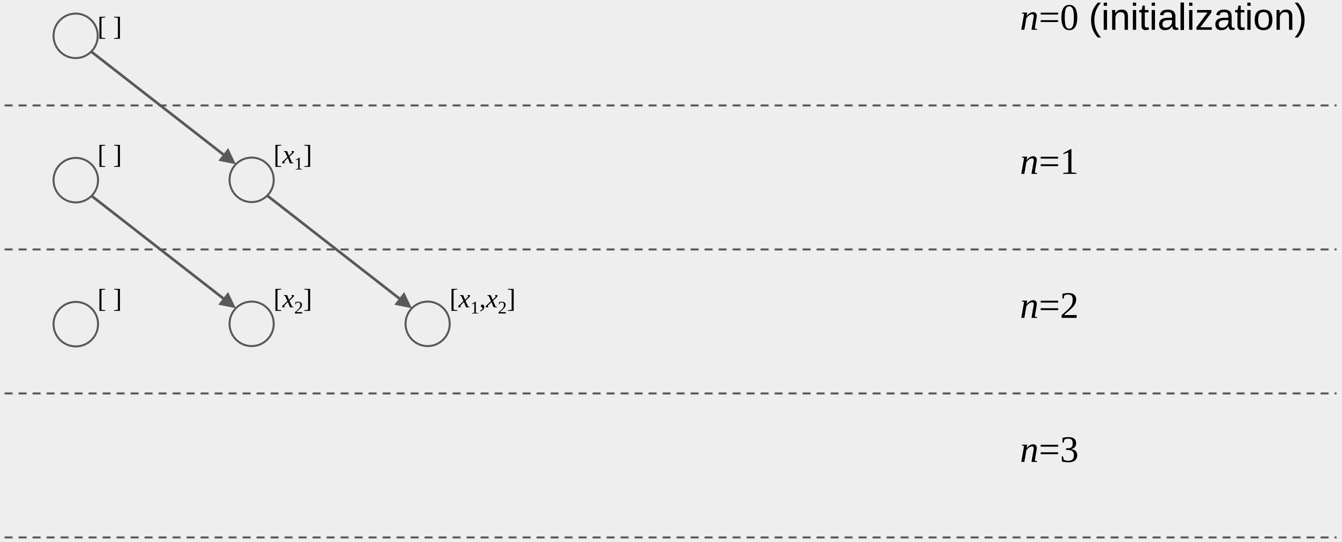
n=1

n=2

n=3

n=4

SDP: exhaustive tail subsequences



n=0 (initialization)

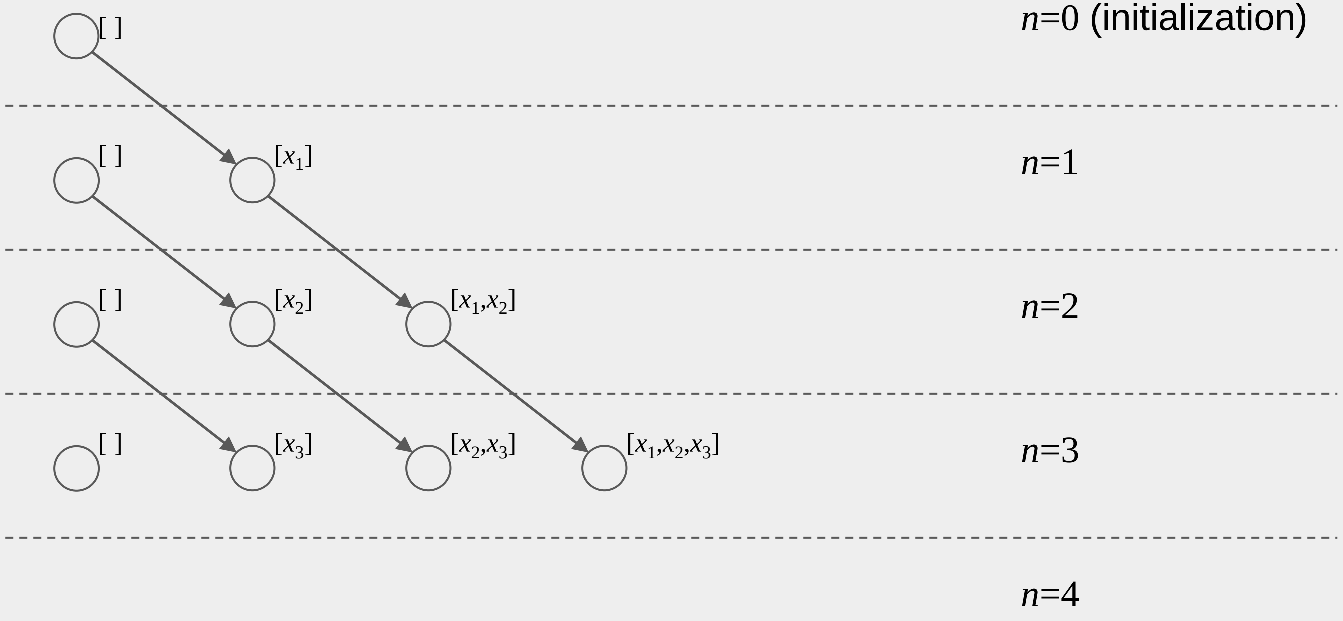
n=1

n=2

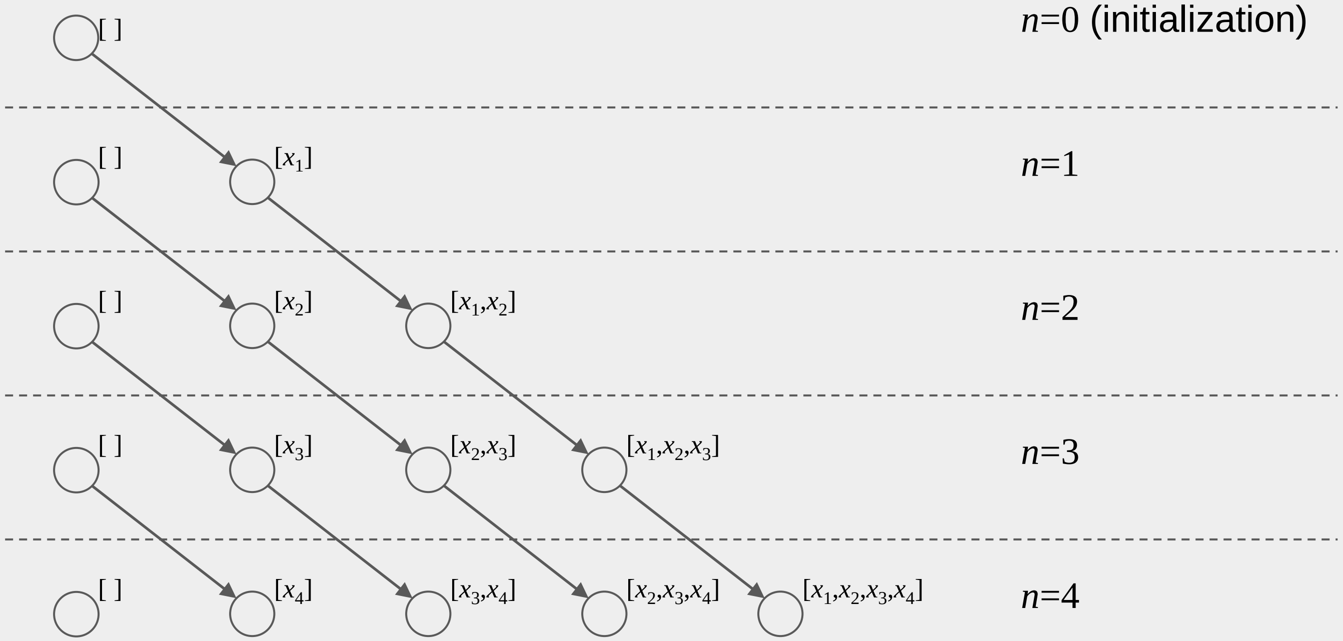
n=3

n=4

SDP: exhaustive tail subsequences



SDP: exhaustive tail subsequences



DP: optimal profit containerized ship loading

- Solve the **maximum sum tail subsequence** optimization problem:

argument of the maximum

$$X^\star = \arg\max_{X' \in \mathcal{X}} \left(\sum_{x' \in X'} x' \right)$$

Summation of x’

X’ belongs X   x’ belongs X’

DP: optimal profit containerized ship loading

- Solve the **maximum sum tail subsequence** optimization problem:

$$X^\star = \underset{X' \in \mathcal{X}}{\arg\max} \left(\sum_{x' \in X'} x' \right)$$

where $\mathcal{X} = \{[x_1, x_2, ..., x_{N-1}, x_N], [x_2, ..., x_{N-1}, x_N], \ldots, [x_{N-1}, x_N], [x_N], []\}$.

DP: optimal profit containerized ship loading

- Solve the **maximum sum tail subsequence** optimization problem:

$$X^{\star} = \underset{X' \in \mathcal{X}}{\arg\max} \left(\sum_{x' \in X'} x' \right)$$

where $\mathcal{X} = \{[x_1, x_2, ..., x_{N-1}, x_N], [x_2, ..., x_{N-1}, x_N], \ldots, [x_{N-1}, x_N], [x_N], []\}$.

- Corresponding DP Bellman recursion

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n).$$

DP: optimal profit containerized ship loading

$P^*_0 = 0$, $X^*_0 = [\ ]$

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n)$$

(initialization, n=0)

x_1=2 (n=1)

x_2=-3 (n=2)

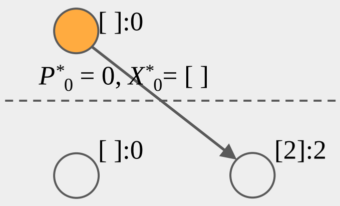
x_3=-4 (n=3)

x_4=6 (n=4)

59534809

DP: optimal profit containerized ship loading

$P^*_0 = 0$
$P^*_n = \max(0, P^*_{n-1} + x_n)$



(initialization, n=0)

x_1=2 (n=1)

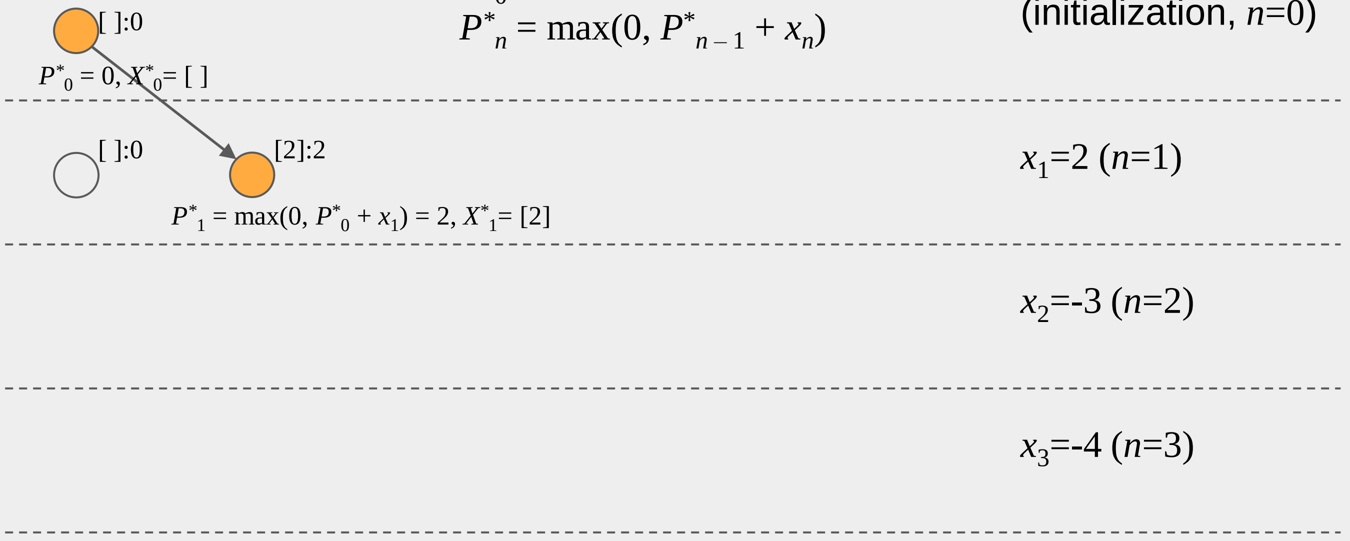
x_2=-3 (n=2)

x_3=-4 (n=3)

x_4=6 (n=4)

DP: optimal profit containerized ship loading

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n)$$


[ ]:0
$P^*_0 = 0, X^*_0 = [\ ]$
[ ]:0
[2]:2
$P^*_1 = \max(0, P^*_0 + x_1) = 2, X^*_1 = [2]$

(initialization, n=0)

x_1=2 (n=1)

x_2=-3 (n=2)

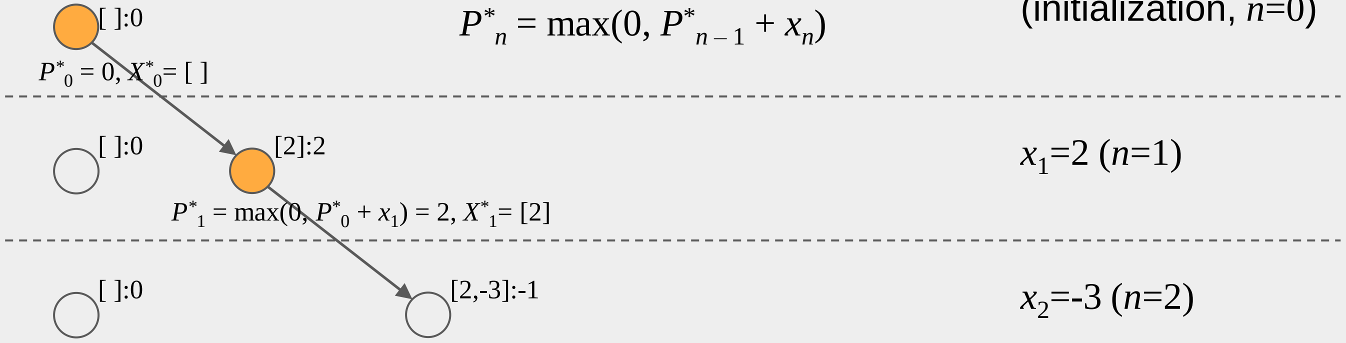
x_3=-4 (n=3)

x_4=6 (n=4)

DP: optimal profit containerized ship loading

$P^*_0 = 0$

$P^*_n = \max(0, P^*_{n-1} + x_n)$


[ ]:0
$P^*_0 = 0, X^*_0 = [\ ]$
[ ]:0
[2]:2
$P^*_1 = \max(0, P^*_0 + x_1) = 2, X^*_1 = [2]$
[ ]:0
[2,-3]:-1

(initialization, n=0)

x_1=2 (n=1)

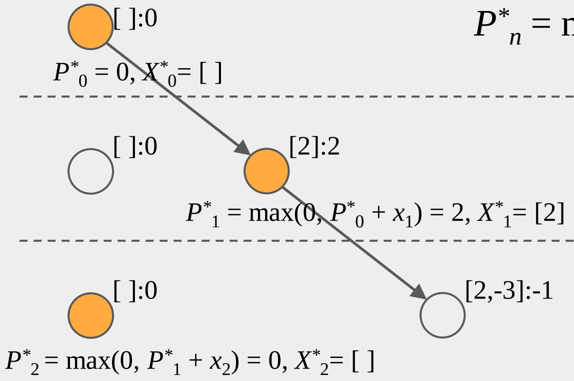
x_2=-3 (n=2)

x_3=-4 (n=3)

x_4=6 (n=4)

DP: optimal profit containerized ship loading

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n)$$


[ ]:0
$P^*_0 = 0, X^*_0 = [\ ]$
[ ]:0
[2]:2
$P^*_1 = \max(0, P^*_0 + x_1) = 2, X^*_1 = [2]$
[ ]:0
[2,-3]:-1
$P^*_2 = \max(0, P^*_1 + x_2) = 0, X^*_2 = [\ ]$

(initialization, n=0)

x_1=2 (n=1)

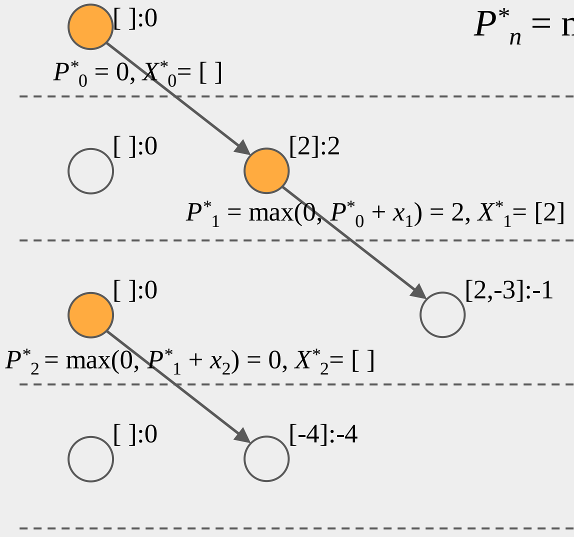
x_2=-3 (n=2)

x_3=-4 (n=3)

x_4=6 (n=4)

DP: optimal profit containerized ship loading

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n)$$



[ ]:0
$P^*_0 = 0, X^*_0 = [\ ]$
[ ]:0
[2]:2
$P^*_1 = \max(0, P^*_0 + x_1) = 2, X^*_1 = [2]$
[ ]:0
[2,-3]:-1
$P^*_2 = \max(0, P^*_1 + x_2) = 0, X^*_2 = [\ ]$
[ ]:0
[-4]:-4

(initialization, n=0)

x_1=2 (n=1)

x_2=-3 (n=2)

x_3=-4 (n=3)

x_4=6 (n=4)

DP: optimal profit containerized ship loading

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n)$$

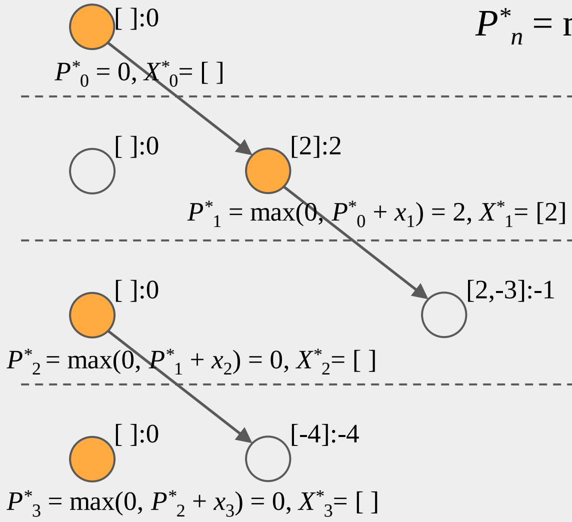
(initialization, n=0)

x_1=2 (n=1)

x_2=-3 (n=2)

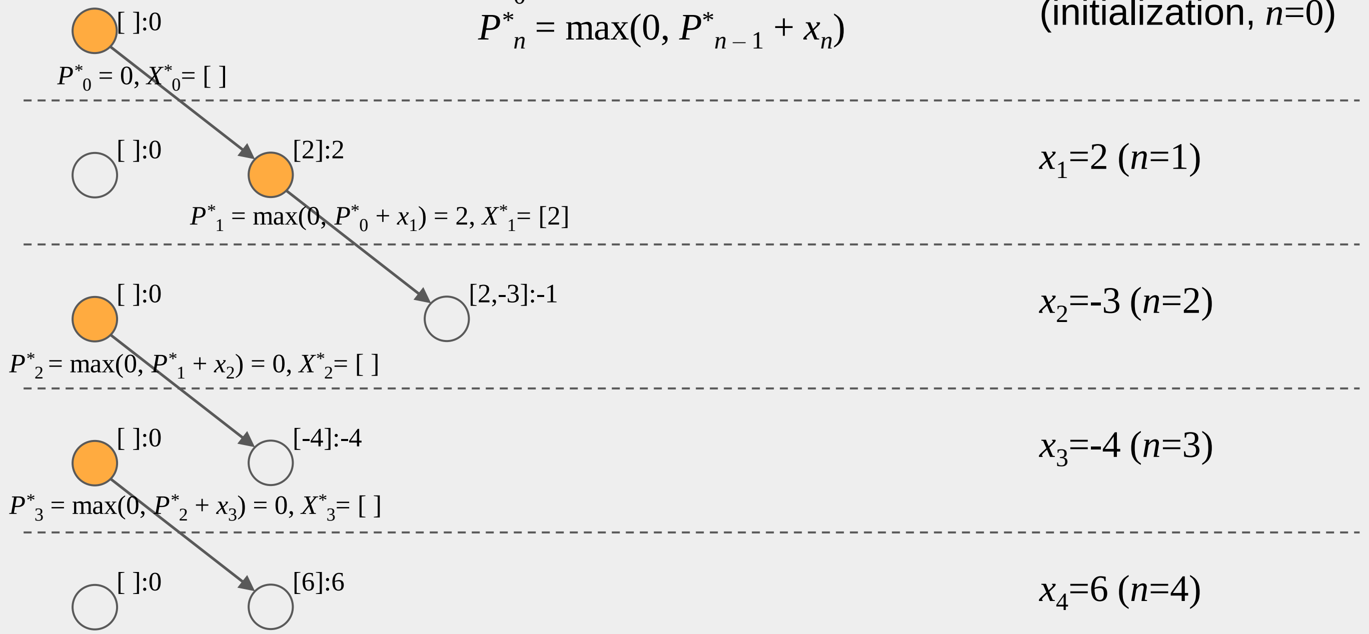
x_3=-4 (n=3)

x_4=6 (n=4)


[ ]:0
$P^*_0 = 0, X^*_0$= [ ]
[ ]:0
[2]:2
$P^*_1 = \max(0, P^*_0 + x_1) = 2, X^*_1$= [2]
[ ]:0
[2,-3]:-1
$P^*_2 = \max(0, P^*_1 + x_2) = 0, X^*_2$= [ ]
[ ]:0
[-4]:-4
$P^*_3 = \max(0, P^*_2 + x_3) = 0, X^*_3$= [ ]

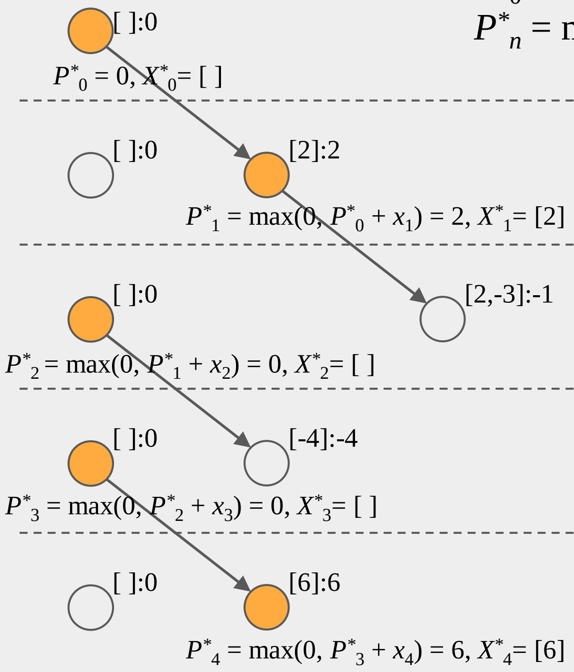
DP: optimal profit containerized ship loading

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n)$$


[ ]:0
$P^*_0 = 0, X^*_0 = [\ ]$
(initialization, n=0)
[ ]:0
[2]:2
$P^*_1 = \max(0, P^*_0 + x_1) = 2, X^*_1 = [2]$
x_1=2 (n=1)
[ ]:0
[2,-3]:-1
$P^*_2 = \max(0, P^*_1 + x_2) = 0, X^*_2 = [\ ]$
x_2=-3 (n=2)
[ ]:0
[-4]:-4
$P^*_3 = \max(0, P^*_2 + x_3) = 0, X^*_3 = [\ ]$
x_3=-4 (n=3)
[ ]:0
[6]:6
x_4=6 (n=4)

DP: optimal profit containerized ship loading

$$P^*_0 = 0$$
$$P^*_n = \max(0, P^*_{n-1} + x_n)$$



[ ]:0
$P^*_0 = 0, X^*_0 = [\ ]$
[ ]:0
[2]:2
$P^*_1 = \max(0, P^*_0 + x_1) = 2, X^*_1 = [2]$
[ ]:0
[2,-3]:-1
$P^*_2 = \max(0, P^*_1 + x_2) = 0, X^*_2 = [\ ]$
[ ]:0
[-4]:-4
$P^*_3 = \max(0, P^*_2 + x_3) = 0, X^*_3 = [\ ]$
[ ]:0
[6]:6
$P^*_4 = \max(0, P^*_3 + x_4) = 6, X^*_4 = [6]$

(initialization, $n=0$)

$x_1=2$ ($n=1$)

$x_2=-3$ ($n=2$)

$x_3=-4$ ($n=3$)

$x_4=6$ ($n=4$)

Exact SDP methods: summary

Method	**Exhaustive**	**Greedy**	**Dynamic programming**
Applicability	Always	Matroid/ greedoid	Optimality principle
Typical complexity	$O(k^N)$, $O(N!)$	$O(Nk)$, $O(N^k)$	$O(Nk)$, $O(N^k)$

References and further reading

- **CLRS**, Chapter 14