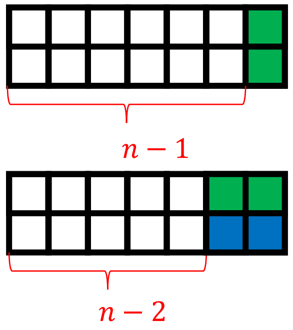
CSE 421 Winter 2025
Lecture 13: DP2

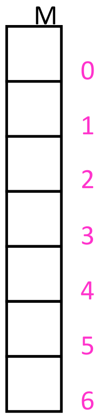
Nathan Brunelle

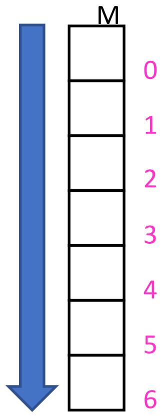
http://www.cs.uw.edu/421

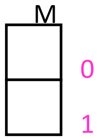
Four Steps to Dynamic Programming



Conclusion: a 1-dimensional memory of size n







1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a "Bottom-up" (iterative) algorithm
   - Without this step: a "Top-down" (recursive) algorithm
4. See if there's a way to save space
   - Is it possible to reuse some memory locations?

Top-Down DP Idea

```
def myDPalgo(problem):
    if mem[problem] not blank:    // Check if we've solved this already
        return mem[problem]
    if baseCase(problem):    // Check if this is a base case
        solution = solve(problem)
        mem[problem] = solution    // Always save your solution before returning
        return solution
    for subproblem of problem:
        subsolutions.append(myDPalgo(subproblem)) // solve each subproblem
    solution = selectAndExtend(subsolutions) // Pick the subproblem to use
    mem[problem] = solution    // Always save your solution before returning
    return solution
```

Bottom-Up DP Idea

```
def myDPalgo(problem):
    for each baseCase:    // Identify which subproblems are base cases
        solution = solve(baseCase)
        mem[baseCase] = solution   // Save the solution for reuse
    for each subproblem in bottom-up order:
        // The order should be chosen so that every subsolution is
        // guaranteed to already be in memory when it's needed
        solution = selectAndExtend(subsolutions)
        mem[subproblem] = solution // Save the solution for reuse
    return mem[problem]
```

Weighted Interval Scheduling

Input: Like interval scheduling each request $\boldsymbol{i}$ has start and finish times $\boldsymbol{s_i}$ and $\boldsymbol{f_i}$.
Each request $\boldsymbol{i}$ also has an associated value or weight $\boldsymbol{v_i}$

$\boldsymbol{v_i}$ might be

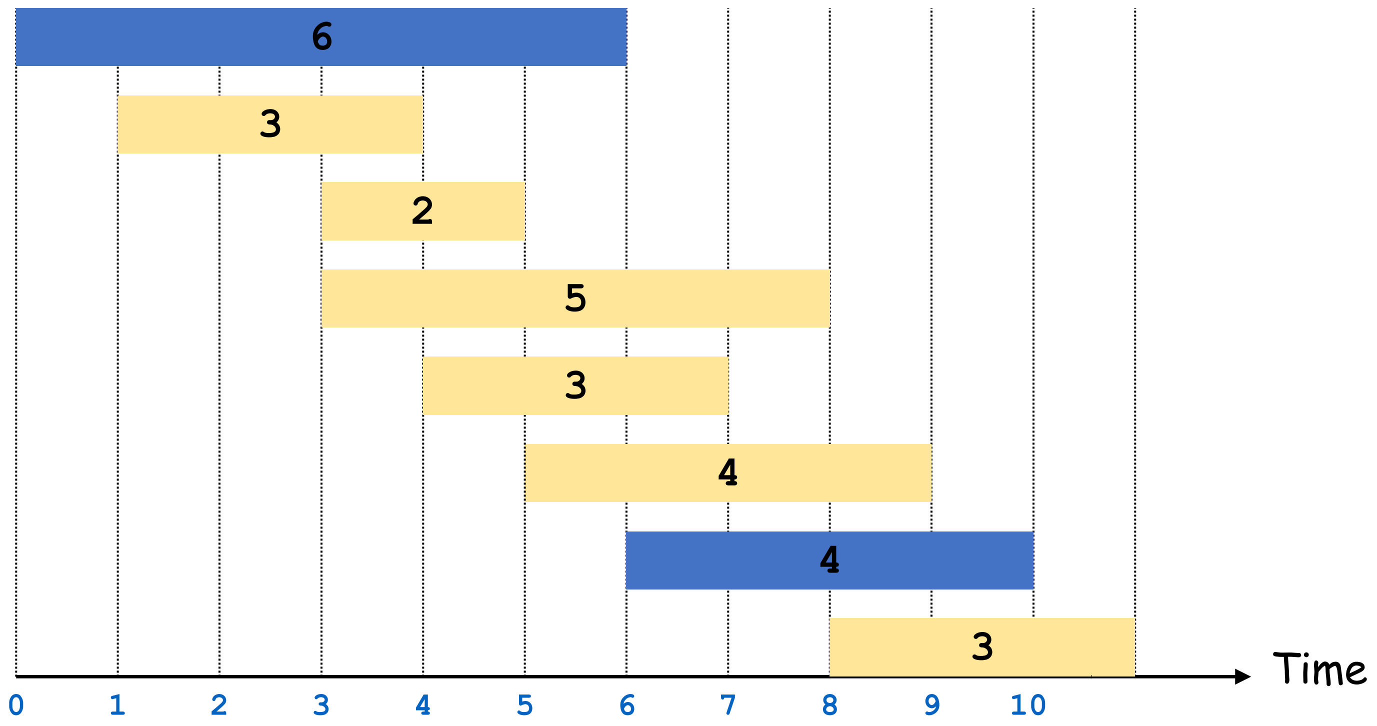
- the amount of money we get from renting out the resource
- the amount of time the resource is being used ($\boldsymbol{v_i = f_i - s_i}$)

Find: A maximum-weight compatible subset of requests.

Weighted Interval Scheduling

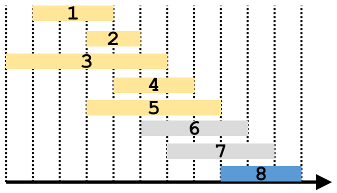
Input: Set of jobs with start times, finish times, and weights

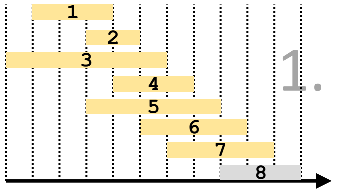
Goal: Find maximum weight subset of mutually compatible jobs.



Optimal yields 10

Weighted Interval Scheduling – Four Steps





$$\max\{OPT(p(j)) + v_j, \mathrm{OPT}(j-1)\}$$

j	OPT[j]
0	0
1	
2	
3	
4	
5	
6	
7	
8	

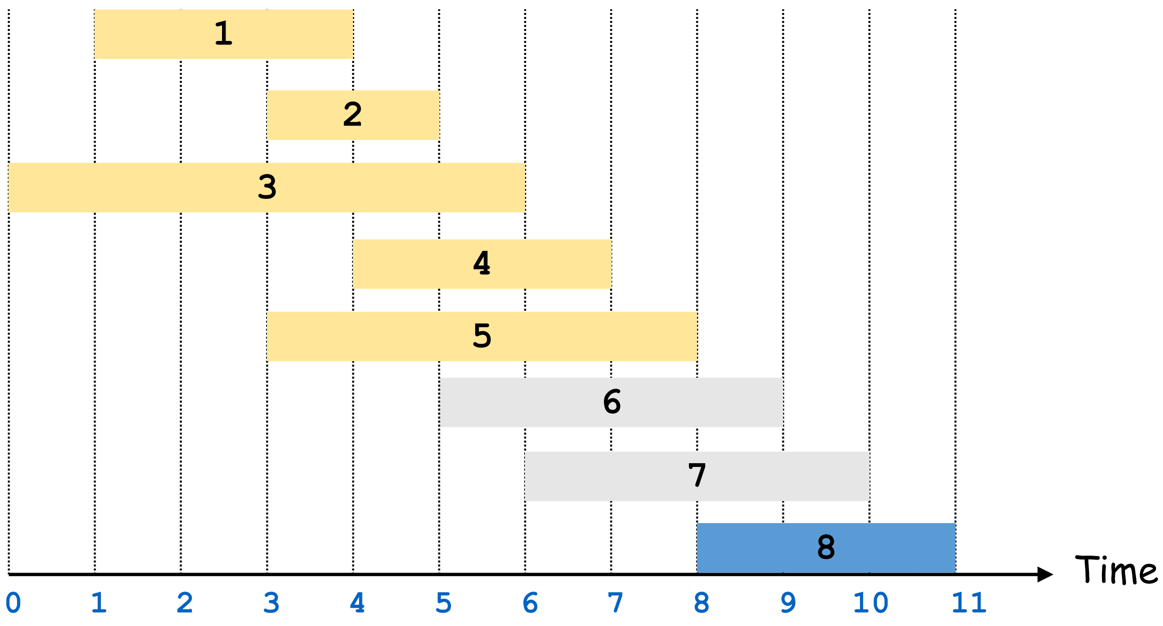
1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a "Bottom-up" (iterative) algorithm
   - Without this step: a "Top-down" (recursive) algorithm
4. See if there's a way to save space
   - Is it possible to reuse some memory locations?

Weighted Interval Scheduling Top-Down DP

```
WIS(j):
    if OPT[j] not blank:    // Check if we've solved this already
        return OPT[j]
    if j==0:    // Check if this is a base case
        mem[j] = 0    // Always save your solution before returning
        return mem[j]
    includej = WIS(p(j)) // Solve each subproblem
    excludej = WIS(j -1) // Solve each subproblem
    solution = max(includej+value[j], excludej) // Pick the subproblem to use
    mem[j] = solution // Always save your solution before returning
    return solution
```

Towards Dynamic Programming: Step 1 – Recursive Algorithm

Option 1: Include the last request

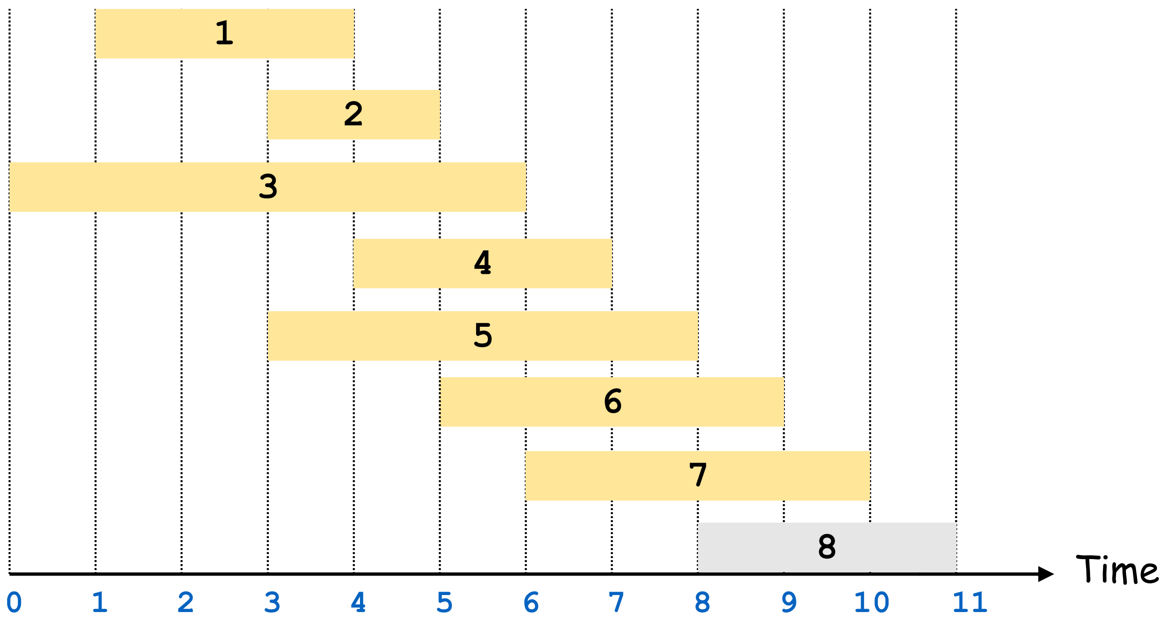


After making this choice, the best solution possible is given by:

- The value of the solution to subproblem consisting of everything compatible
- Plus the value of the last request

$$OPT\big(p(j)\big) + v_j$$

Option 2: Exclude the last request



After making this choice, the best solution possible is given by:

- The value of the solution to subproblem consisting of everything except the last request

$$OPT(j-1)$$

$$\boxed{OPT(j) = \max\{OPT\big(p(j)\big) + v_j, OPT(j-1)\}}$$

Towards Dynamic Programming: Step 2 – Memory Structure

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$

Subproblems are identified by a single parameter

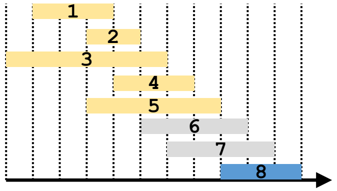
1-dimensional array

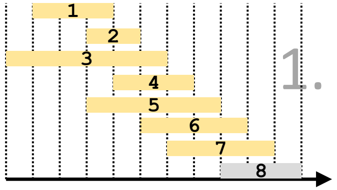
That parameter is the last-ending compatible request

length is the number of requests

j	OPT[j]
0	0
1	
2	
3	
4	
5	
6	
7	
8	

Weighted Interval Scheduling – Four Steps



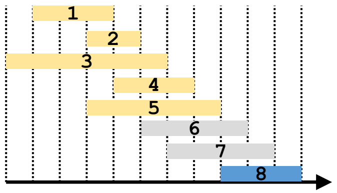


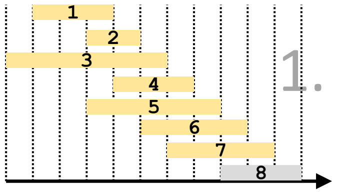
$$\max\{OPT(p(j)) + v_j, \mathrm{OPT}(j-1)\}$$

j	OPT[j]
0	0
1	
2	
3	
4	
5	
6	
7	
8	

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Weighted Interval Scheduling – Four Steps





$$\max\{OPT(p(j)) + v_j, \mathrm{OPT}(j-1)\}$$

j	OPT[j]
0	0
1	
2	
3	
4	
5	
6	
7	
8	

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Towards Dynamic Programming: Step 3 – Order of Evaluation

$$\boxed{OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}}$$

For any given cell j, which other cells might I need?

- $j - 1$
- $p(j)$

It's hard to know in advance what $p(j)$ might be, but certainly $p(j) < j$

Order: increasing order of j will work

j	**OPT[*j*]**
0	0
1	
2	
3	
4	
5	
6	
7	
8	

Bottom-Up DP Idea

```
def myDPalgo(problem):
    for each baseCase:    // Identify which subproblems are base cases
        solution = solve(baseCase)
        mem[baseCase] = solution   // Save the solution for reuse
    for each subproblem in bottom-up order:
        // The order should be chosen so that every subsolution is
        // guaranteed to already be in memory when it's needed
        solution = selectAndExtend(subsolutions)
        mem[subproblem] = solution // Save the solution for reuse
    return mem[problem]
```

Weighted Interval Scheduling Bottom-Up DP

WIS(j):

OPT[0] = 0 // Save the solution for the base case

for each $i = 1$ up to j:

// The order should be chosen so that every subsolution is

// guaranteed to already be in memory when it's needed

solution = max(OPT[$p(i)$]+value[i], OPT[$i - 1$])

mem[i] = solution // Save the solution for reuse

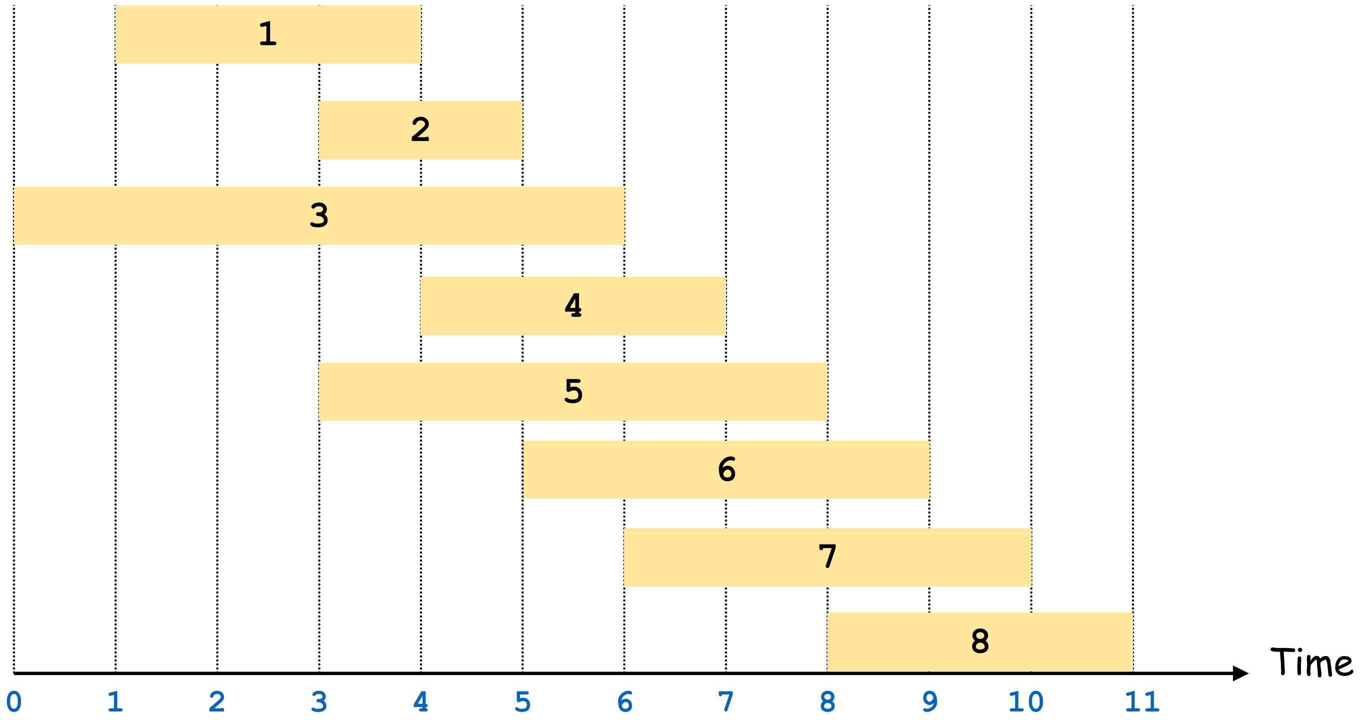
return OPT[j]

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \le f_2 \le \cdots \le f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



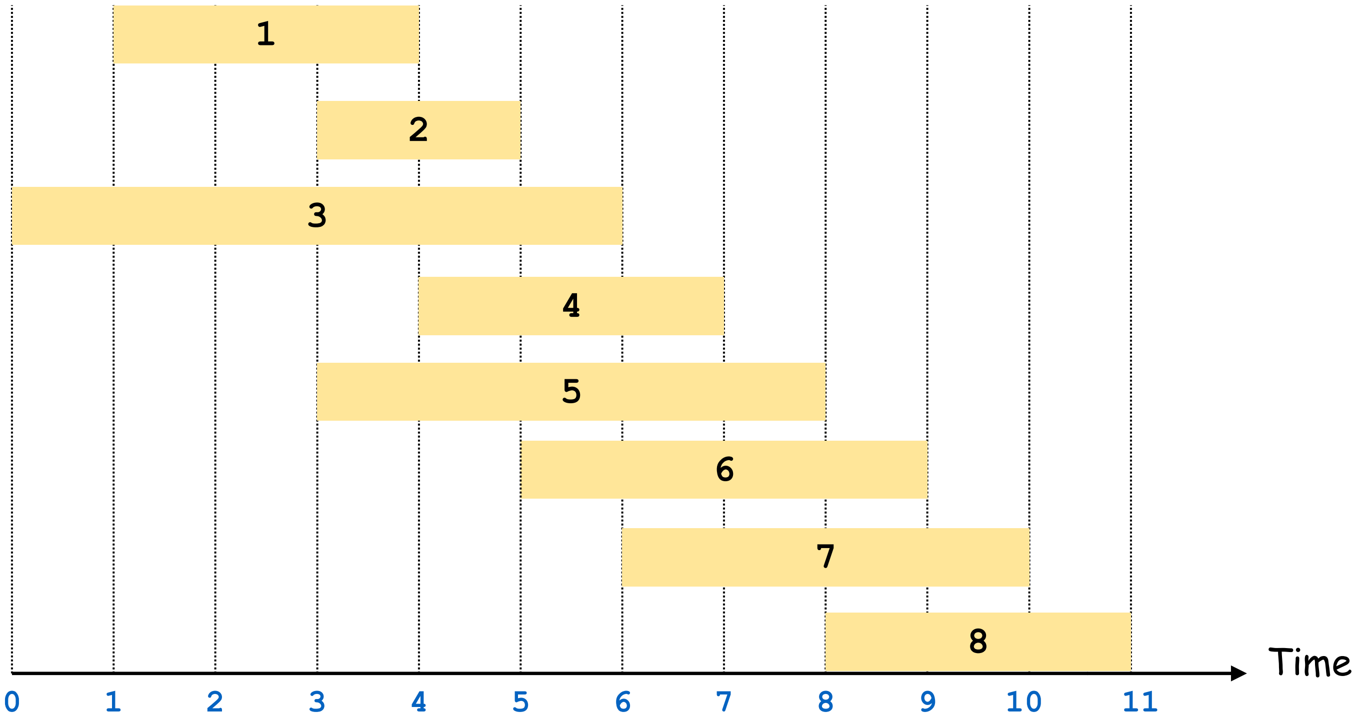
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \leq f_2 \leq \cdots \leq f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



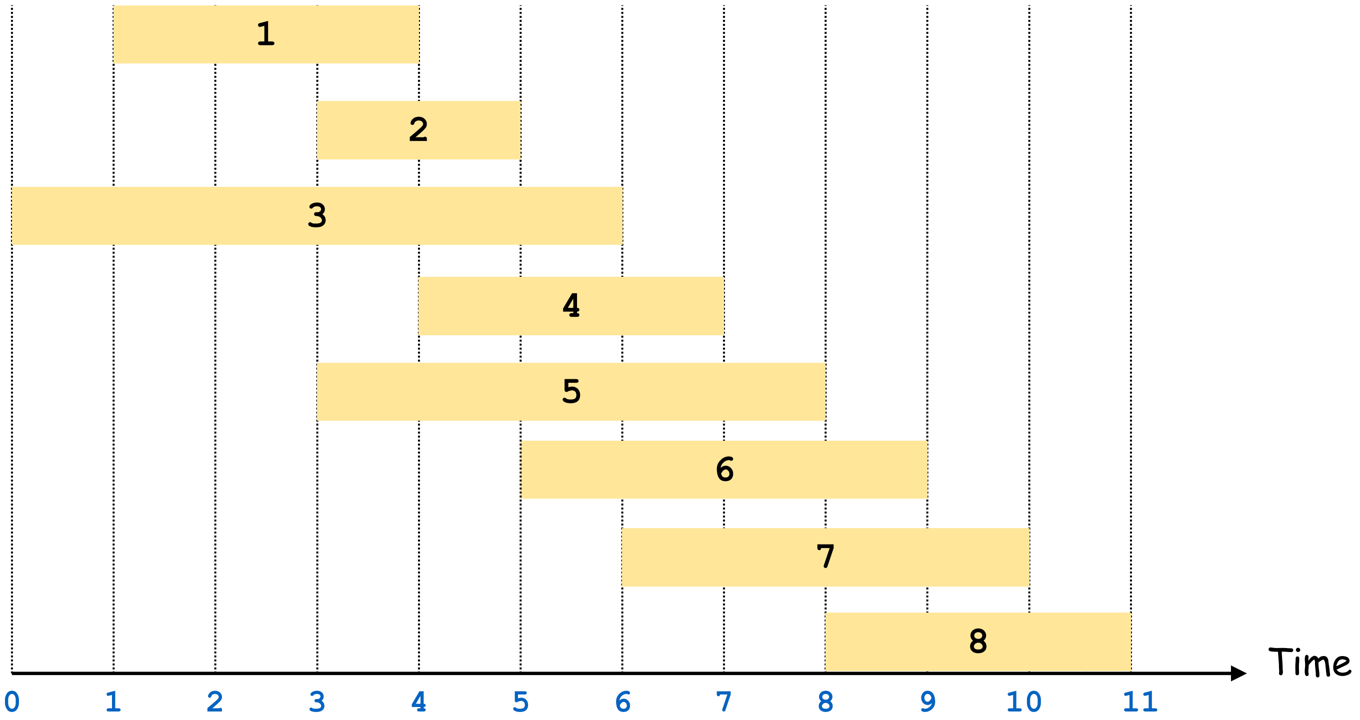
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \le f_2 \le \cdots \le f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



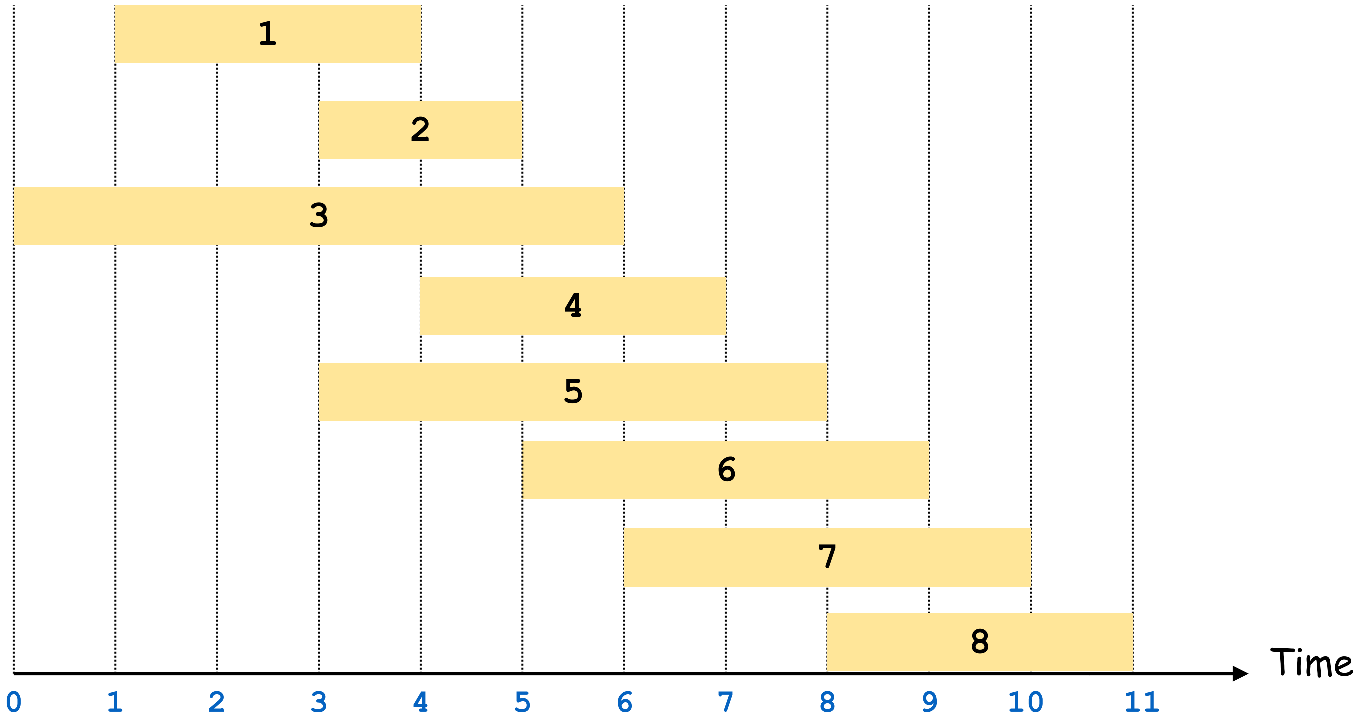
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \leq f_2 \leq \cdots \leq f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT\big(p(j)\big) + v_j, OPT(j-1)\}$$



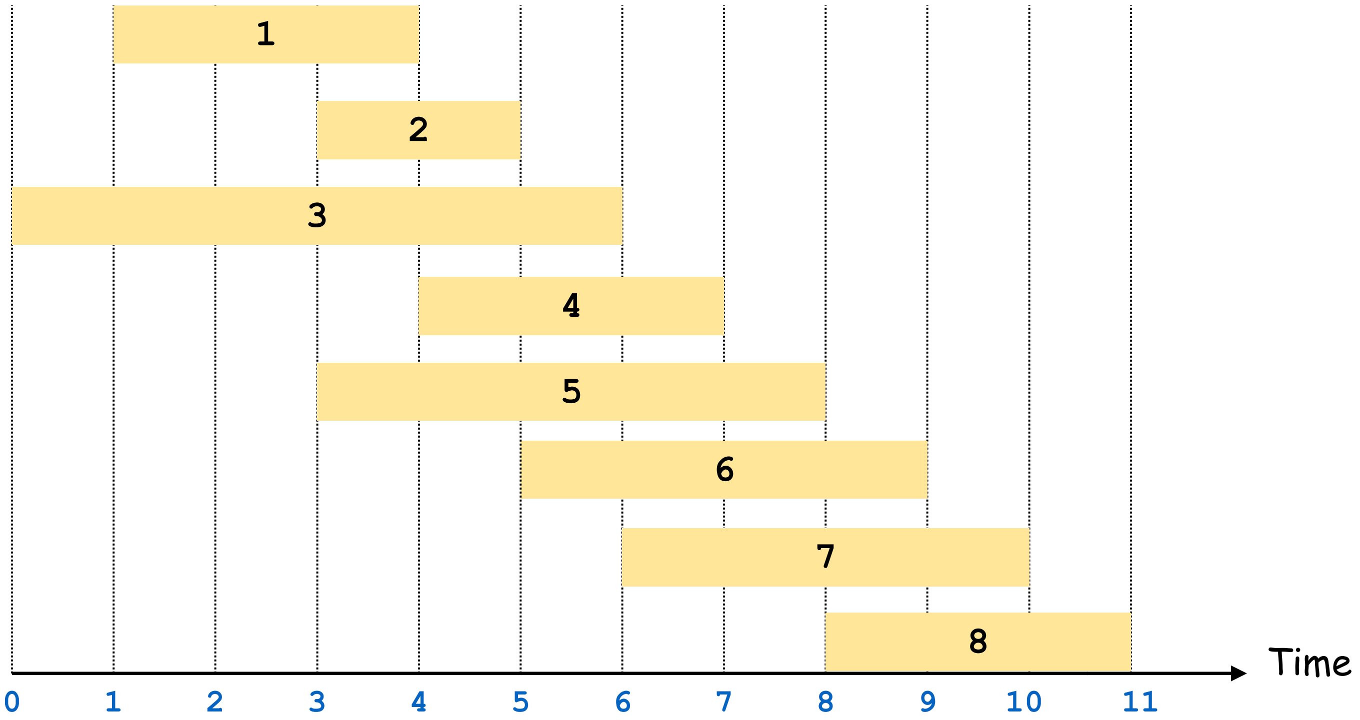
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $f_1 \le f_2 \le \cdots \le f_n$.

Defn: $p(j)$ = largest index $i < j$ s.t. job i is compatible with j.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



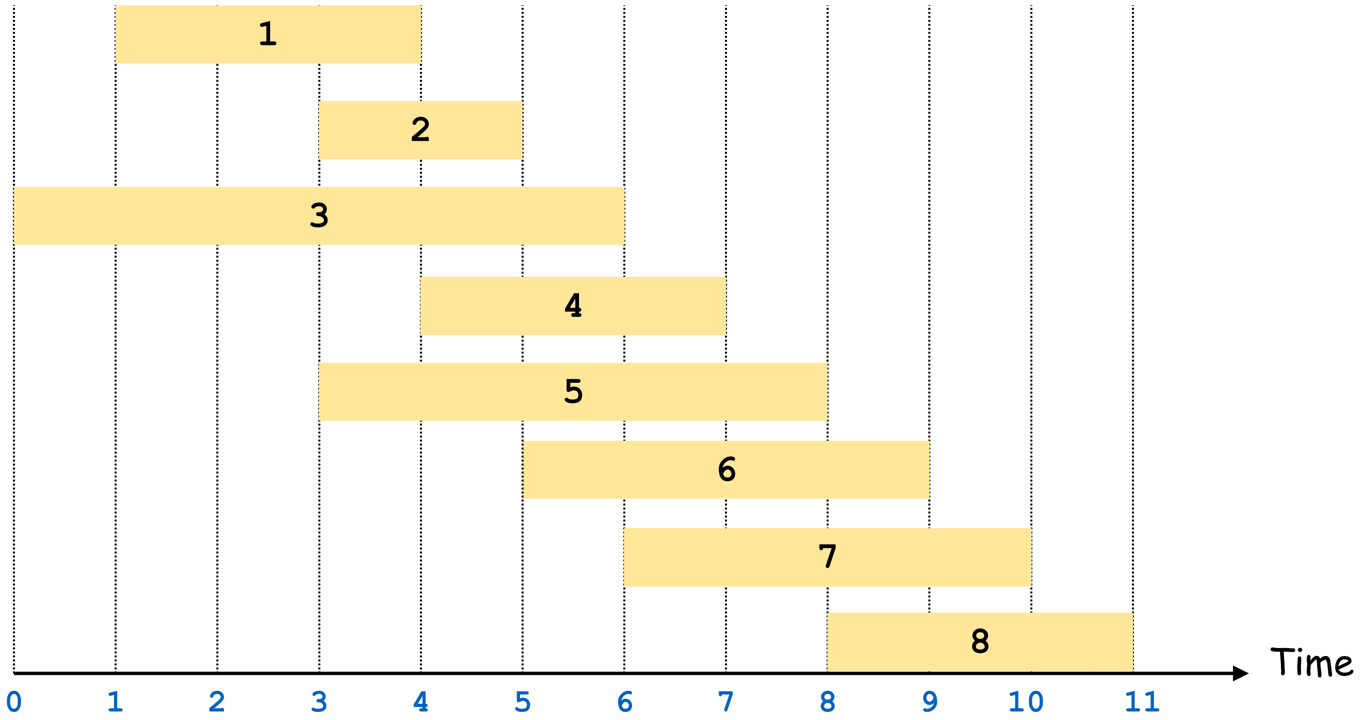
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Defn: $p(j)$ = largest index $i < j$ s.t. job i is compatible with j.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



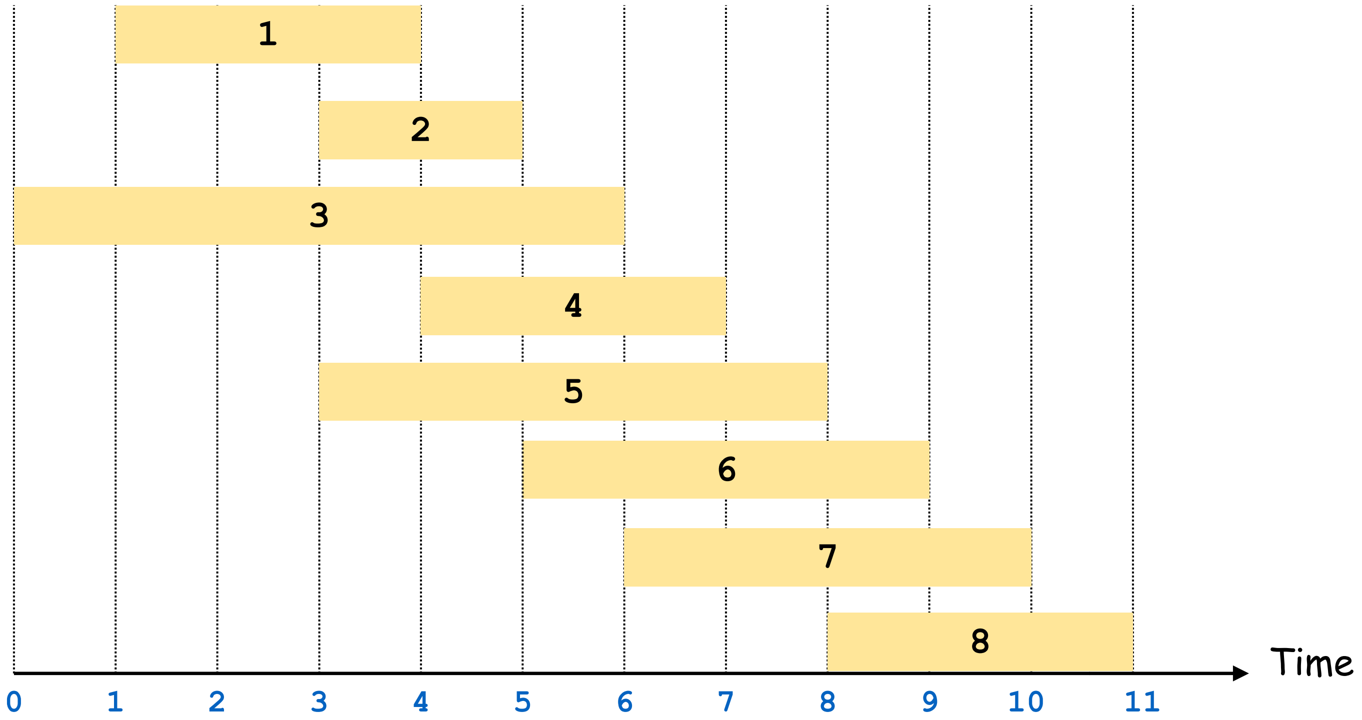
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \le f_2 \le \cdots \le f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



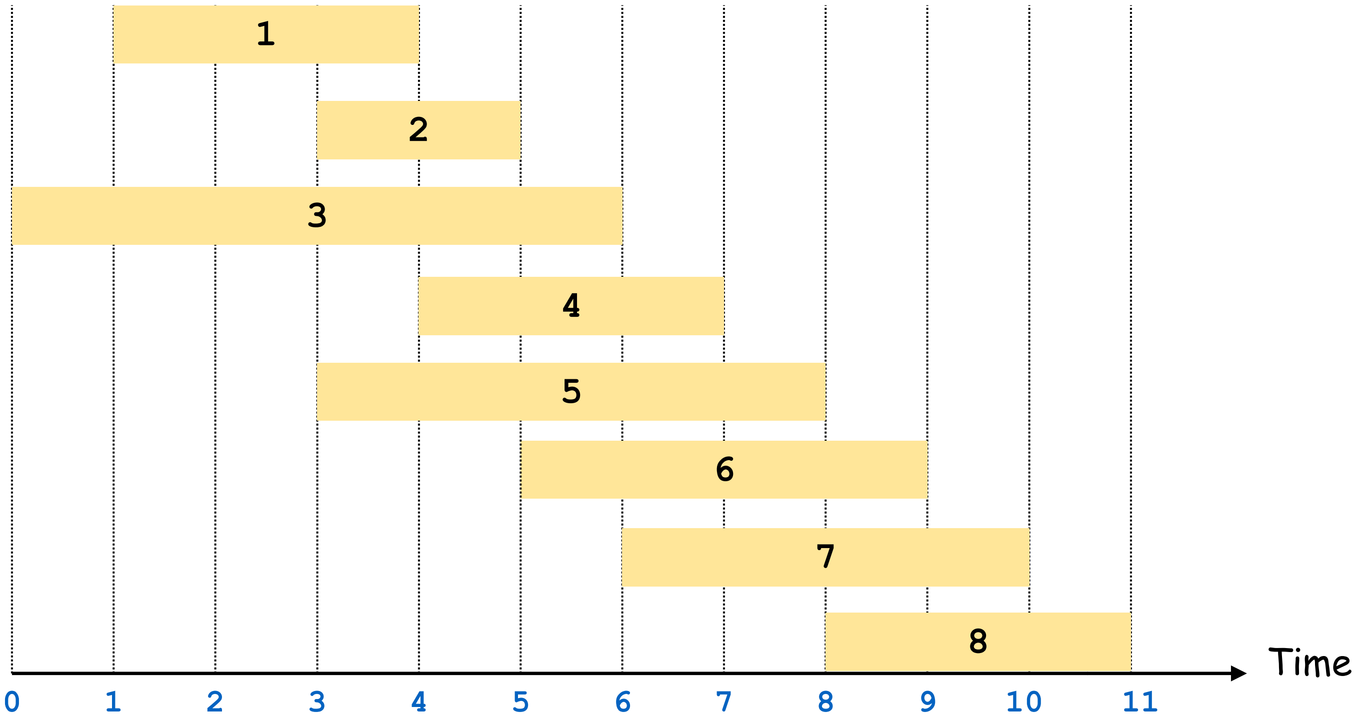
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \leq f_2 \leq \cdots \leq f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



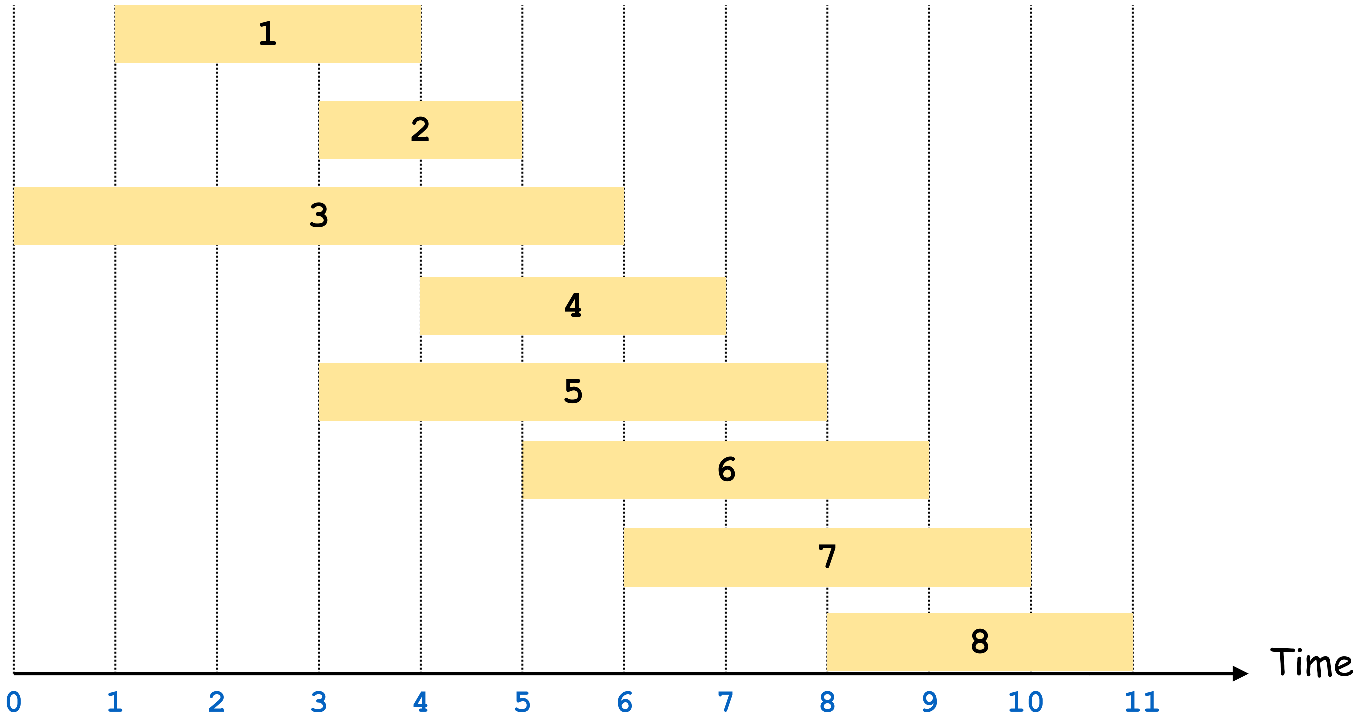
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Defn: $p(j)$ = largest index $i < j$ s.t. job i is compatible with j.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



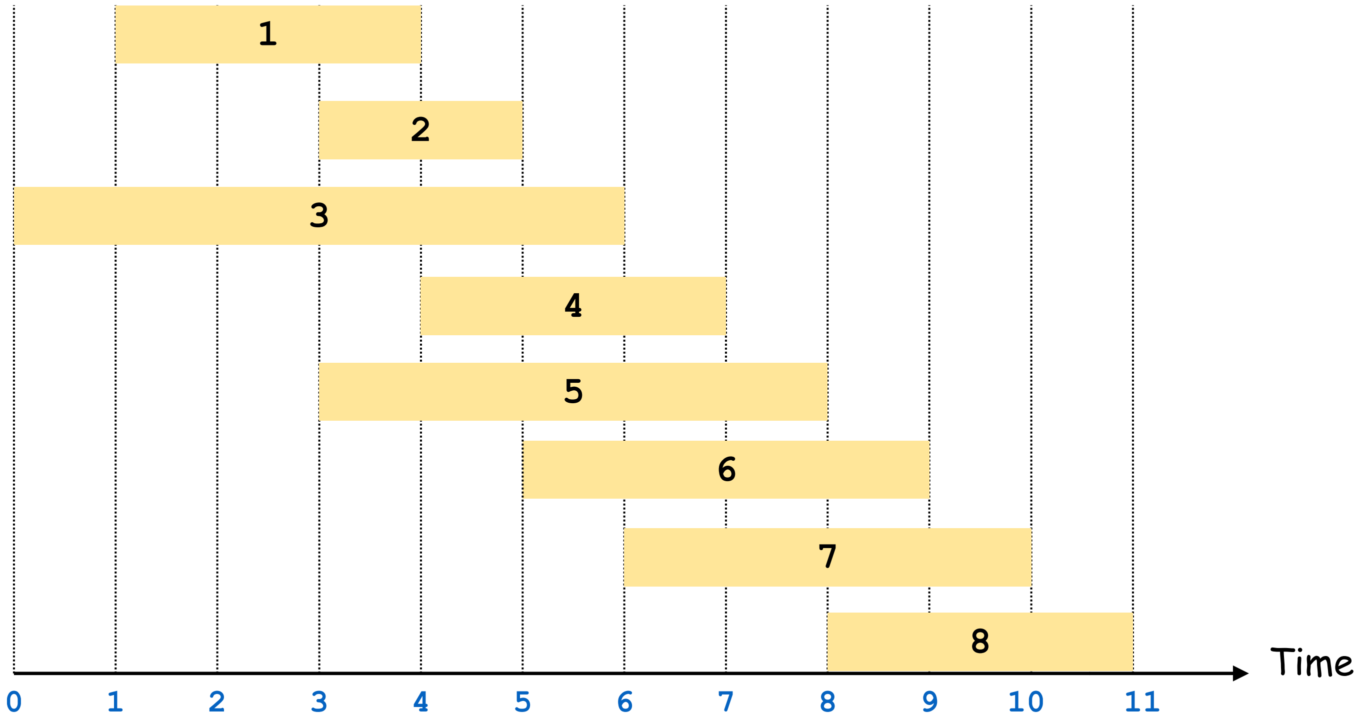
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	5
6	4	2	
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \leq f_2 \leq \cdots \leq f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



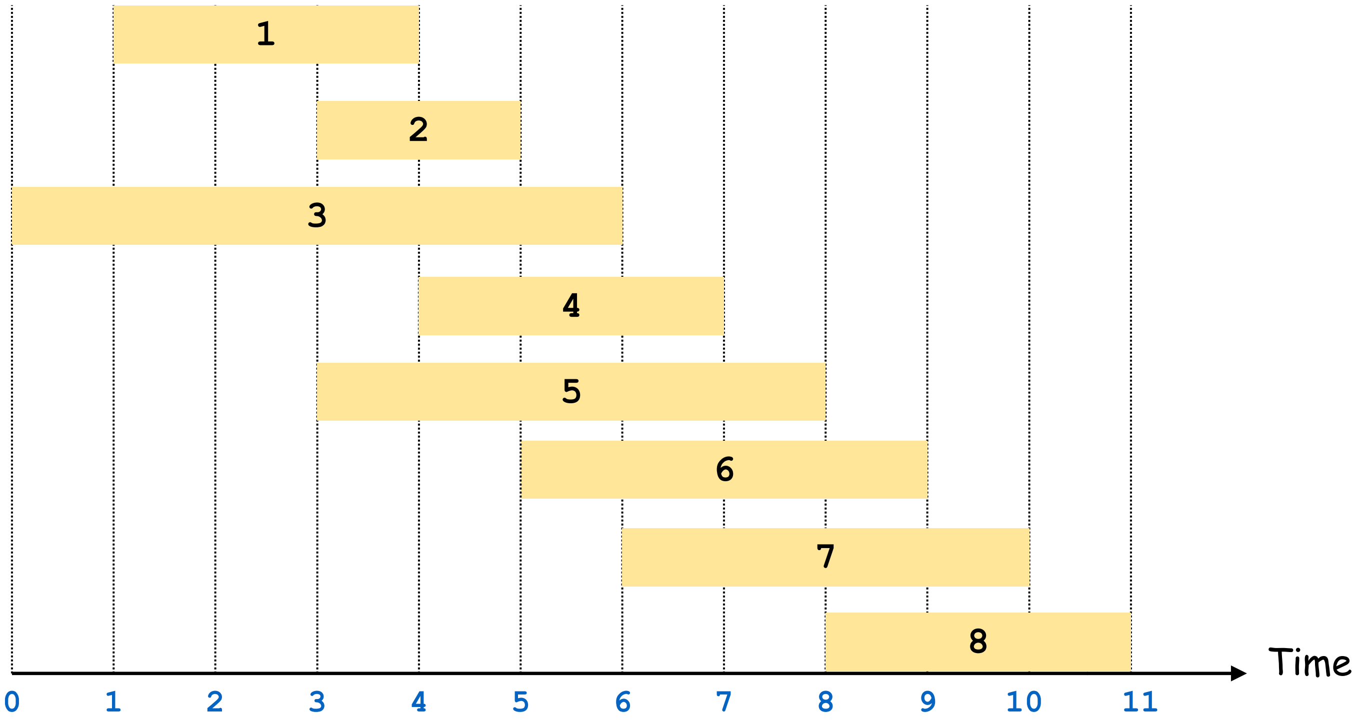
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \leq f_2 \leq \cdots \leq f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



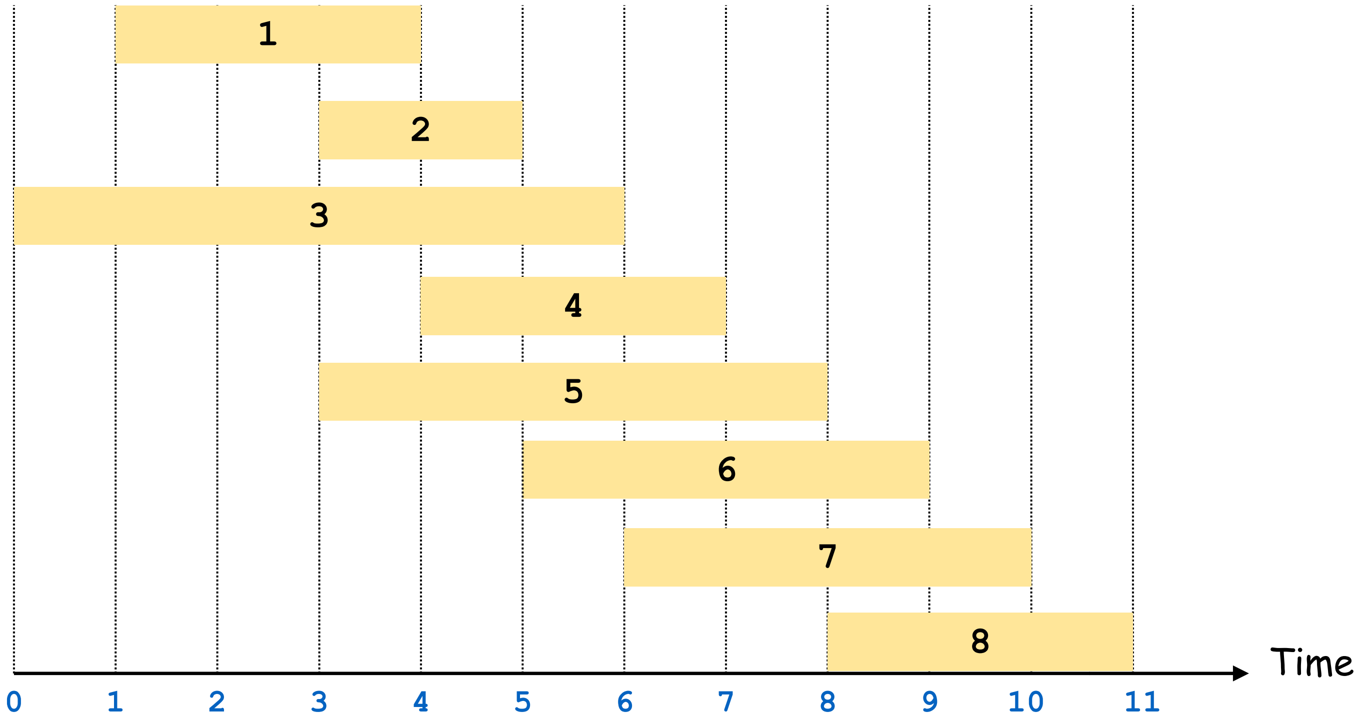
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Defn: $p(j)$ = largest index $i < j$ s.t. job i is compatible with j.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



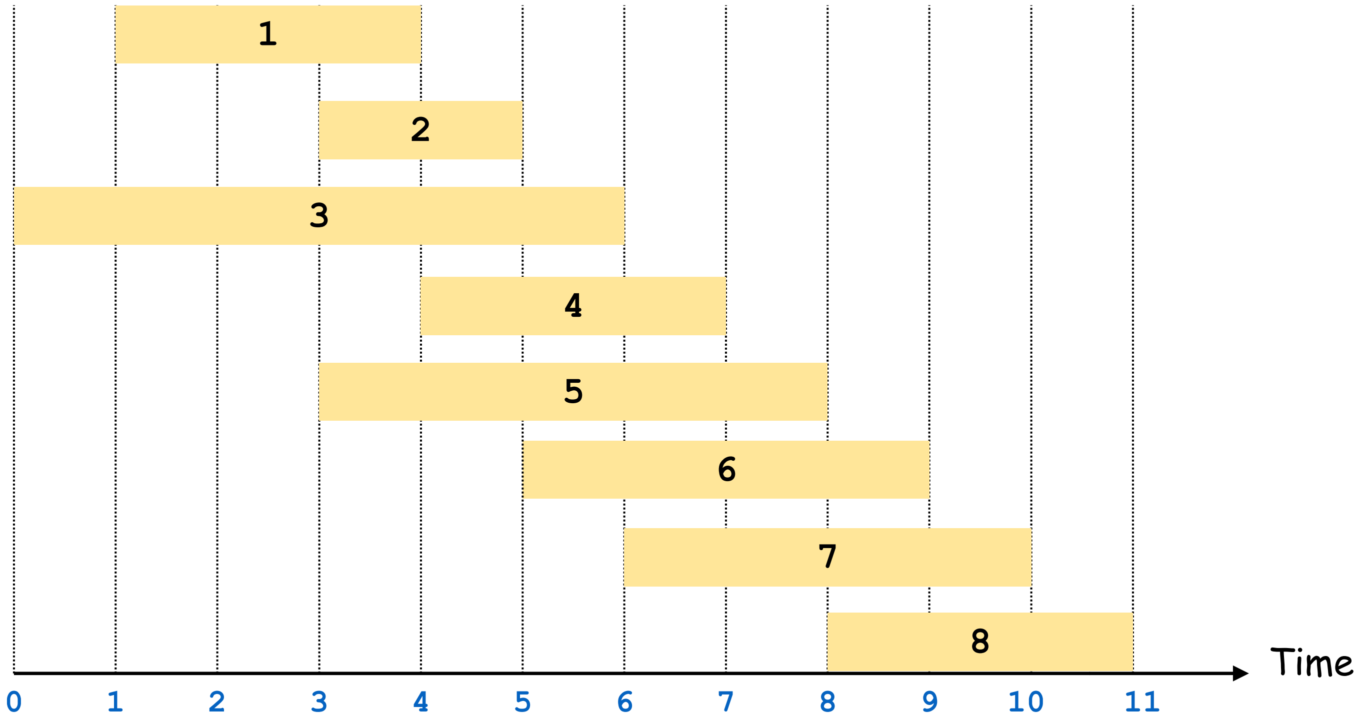
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Defn: $p(j)$ = largest index $i < j$ s.t. job i is compatible with j.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



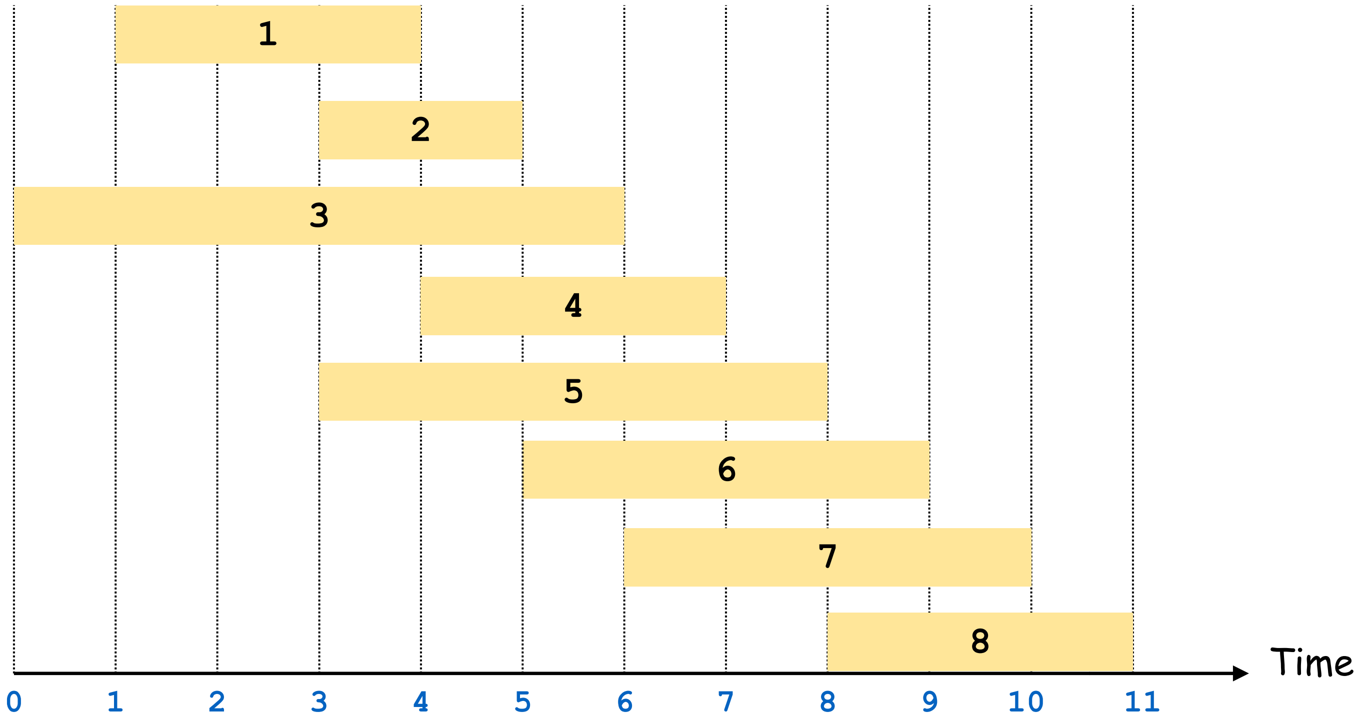
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	

Example Execution (iterative)

Notation: Label jobs by finishing time: $\boldsymbol{f_1 \le f_2 \le \cdots \le f_n}$.

Defn: $\boldsymbol{p(j)}$ = largest index $\boldsymbol{i < j}$ s.t. job $\boldsymbol{i}$ is compatible with $\boldsymbol{j}$.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	10

Weighted Interval Scheduling: Finding the Solution

So far we have computed the value **OPT**$(\boldsymbol{n})$ but we probably want to know what that solution **OPT** actually is!

We can do this, too, by keeping track of which option was better at each step.

Define **Used**$[\boldsymbol{j}]$ = $\begin{cases} \mathbf{1} & \text{solution with value } \textbf{OPT}(\boldsymbol{j}) \text{ includes request } \boldsymbol{j} \\ \mathbf{0} & \text{otherwise} \end{cases}$

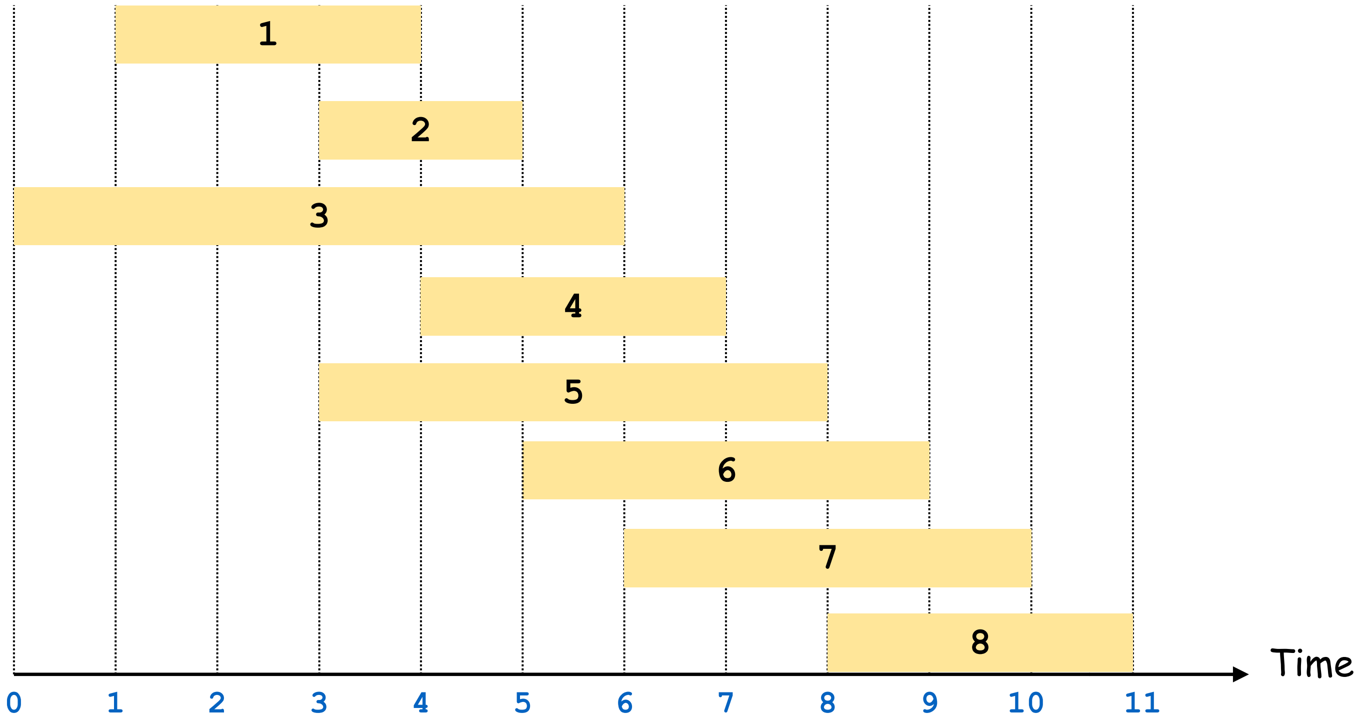
This gives a "pointer" that leads the way along a path to the optimal solution…

Weighted Interval Scheduling: Finding the Solution

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Defn: $p(j)$ = largest index $i < j$ s.t. job i is compatible with j.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



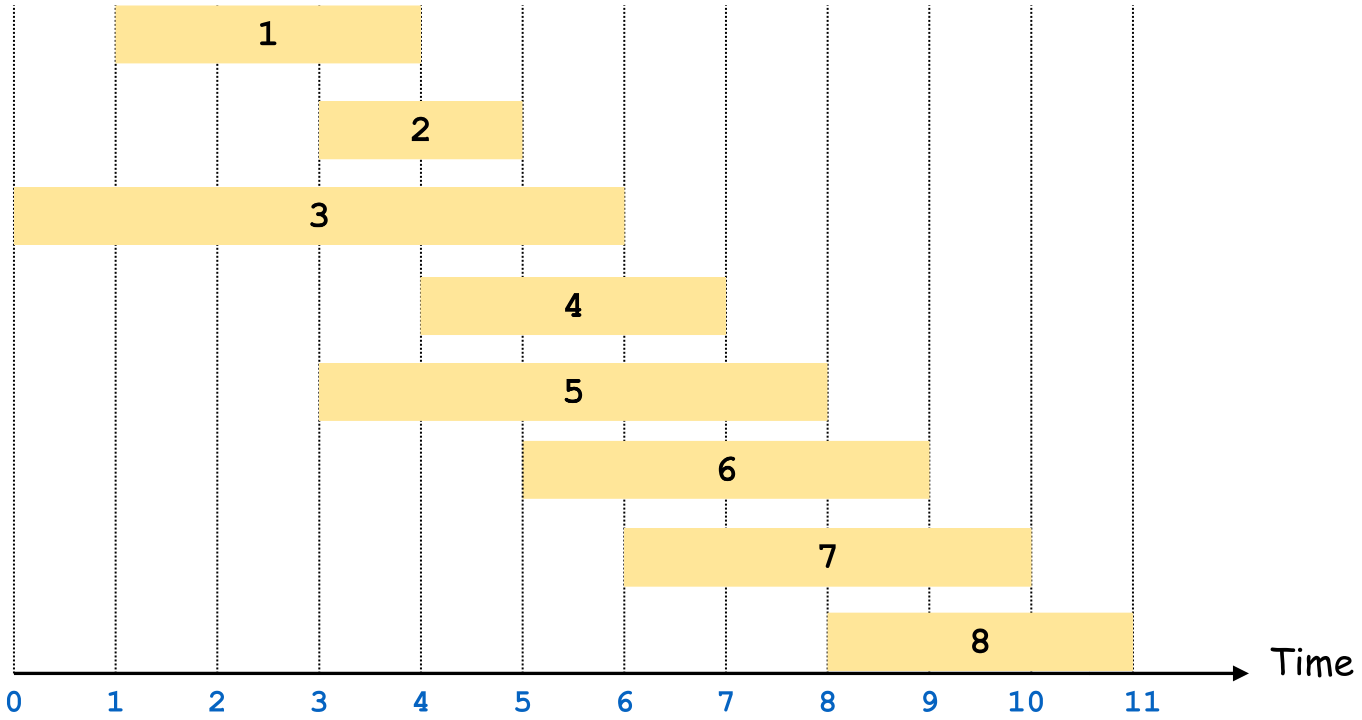
j	v_j	$p(j)$	OPT[j]	Used[j]
0	-	-	0	-
1	3	0	3	1
2	2	0	3	0
3	6	0	6	1
4	3	1	6	1
5	5	0	6	0
6	4	2	7	1
7	4	3	10	1
8	3	5	10	0

Weighted Interval Scheduling: Iterative Solution

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Defn: $p(j)$ = largest index $i < j$ s.t. job i is compatible with j.

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$



j	v_j	$p(j)$	OPT[j]	Used[j]
0	-	-	0	-
1	3	0	3	1
2	2	0	3	0
3	6	0	6	1
4	3	1	6	1
5	5	0	6	0
6	4	2	7	1
7	4	3	10	1
8	3	5	10	0

Weighted Interval Scheduling - Complete

Sort requests by finish time

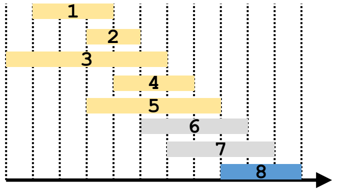
Compute each p(i)

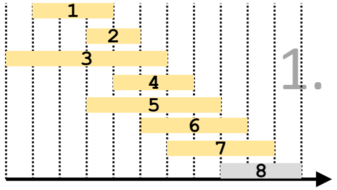
WIS(j):

```
OPT[0] = 0
for each i = 1 up to j:
    includei =  OPT[p(i)]+value[i]
    excludei = OPT[i − 1]
    if includei > excludei:
        OPT[i] = includei
        used[i] = 1
    else:
        OPT[i] = excludei
        used[i] = 0
return find_opt(used);
```

```
find_opt(used):
    j= n
    intervals = {}
    while j > 0:
        if used[j]==0:
            j = j − 1
        else:
            intervals.add(j)
            j = p(j)
    return intervals
```

Weighted Interval Scheduling – Four Steps





$$\max\{OPT(p(j)) + v_j, \mathrm{OPT}(j-1)\}$$

j	OPT[*j*]
0	0
1	
2	
3	
4	
5	
6	
7	
8	

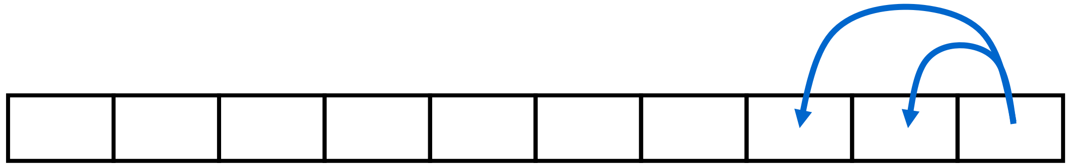
j	OPT[*j*]
0	0
1	
2	
3	
4	
5	
6	
7	
8	

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a "Bottom-up" (iterative) algorithm
   - Without this step: a "Top-down" (recursive) algorithm
4. See if there's a way to save space
   - Is it possible to reuse some memory locations?

Dynamic Programming Patterns

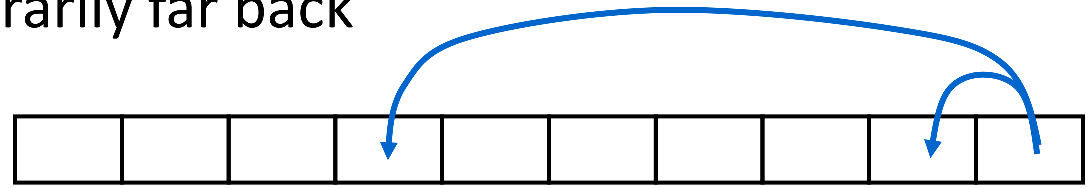
Fibonacci pattern:

- 1-dimensional, $O(1)$ values immediately prior
- Space saving possible



Weighted interval scheduling pattern:

- 1-dimensional, $O(1)$ values arbitrarily far back
- No space saving possible



Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

10	9	8	7	6	5	4	3	2	8

Weighted Interval Scheduling – Four Steps

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Step 1: Finding a Recursive Structure

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

If the value at the last index were included, then best solution would look like:

- The longest sequence ending with something less than that value,
- Followed by that value

$OPT(9)$ is 1 plus the max of:

- $OPT(3)$
- $OPT(2)$
- $OPT(1)$

6	3	4	2	7	5	10	6	8	5

Extend the longest solution that ends with something less than 5

$$\boldsymbol{OPT(j) = \begin{cases} 1 & j = 0 \\ 1 + max\{OPT(k) : k < j \text{ and } A[k] < A[j]\} & j > 0 \end{cases}}$$

Step 1: Finding a Recursive Structure

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

If the value at the last index were included, then best solution would look like:

- The longest sequence ending with something less than that value,
- Followed by that value

6	**3**	**4**	**2**	**7**	**5**	**10**	**6**	**8**	**5**

$OPT(8)$ is 1 plus the max of:

- $OPT(7)$
- $OPT(5)$
- $OPT(4)$
- $OPT(3)$
- $OPT(2)$
- $OPT(1)$
- $OPT(0)$

$$\boldsymbol{OPT(j) = \begin{cases} 1 & j = 0 \\ 1 + max\{OPT(k) : k < j \text{ and } A[k] < A[j]\} & j > 0 \end{cases}}$$

Step 1: Finding a Recursive Structure

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

If the value at the last index were included, then best solution would look like:

- The longest sequence ending with something less than that value,
- Followed by that value

6	3	4	2	7	5	10	6	8	5

$OPT(7)$ is 1 plus the max of:

- $OPT(5)$
- $OPT(3)$
- $OPT(2)$
- $OPT(1)$

$$\boldsymbol{OPT(j) = \begin{cases} 1 & j = 0 \\ 1 + max\{OPT(k) : k < j \text{ and } A[k] < A[j]\} & j > 0 \end{cases}}$$

Step 1: Finding a Recursive Structure

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

If the value at the last index were included, then best solution would look like:

- The longest sequence ending with something less than that value,
- Followed by that value

$OPT(3)$ is 1 plus the max of:

- $\emptyset$

6	**3**	**4**	**2**	**7**	**5**	**10**	**6**	**8**	**5**

$$\boldsymbol{OPT(j) = \begin{cases} 1 & j = 0 \\ 1 + max\{OPT(k) : k < j \text{ and } A[k] < A[j]\} & j > 0 \end{cases}}$$

LIS – Four Steps

6	3	4	2	7	5	10	6	8	5

$$1 + \boldsymbol{max\{OPT(k) : k < j \ and \ A[k] < A[j]\}}$$

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Step 2: Memory Structure

$$OPT(j) = \begin{cases} 1 & j = 0 \\ 1 + max\{OPT(k) : k < j \text{ and } A[k] < A[j]\} & j > 0 \end{cases}$$

- How many parameters?
  - Just 1
- What does each represent?
  - An index in the array
- How many different values?
  - Length of the array

$OPT =$

LIS Top-Down DP

```
LISRec(j):
    if OPT[j] not blank:    // Check if we've solved this already
        return OPT[j]
    if j==0:   // Check if this is a base case
        OPT[j] = 1    // Always save your solution before returning
        return OPT[j]
    best = 0
    for k = 0 up to j − 1:
        if A[k] < A[j]:
            best = max(best, LIS(k)) // Solve each subproblem, pick which to use
    OPT[j] = 1+best // Always save your solution before returning
    return 1+best
LIS(A):
    solution = 1
    for i=0 up to A.length:
        solution = max(solution, LISRec(i))
    return solution
```

LIS – Four Steps

6	3	4	2	7	5	10	6	8	5

$1 + max\{OPT(k) : k < j \text{ and } A[k] < A[j]\}$

1-dimensional memory of size n

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Step 2: Memory Structure

$$OPT(j) = \begin{cases} 1 & j = 0 \\ 1 + max\{OPT(k) : k < j \; and \; A[k] < A[j]\} & j > 0 \end{cases}$$

$$OPT = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline \quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad \\ \hline \end{array}$$

- For each choice of j we might need any solution before it
- Solve in order of increasing index.

LIS – Four Steps

6	3	4	2	7	5	10	6	8	5

$1 + max\{OPT(k) : k < j \text{ and } A[k] < A[j]\}$

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?

1-dimensional memory of size n

2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?

Increasing order of index

3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm

~~4. See if there’s a way to save space~~
   - ~~Is it possible to reuse some memory locations?~~

LIS Bottom-up DP

```
LIS(A):
    OPT[0]=1 // Solve and save the bast case solution
    for j = 0 up to A.length: // Going in bottom-up order
        best = 0 // Applying the recursive structure
        for k = 0 up to j:
            if A[k]<A[j]:
                best = max(best, OPT[k])
        OPT[j] = 1 + best // Save the solution for reuse
    solution = 0
    for i = 0 up to A.length: // This was the for loop from the “public” method
        solution = max(solution, OPT(i))
    return solution
```

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1									
pred[j]	0									
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1								
pred[j]	0	0								
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2							
pred[j]	0	0	2							
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1						
pred[j]	0	0	2	0						
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3					
pred[j]	0	0	2	0	3					
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3				
pred[j]	0	0	2	0	3	3				
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4			
pred[j]	0	0	2	0	3	3	5			
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4		
pred[j]	0	0	2	0	3	3	5	6		
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4	5	
pred[j]	0	0	2	0	3	3	5	6	8	
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4	5	3
pred[j]	0	0	2	0	3	3	5	6	8	3
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array $\boldsymbol{A}$ of $\boldsymbol{n}$ integers.

Find: A longest possible sequence $\boldsymbol{i_1 < i_2 < \cdots < i_k}$ such that $\boldsymbol{A[i_1] < A[i_2] < \cdots < A[i_k]}$.

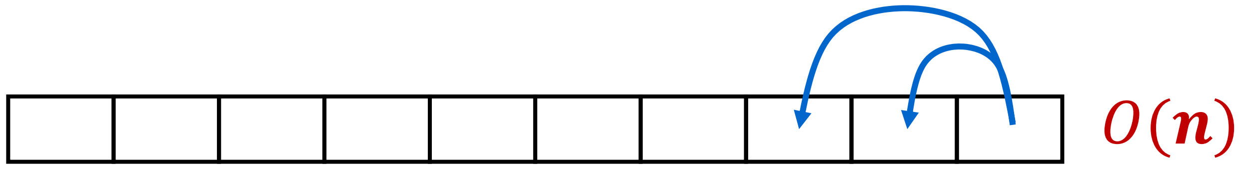
6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4	5	3
pred[j]	0	0	2	0	3	3	5	6	8	3
j	1	2	3	4	5	6	7	8	9	10

Dynamic Programming Patterns

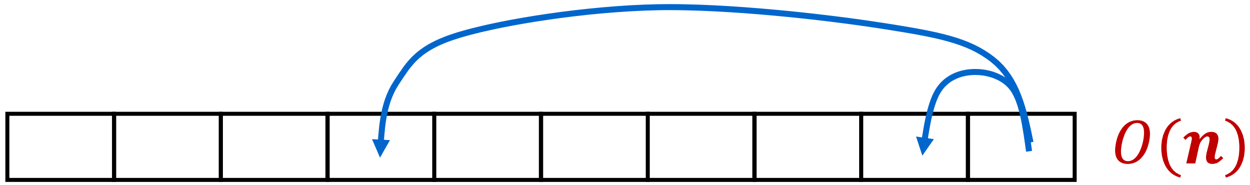
Fibonacci pattern:

- 1-D, $O(1)$ immediately prior
- $O(1)$ space



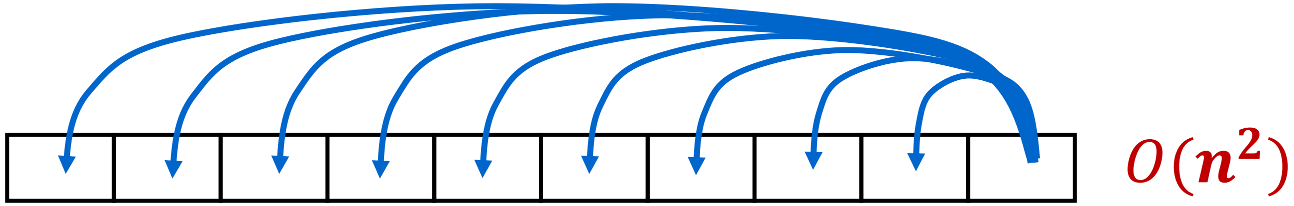
Weighted interval scheduling pattern:

- 1-D, $O(1)$ arbitrary prior
- $O(n)$ space



Longest increasing subsequence pattern:

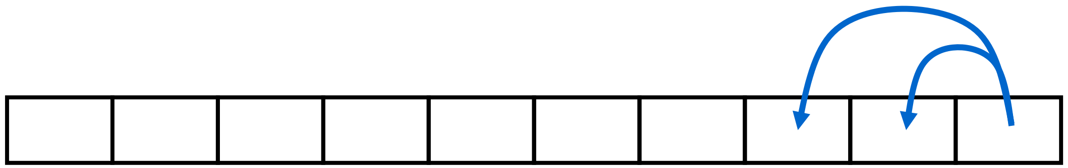
- 1-D, all $n-1$ prior
- $O(n)$ space



Dynamic Programming Patterns

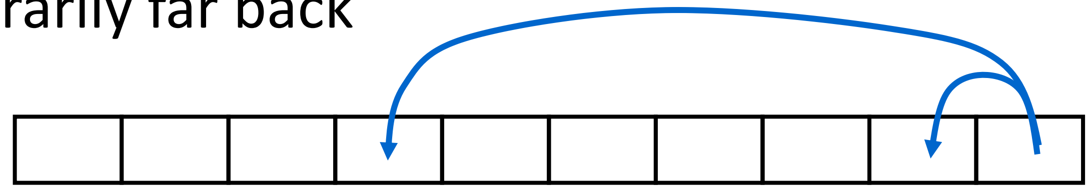
Fibonacci pattern:

- 1-dimensional, $O(1)$ values immediately prior
- Space saving possible



Weighted interval scheduling pattern:

- 1-dimensional, $O(1)$ values arbitrarily far back
- No space saving possible



String Similarity

How similar are two strings?

- `ocurrance`
- `occurrence`

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

Clearly a better matching

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

Maybe a better matching

- depends on cost of gaps vs mismatches

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

62

Edit Distance

Applications:

- Basis for Unix **`diff`**.
- Speech recognition.
- Computational biology.
- autocorrect

Edit distance: [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq} if symbol p is replaced by symbol q.
- **Cost** = gap penalties + mismatch penalties.

C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	T	A	C	A	T

$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

-	C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	-	T	A	C	A	T

$2\delta + \alpha_{CA}$

63

Sequence Alignment

Sequence Alignment:

Given: Two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$

Find: "Alignment" of X and Y of minimum edit cost.

Defn: An alignment M of X and Y is a set of ordered pairs x_i-y_j
s.t. each symbol of X and Y occurs in at most one pair
with no "crossing pairs".

The pairs x_i-y_j and $x_{i'}$-$y_{j'}$ cross iff $i < i'$ but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i,y_j)\in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i:\ x_i \text{ unmatched}} \delta + \sum_{j:\ y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Note: if $x_i = y_j$ then $\alpha_{x_i y_j} = 0$

Example:

CTACCG vs TACATG

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G
-	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

$M = \{x_2\text{-}y_1, x_3\text{-}y_2, x_4\text{-}y_3, x_5\text{-}y_4, x_6\text{-}y_6\}$

Edit Distance – Four Steps

1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Step 1: Identify Recursive Structure

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	G
T	A	C	A	T	G
y_1	y_2	y_3	y_4	y_5	y_6

Consider the last two indices x_i and y_j

Options for what to do with them:

Option 1:
Match them

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	G
T	A	C	A	T	G
y_1	y_2	y_3	y_4	y_5	y_6

We use up one index from x and y

Accrue a mismatch penalty

$$OPT(i-1, j-1) + \alpha_{x_i y_j}$$

Option 2:
Don't match x_i

	x_1	x_2	x_3	x_4	x_5	x_6
	C	T	A	C	C	G
T	A	C	A	T	G	-
y_1	y_2	y_3	y_4	y_5	y_6	

We use up one index from x only

Accrue a gap penalty

$$OPT(i-1, j) + \delta$$

Option 3:
Don't match y_i

x_1	x_2	x_3	x_4	x_5	x_6	
C	T	A	C	C	G	-
	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

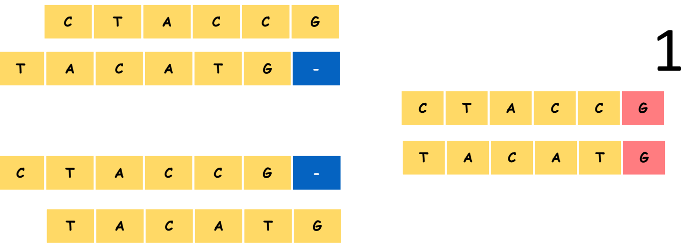
We use up one index from y only

Accrue a gap penalty

$$OPT(i, j-1) + \delta$$

$$OPT(i,j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min \begin{cases} OPT(i-1, j-1) + \alpha_{x_i y_j} \\ OPT(i-1, j) + \delta \\ OPT(i, j-1) + \delta \end{cases} \end{cases}$$

Edit Distance – Four Steps



1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Step 2: Identify Memory Structure

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	G
T	A	C	A	T	G
y_1	y_2	y_3	y_4	y_5	y_6

$$OPT(i,j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min \begin{cases} OPT(i-1,j-1) + \alpha_{x_i y_j} \\ OPT(i-1,j) + \delta \\ OPT(i,j-1) + \delta \end{cases} \end{cases}$$

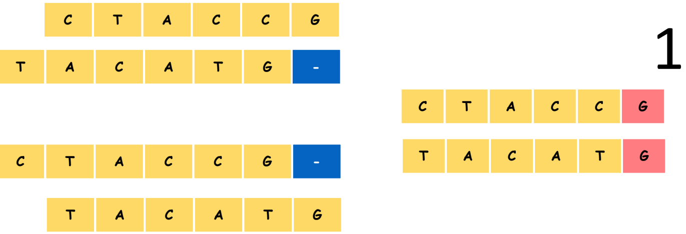
- How many parameters?
  - 2
- What does each represent?
  - The number of items in each sequence
- How many different values?
  - Length of sequence x for i
  - Length of sequence y for j
  - $n \cdot m$ overall

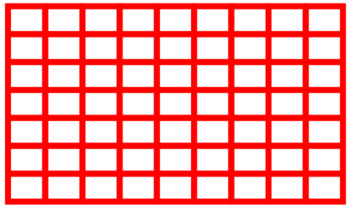
		y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
x_1									
x_2									
x_3									
x_4									
x_5									
x_6									

Top-Down Sequence Alignment

```
align(i, j):
    if OPT[i][j] not blank:    // Check if we've solved this already
        return OPT[i][j]
    if i · j == 0:    // Check if this is a base case
        solution = (i + j) · δ
        OPT[i][j] = solution    // Always save your solution before returning
        return solution
    match = align(i − 1, j − 1) // solve each subproblem
    gapx = align(i − 1,j) // solve each subproblem
    gapy = align(j, i − 1) // solve each subproblem
    solution = min(match + α_{x_i y_j}, gapx + δ, gapy+ δ) // Pick the subproblem to use
    OPT[i][j] = solution    // Always save your solution before returning
    return solution
```

Edit Distance – Four Steps

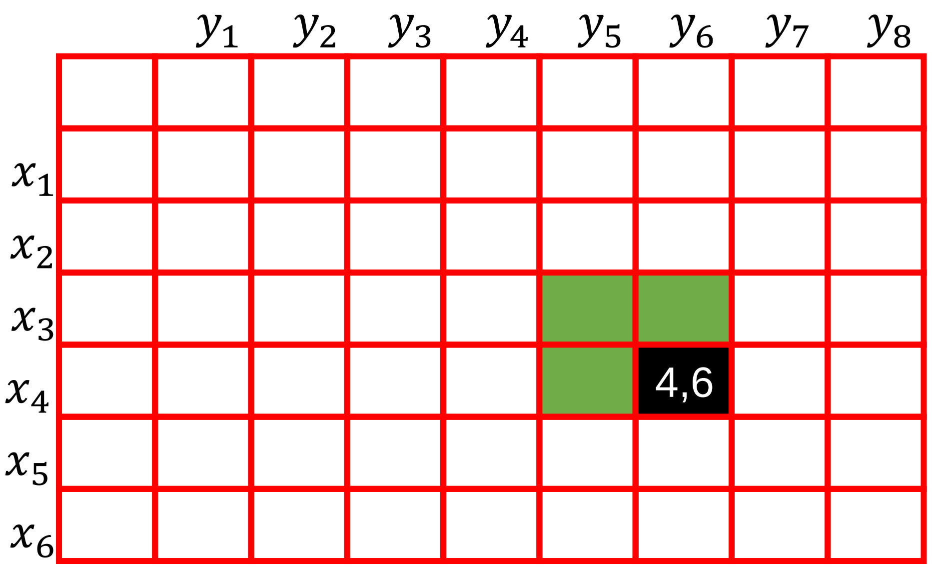




1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Step 3: Identify Order of Evaluation

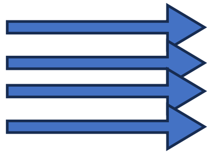
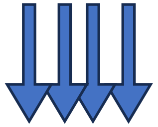
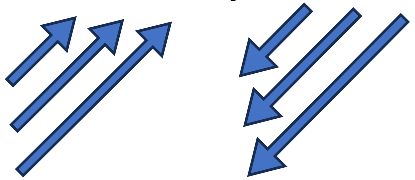
$$OPT(i,j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min \begin{cases} OPT(i-1, j-1) + \alpha_{x_i y_j} \\ OPT(i-1, j) + \delta \\ OPT(i, j-1) + \delta \end{cases} \end{cases}$$



Any of these orders will work:

Each index depends on 3 others:

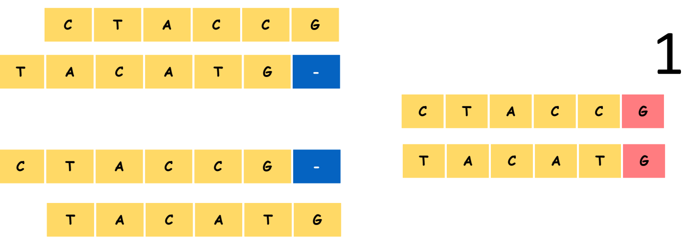
1. The one above it: $(i-1, j)$
2. The one to its left: $(i, j-1)$
3. The one to it's upper left: $(i-1, j-1)$

- Top-to-bottom, then left-to-right 
- Left-to-right, then top-to-bottom 
- Diagonally 

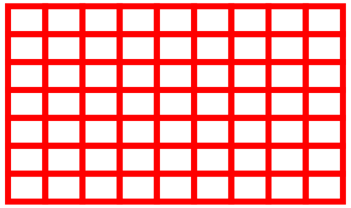
Bottom-Up Sequence Alignment

```
align(x, y):
    for i = 0 up to n:
        OPT[i][0] = 0   // Solve and save base cases
    for j = 0 up to m:
        OPT[0][j] = 0   // Solve and save base cases
    for i = 1 up to n:
        for j = 1 up to m :
            match = OPT[i − 1][j − 1] // solve each subproblem
            gapx = OPT[i][j − 1] // solve each subproblem
            gapy = OPT[i − 1][j] // solve each subproblem
            solution = min(match + α_{x_i y_j}, gapx + δ, gapy+ δ) // pick solution
            OPT[i][j] = solution // save solution
    return OPT[n][m]
```

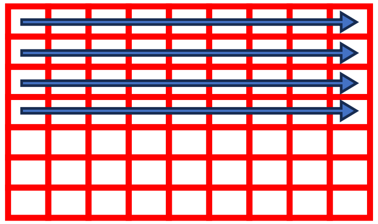
Edit Distance – Four Steps



1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?



2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?



3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a "Bottom-up" (iterative) algorithm
   - Without this step: a "Top-down" (recursive) algorithm
4. See if there's a way to save space
   - Is it possible to reuse some memory locations?

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\text{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1								
A	2								
G	3								
T	4								
T	5								
A	6								

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\text{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2								
G	3								
T	4								
T	5								
A	6								

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\mathrm{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1					
G	3								
T	4								
T	5								
A	6								

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\mathrm{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4								
T	5								
A	6								

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta} = \boldsymbol{\alpha}_{\mathbf{mis}} = \mathbf{1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\mathrm{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with *AGACATTG* and *GAGTTA*: $\boldsymbol{\delta = \alpha_{\mathrm{mis}} = 1}$

		A	G	A	C	A	T	T	G
	0	**1**	2	3	4	5	6	7	8
G	1	1	**1**	2	3	4	5	6	7
A	2	1	2	**1**	2	3	4	5	6
G	3	2	1	2	**2**	**3**	4	5	5
T	4	3	2	2	3	3	**3**	4	5
T	5	4	3	3	3	4	3	**3**	4
A	6	5	4	3	4	3	4	4	**4**

Optimal Alignment

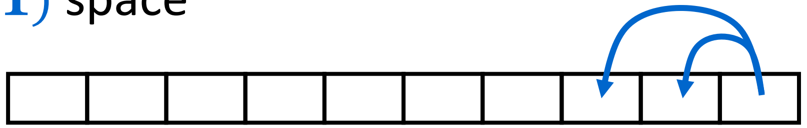
A G A C A T T G

_ G A G _ T T A

Dynamic Programming Patterns

Fibonacci pattern:

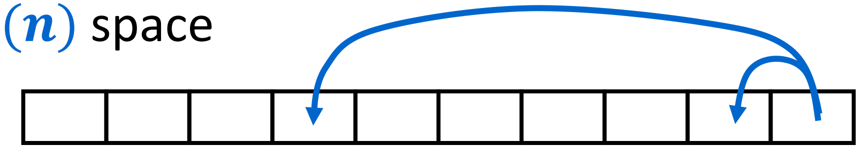
- 1-D, $O(1)$ immediately prior
- $O(1)$ space



$O(n)$

Weighted interval scheduling pattern:

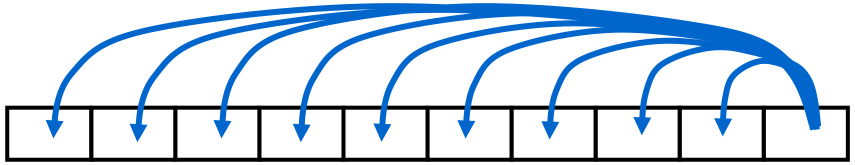
- 1-D, $O(1)$ arbitrary prior
- $O(n)$ space



$O(n)$

Longest increasing subsequence pattern:

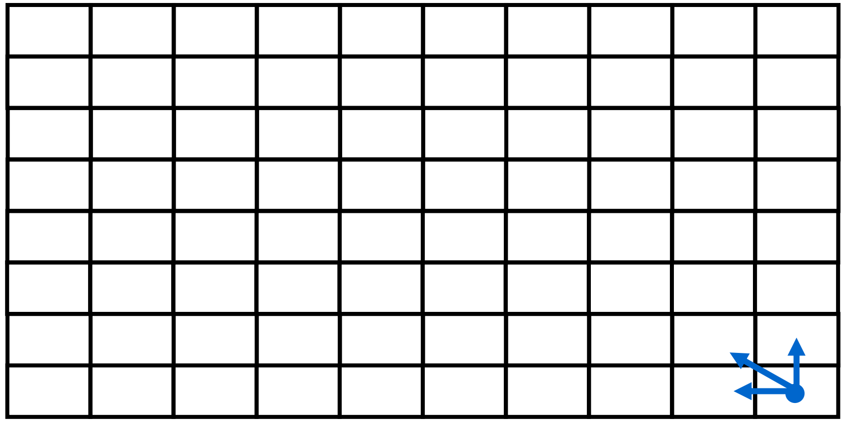
- 1-D, all $n-1$ prior
- $O(n)$ space



$O(n^2)$

Alignment pattern:

- 2-D, $O(1)$ in previous row, above and arbitrary prior
- $O(n \cdot m)$ space



$O(nm)$