COMP 251

Algorithms & Data Structures (Winter 2022)

Algorithm Paradigms – Dynamic Programming 1

School of Computer Science
McGill University

Slides of (Comp321 ,2021), Langer (2014), Kleinberg & Tardos, 2005 & Cormen et al., 2009, Jaehyun Park' slides CS 97SI, Top-coder tutorials, T-414-AFLV Course, Programming Challenges books.

Announcements

Outline

- Complete Search
- Divide and Conquer.
- Dynamic Programming.
  - Introduction.
  - Examples.
- Greedy.

Dynamic Programming– What is it?

- What’s the following equal to

  1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = ?

Dynamic Programming– What is it?

- What's the following equal to

  1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 21

  1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = ?

Dynamic Programming– What is it?

- What's the following equal to

1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 21

1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 22

subproblem

"DP is just a fancy way to remembering stuff to save time later!"

Jonathan Paulson

Famous saying:

Algorithms who cannot remember the past are condemned to recompute it.

Dynamic Programming– What is it?

- What's the following equal to

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \ = 21$$

$$\underbrace{1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1}_{\text{subproblem}}+1 \ = ?$$

DP breaks a problem into smaller overlapping sub-problems and avoids the computation of the same results more than once.

Famous saying:

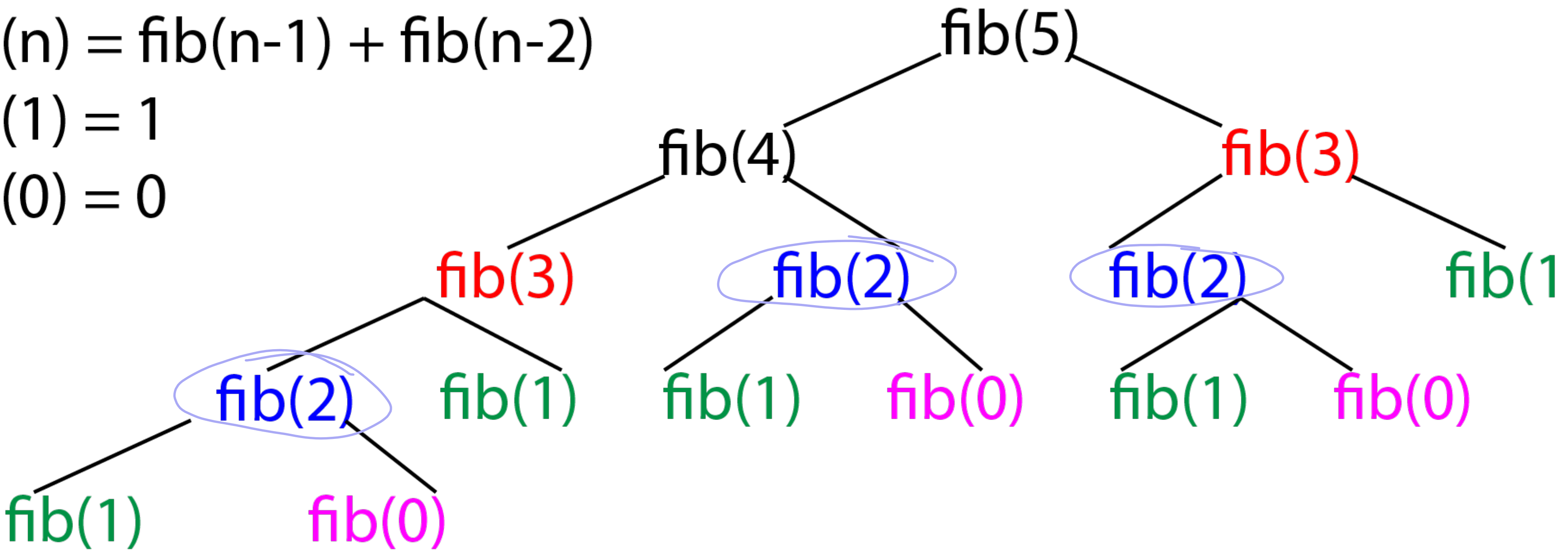
Those who cannot remember the past are condemned to repeat it.	Algorithms who cannot remember the past are condemned to recompute it.

Dynamic Programming– What is it?

fib(n) = fib(n-1) + fib(n-2)

fib(1) = 1

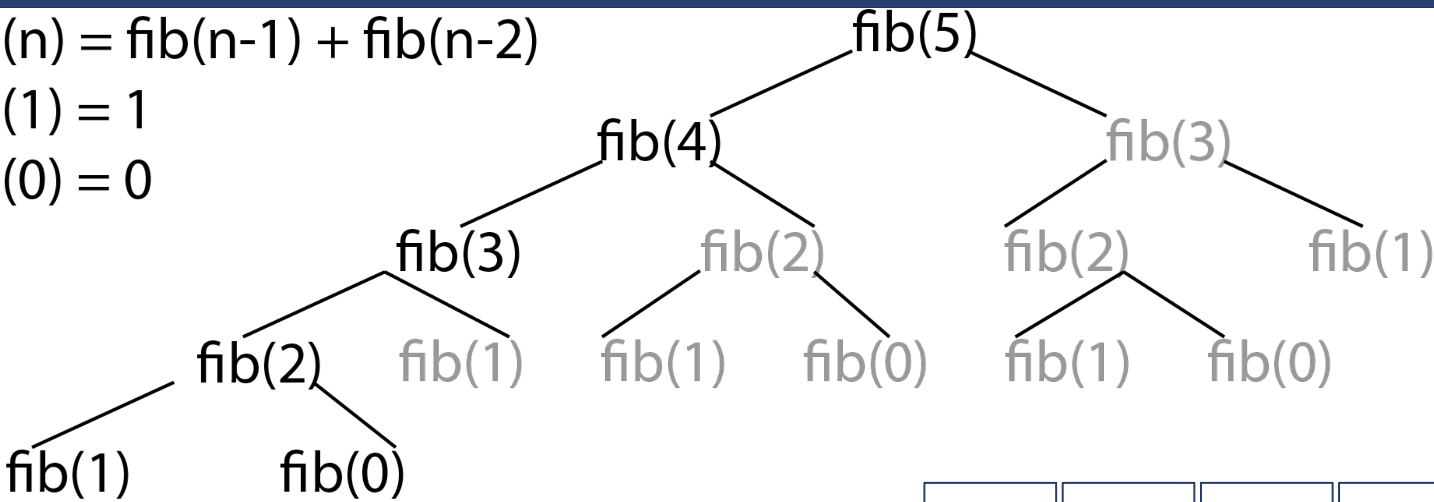
fib(0) = 0




```
fib(n):
    if n <= 1
        return n
    return fib(n - 1) + fib(n - 2)
```

Dynamic Programming– What is it?

fib(n) = fib(n-1) + fib(n-2)
fib(1) = 1
fib(0) = 0



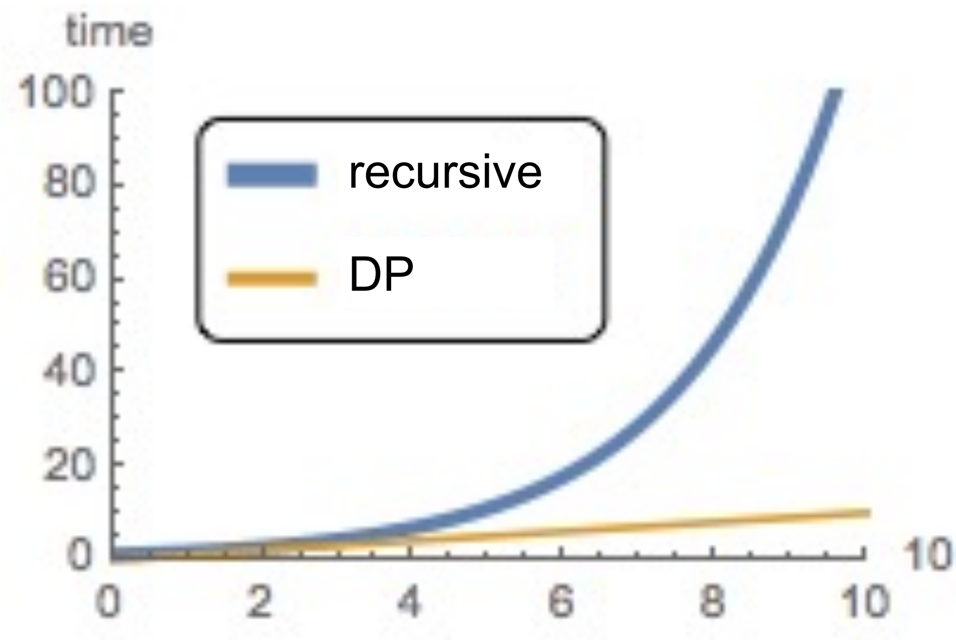
FibArray =

fib(0)	fib(1)	fib(2)	fib(3)	fib(4)	fib(5)

```
fib(n):
    if n <= 1
        return n
    else
       if FibArray[n-1] == 0
           FibArray[n-1] = fib(n-1)
       if FibArray[n-2] == 0
           FibArray[n-2] = fib(n-2)
    FibArray[n] = FibArray[n-2] + FibArray [n-1]
    return FibArray[n]
```

Store what you already computed

Dynamic Programming– What is it?

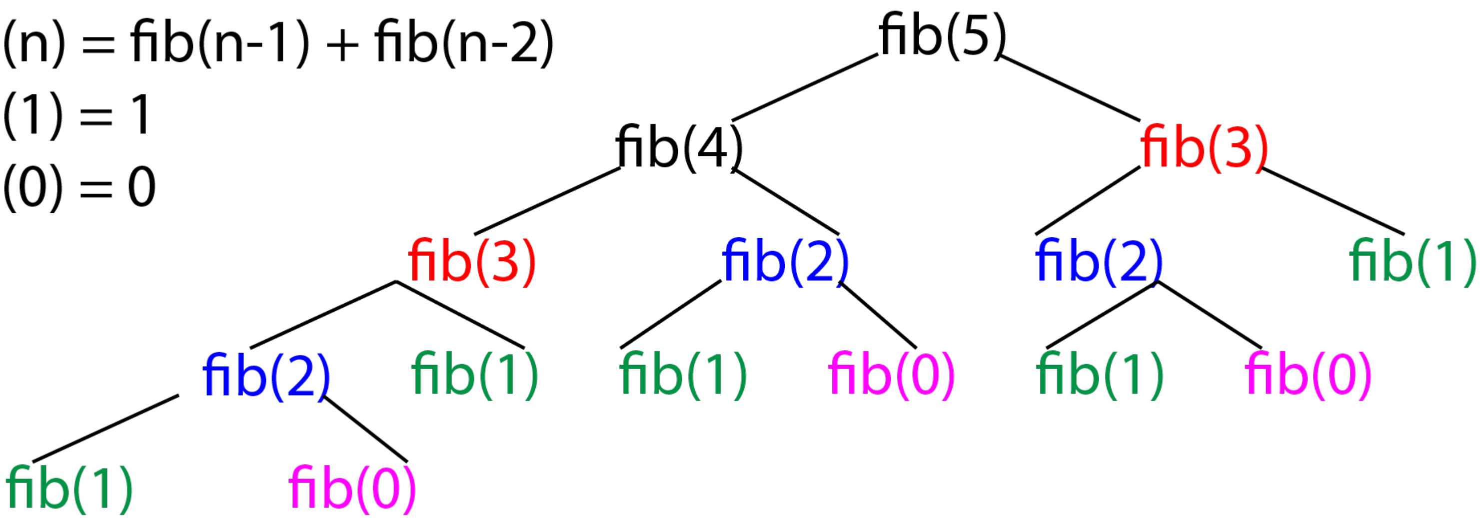


Wolfram Demonstrations Project

DP looks through all possible sub-problems and never re-computes the solution to any sub-problem. This implies correctness and efficiency, which we can not say of most techniques. This alone makes DP special

Dynamic Programming– When to use it?

$$fib(n) = fib(n-1) + fib(n-2)$$
$$fib(1) = 1$$
$$fib(0) = 0$$



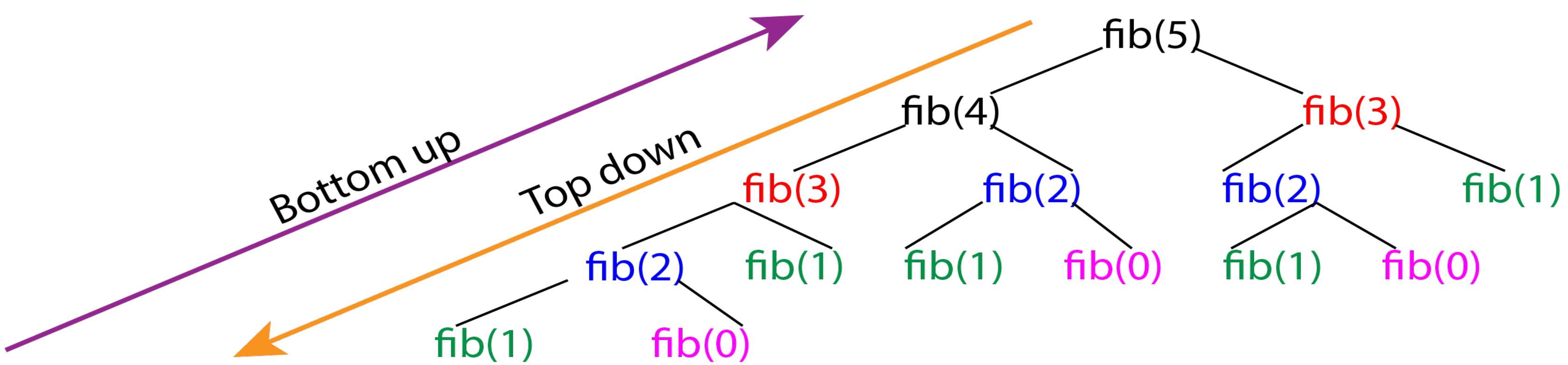
Key ingredients to make DP works

1. This problem has optimal sub-structures.
   - Solution for the sub-problem is part of the solution of the original problem.
2. This problem has overlapping sub-problems.

Dynamic Programming– How to solve it?

Steps for Solving DP Problems

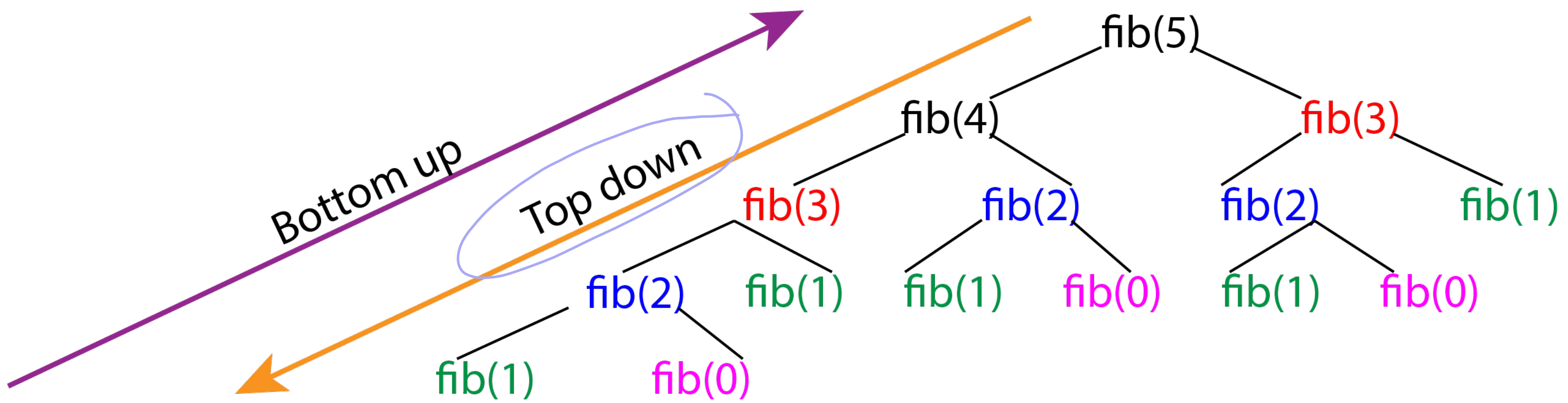
1. Define subproblems.
   fib(n-1) and fib(n-2) are subproblems of fib(n)
2. Write down the recurrence that relates subproblems.
   fib(n) = fib(n-1) + fib(n-2)
3. Recognize and solve the base cases.
   fib(0) = 0; fib(1) = 1
4. Implement a solving methodology.
   - Memoization: Top down approach => FibArray[]
   - Tabulation: Bottom up approach



Top-Down VS Bottom-Up

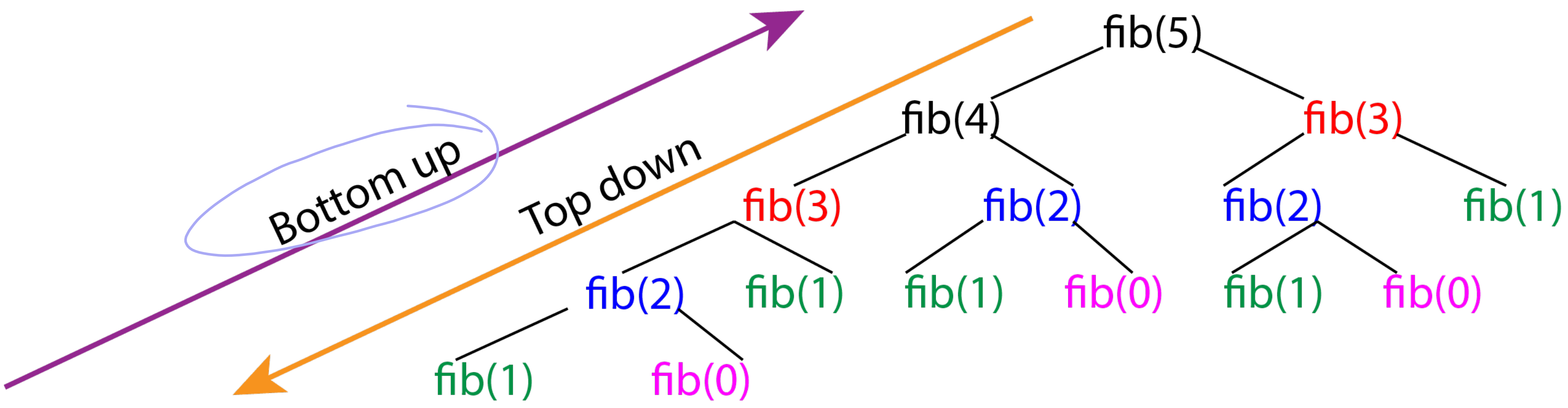
- Top Down (Memoization).
  - I will be an expert in dynamic programming. How? I will work hard like crazy. How? I'll practice more and try to improve. How? I'll actively participate in David classes. How? I will attend all David's classes. Then, I'm going to learn dynamic programming.
- Bottom Up (Tabulation).
  - I'm going to learn dynamic programming. Then, I will attend all David's classes. Then, I will actively participate in David classes. Then, I'll practice even more and try to improve. After working hard like crazy, I will be an expert in dynamic

DP - Top-Down



1. Initialize a DP 'memo' table with dummy values, e.g. '-1'.
   - The dimension of the DP table must be the size of distinct sub-problems.
2. At the start of recursive function, simply check if this current state has been computed before.
   (a) If it is, simply return the value from the DP memo table, O(1).
   (b) If it is not, compute as per normal (just once) and then store the computed value in the DP memo table so that further calls to this sub-problem is fast.

DP – Bottom-Up

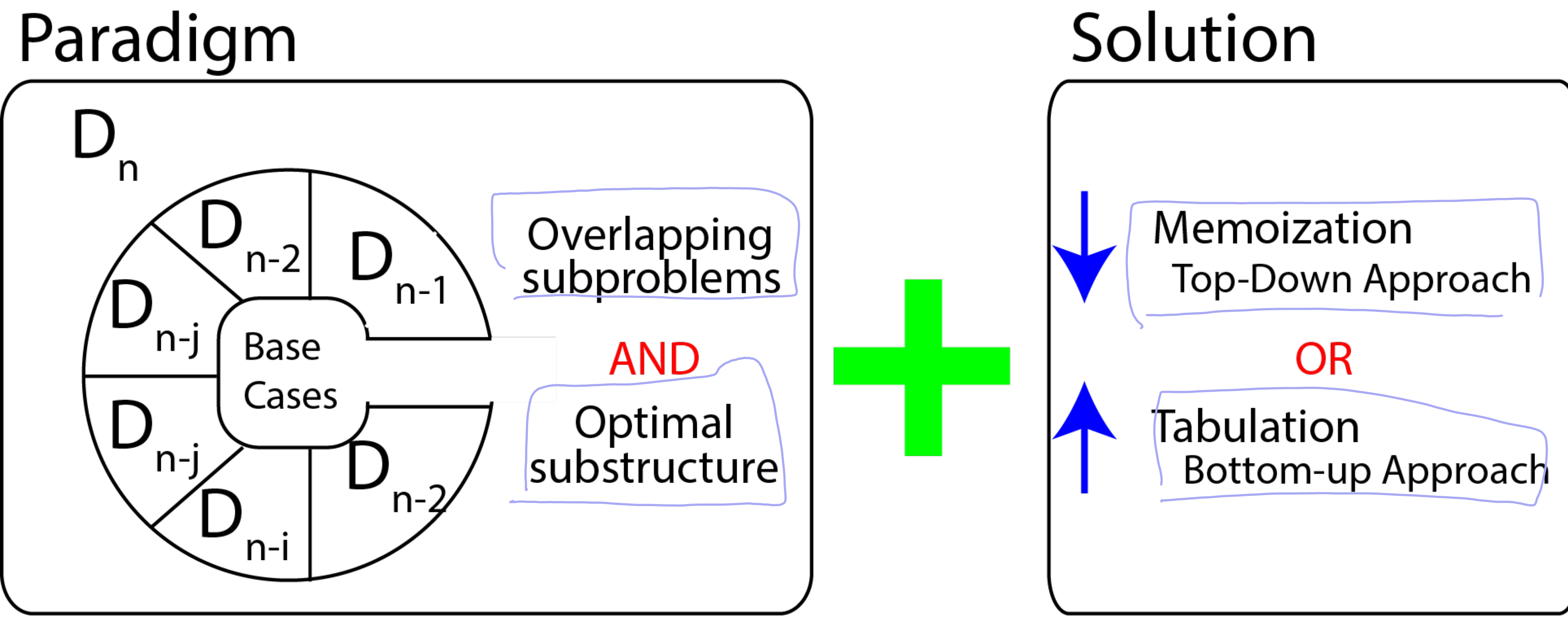


1. Identify the Complete Search recurrence.
2. Initialize some parts of the DP table with known initial values.
3. Determine how to fill the rest of the DP table based on the Complete Search recurrence, usually involving one or more nested loops to do so.

DP - Top-Down VS Bottom-Up

Top-Down	Bottom-Up
Pro: 1. It is a natural transformation form normal complete search recursion. 2. Compute sub-problems only when necessary.	Pro: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls. 2. Can save memory space with DP "on-the-fly" technique.
Cons: 1. Slower if many sub-problems are revisited due to recursive calls overhead. 2. If there are *M* states, it can use up to O(M) table size which can lead to memory problems.	Cons: 1. For some programmers who are inclined with recursion, this may be not intuitive. 2. If there are *M* states, bottom-up DP visits and fills the value of *all* these M states.

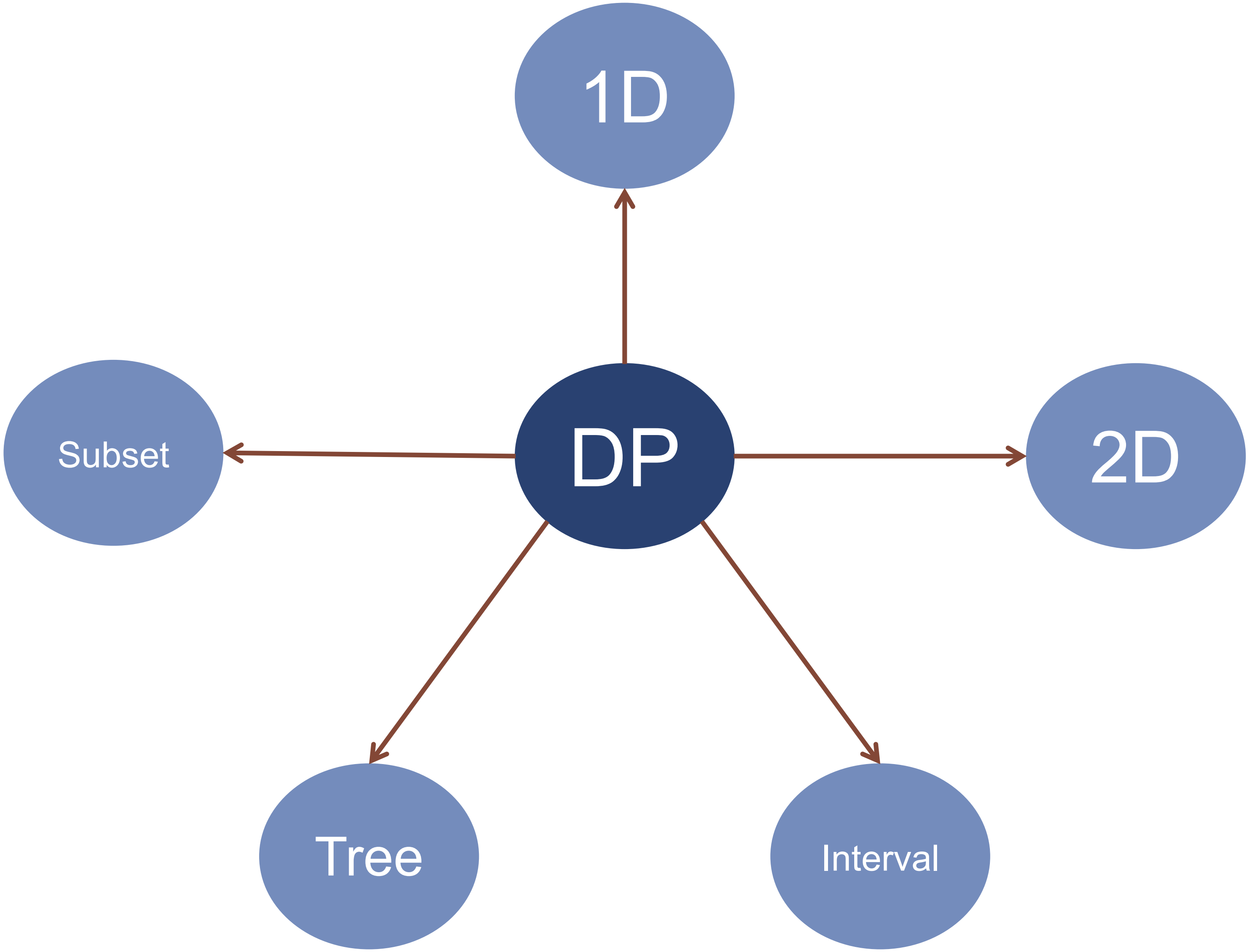
DP – Take home picture



when to use

methods to solve

DP – Examples



DP – 1-dimensional

Problem: Given *n*, find the number of different ways to write **n** as the sum of the numbers 1, 3, 4.

Example: for n = 5, the answer is 6

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

DP – 1-dimensional

Step 1: Identify the sub-problems (in words).

Step 1.1: Identify the possible sub-problems.

Let $D_{n\text{-}i}$ be the number of ways to write *n-i* as the sum of 1, 3, 4.

$D_5 = \underbrace{\boxed{1+1+1+1}}_{D_4} + 1$

$D_5 = \underbrace{\boxed{1+1}}_{D_2} + 3$

$D_5 = \underbrace{\boxed{1+3}}_{D_4} + 1$

$D_5 = \underbrace{\boxed{3+1}}_{D_4} + 1$

$D_5 = \underbrace{\boxed{1}}_{D_1} + 4$

$D_5 = \underbrace{\boxed{4}}_{D_4} + 1$

DP – 1-dimensional

Step 2: Find the recurrence.

Step 2.1: What decision do I make at every step?.

- Consider one possible solution $n = x_1 + x_2 + \ldots + x_m$
- 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}

$D_5 = \boxed{1 + 1 + 1 + 1} + 1$

$D_5 = \boxed{1 + 3} + 1$

$D_5 = \boxed{3 + 1} + 1$

$D_5 = \boxed{4} + 1$

D_4 subproblem D_{N-1}

$D_5 = \boxed{1 + 1} + 3$ n

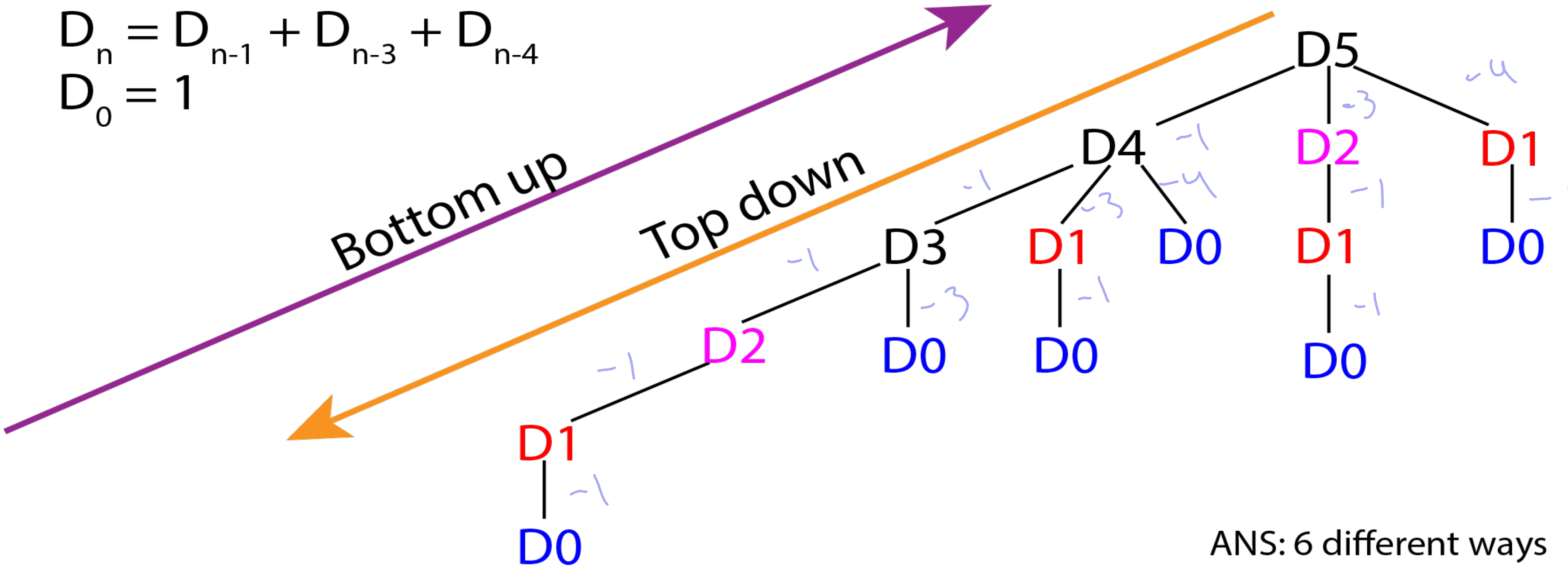
D_2 D_{N-3}

$D_5 = \boxed{1} + 4$

D_1 D_{N-4}

DP – 1-dimensional

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$
$$D_0 = 1$$



ANS: 6 different ways

Key ingredients to make DP works

1. This problem has optimal sub-structures.
   - Solution for the sub-problem is part of the solution of the original problem.
2. This problem has overlapping sub-problems.

DP – 1-dimensional

Step 3: Recognize and solve the base cases.

- $D_0 = 1$, and $D_n = 0$ for all $n < 0$.
- Alternatively, can set: $D_0 = D_1 = D_2 = 1$, and $D_3 = 2$

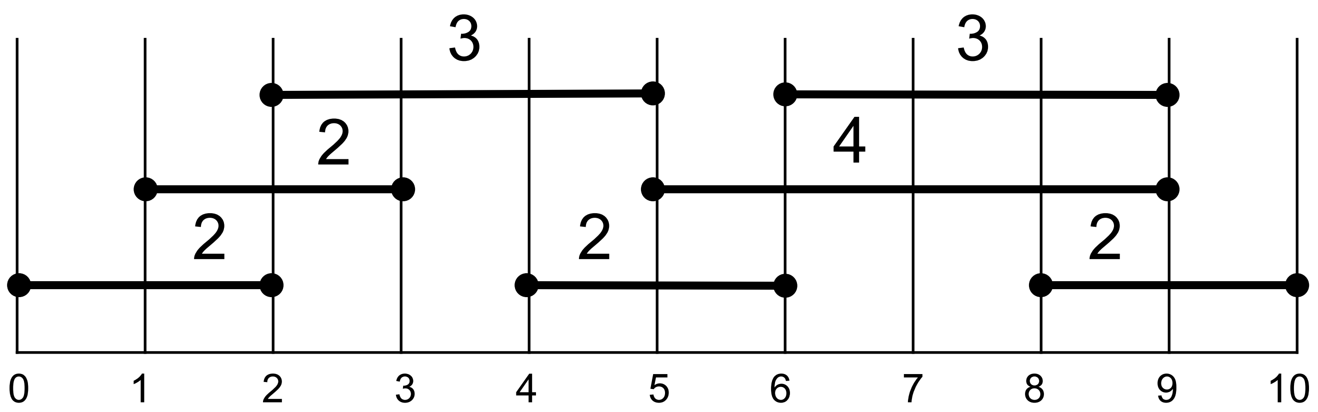
Step 4: Implement a solving methodology.

```
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];
```

all the ways to create this

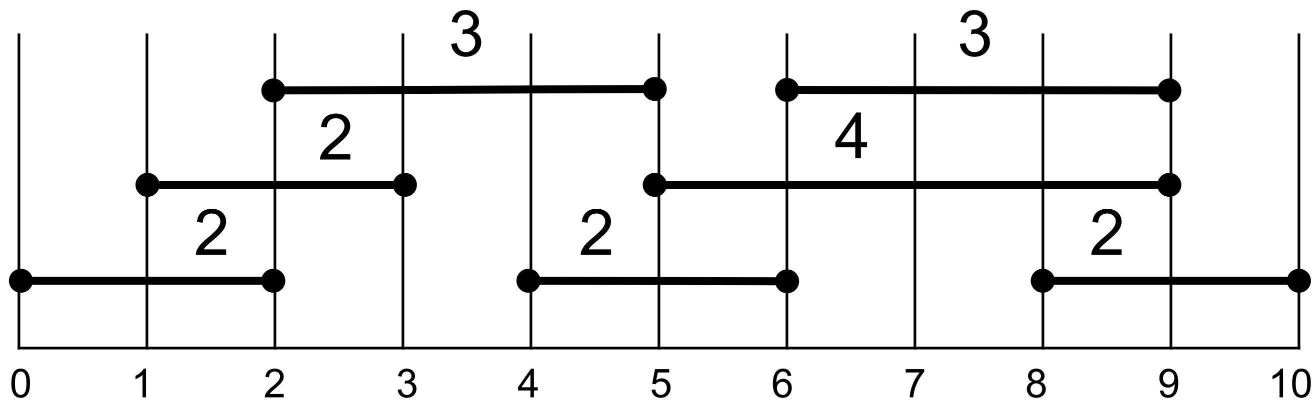
DP – 1-dimensional – weighted interval scheduling

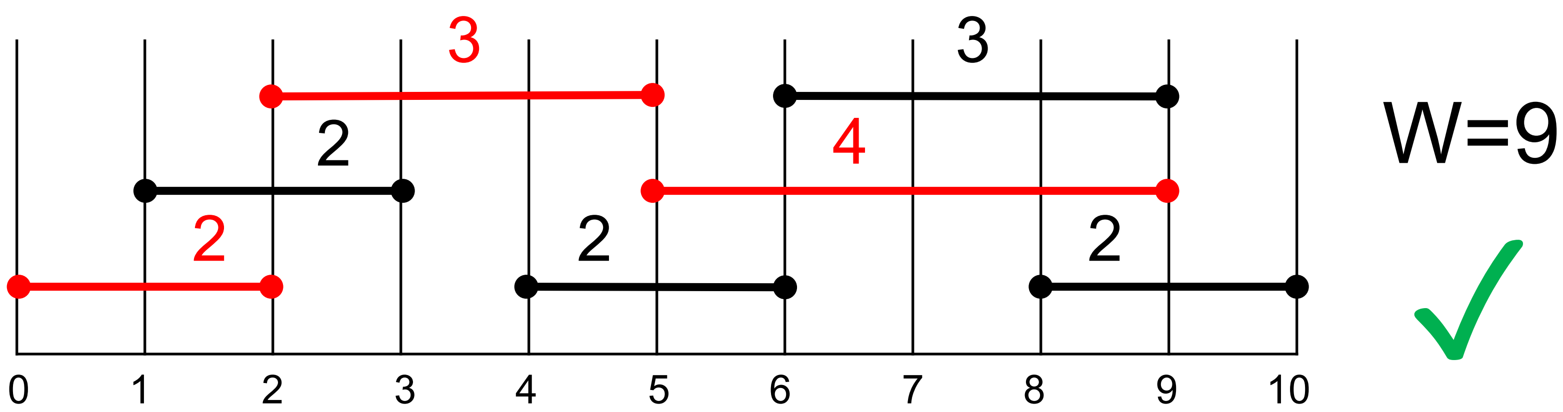
- **Input:** Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - s_i = start time of activity i.
  - f_i = finish time of activity i.
  - w_i= weight of activity i
- **Output:** find maximum weight subset of mutually compatible activities.
  - 2 activities are compatible, if their intervals do not overlap.



DP – 1-dimensional – weighted interval scheduling

Example:



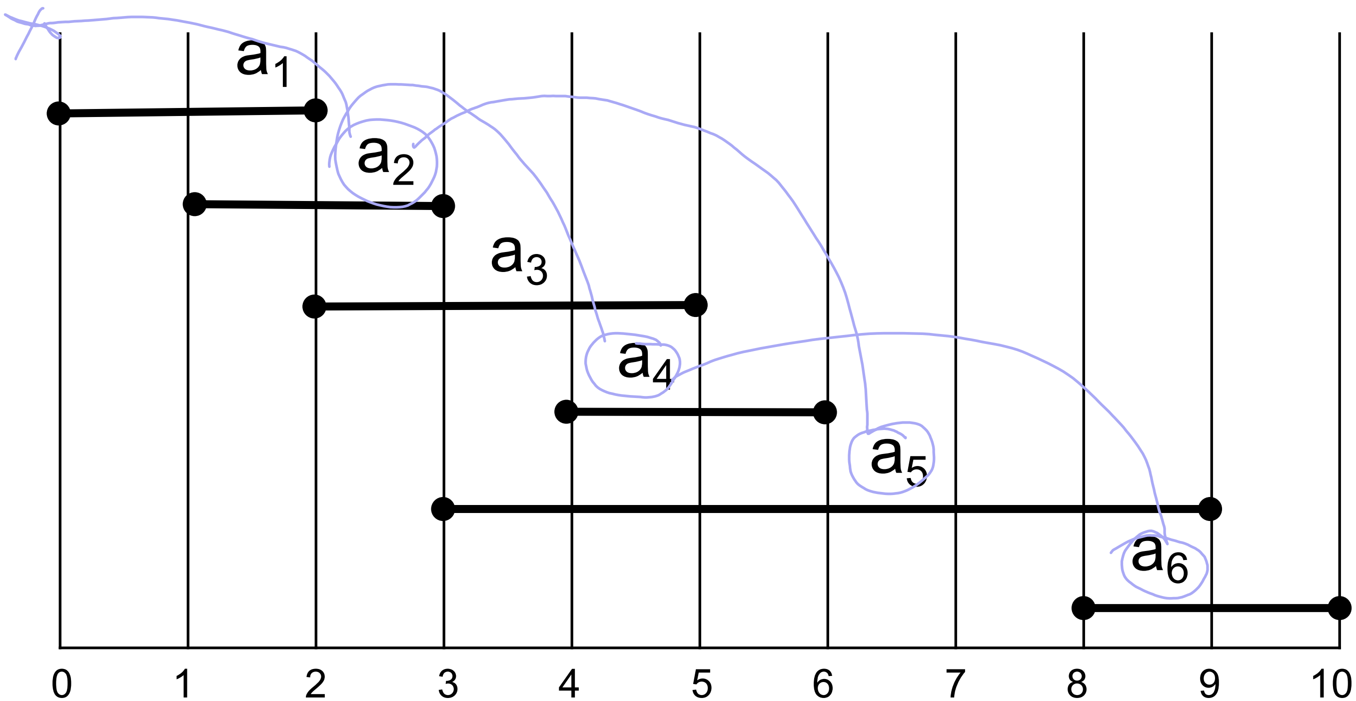


weighted interval scheduling – Data Structure

Notation: All activities are sorted by finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$

Definition: $p(j)$ = largest index $i < j$ such that activity/job i is compatible with activity/job j.

Examples: $p(6)=4$, $p(5)=2$, $p(4)=2$, $p(2)=0$.

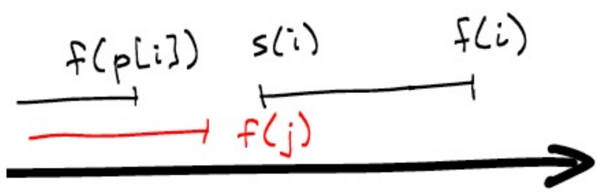


weighted interval scheduling – Data Structure

Let $S(i)$ be a set of intervals in maximal solution of the problem when we can use only intervals $\{1, 2, \ldots i\}$.

activity between i and predecessor

Claim: If $i \in S(i)$ then $S(i)$ cannot contain j where $p[i] < j < i$.



Proof:

To have $p[i] < j < i$, we would need $f(p[i]) < f(j)$ and $f(j) < s(i)$.

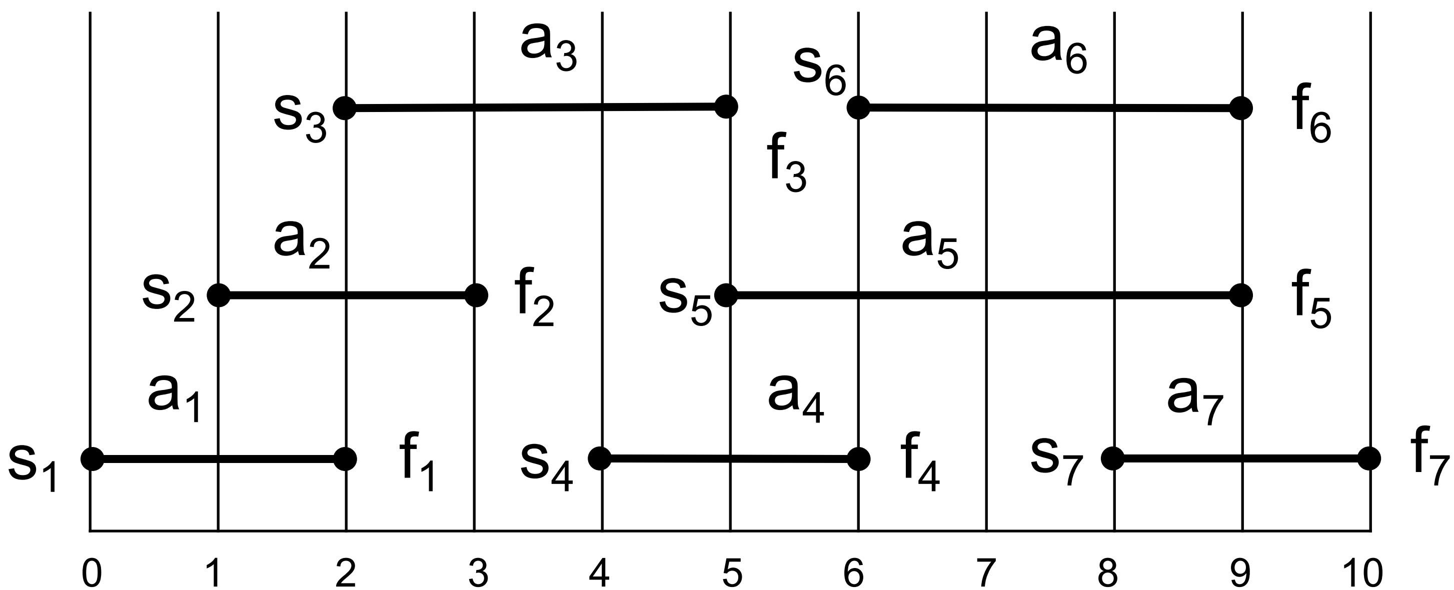
But this would contradict the definition of $p[i]$.

weighted interval scheduling

Step 1: Identify the sub-problems (in words).

Step 1.1: Identify the possible sub-problems.

Let OPT(j) be the maximum total weight of compatible activities 1 to j (i.e., value of the optimal solution to the problem including activities 1 to j).



weighted interval scheduling

$\boxed{\text{Step 2:}}$ Find the recurrence.

Step 2.1: What decision do I make at every step?.

Case 1: OPT selects activity j

- Add weight w_j
- Cannot use incompatible activities - subset is only compatible
- Must include optimal solution on remaining compatible activities { 1, 2, … , p(j) }.

Optimal substructure property

? > ?

Case 2: OPT does not select activity j

- Must include optimal solution on other activities { 1, 2, …, j-1 }.

up to j

weighted interval scheduling

Step 2: Find the recurrence.

- **Case 1:** OPT selects activity j
  - Add weight w_j
  - Cannot use incompatible activities
  - Must include optimal solution on remaining compatible activities { 1, 2, … , p(j) }.
- **Case 2:** OPT does not select activity j
  - Must include optimal solution on other activities { 1, 2, …, j-1 }.

case 1 case 2

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ \underline{max}\{ w_j + OPT(p(j)), OPT(j-1) \} & Otherwise \end{cases}$$

weighted interval scheduling

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ max\{w_j + OPT(p(j)), OPT(j-1)\} & Otherwise \end{cases}$$

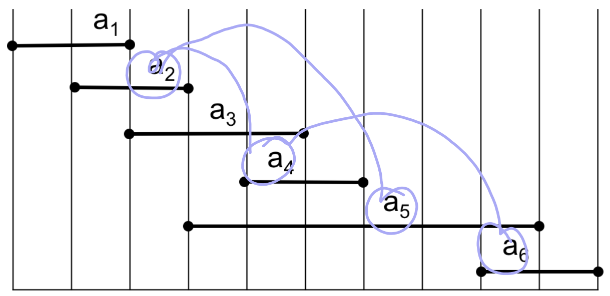
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2] ≤ ... ≤ f[n].

Compute p[1], p[2], ..., p[n].
```

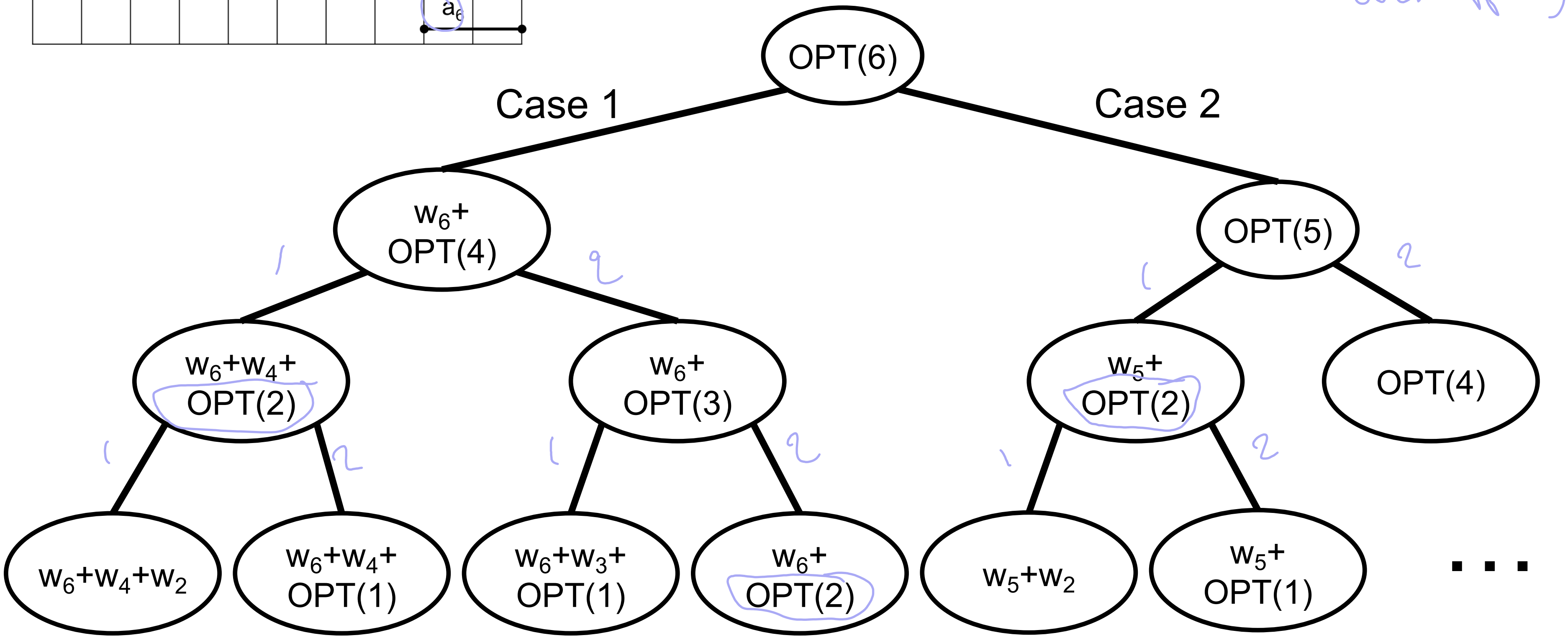
```
Compute-Opt(j)
if j = 0        base case
   return 0.
else
   return max(v[j] + Compute-Opt(p[j]), Compute-Opt(j–1)).
```

weighted interval scheduling – recursion tree



Observation: OPT(j) is calculated multiple times …

overlapping



weighted interval scheduling – top down

Memoization: Cache results of each subproblem; lookup as needed.

Lremember if you computed something already

```
Input: n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1]≤f[2]≤ ... ≤f[n].
Compute p[1], p[2], ..., p[n].

for j = 1 to n
      M[j] ← empty.
M[0] ← 0.
```

Storing subproblems

L asking if already computed

```
M-Compute-Opt(j)
if M[j] is empty
      M[j] ← max(v[j]+M-Compute-Opt(p[j]),M-Compute-Opt(j–1))
return M[j].
```

weighted interval scheduling – bottom-up

Observation: When we compute M[j], we only need values M[k] for k<j.

```
Input: n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1]≤f[2]≤ ... ≤f[n].
Compute p[1], p[2], ..., p[n].

M[0] ← 0.
for j = 1 TO n
      M[j] ← max { vj + M[p(j)], M[j–1] }
```

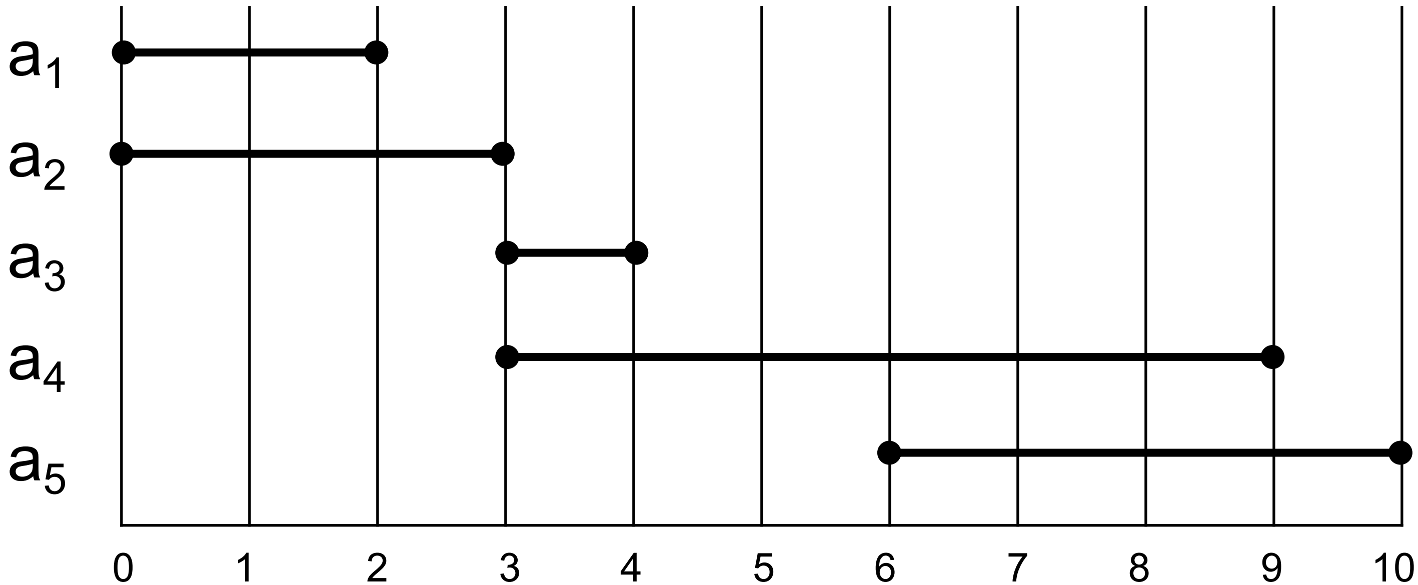
no recursion

Main Idea of Dynamic Programming: Solve the sub-problems in an order that makes sure when you need an answer, it's already been computed.

weighted interval scheduling

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	-	-	-	-	-
V_j+M[p(j)]	-	-	-	-	-
M[j-1]	-	-	-	-	-

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



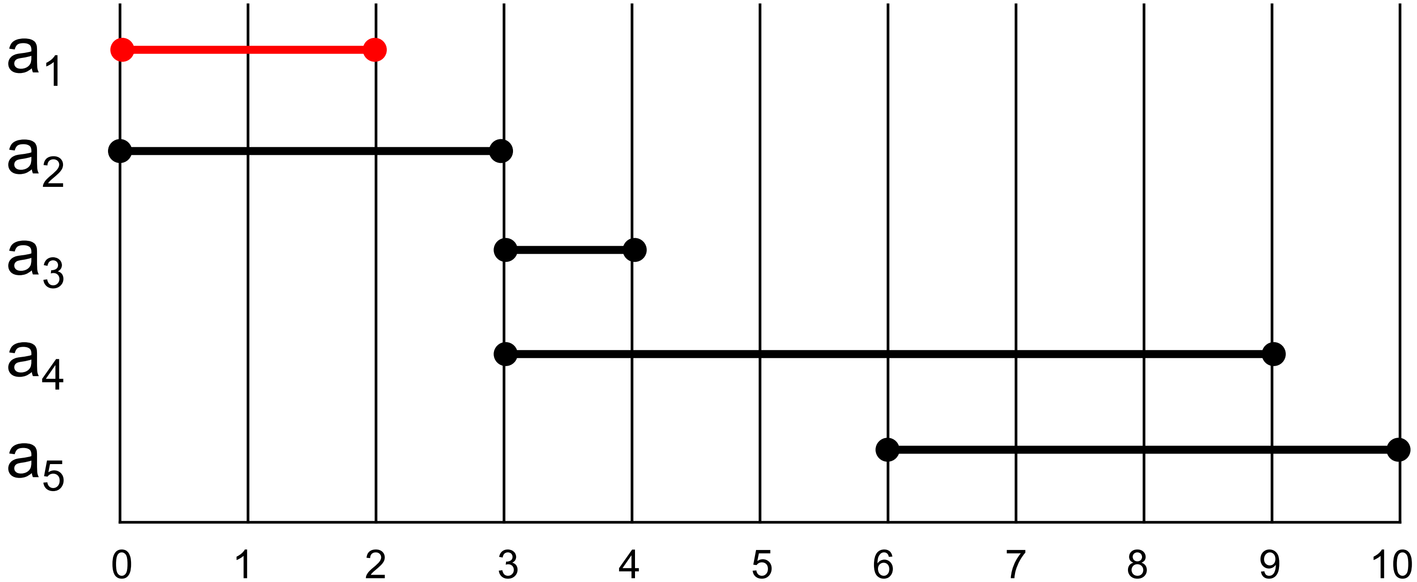
weighted interval scheduling

M[0]=0

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	-	-	-	-
V_j+M[p(j)]	**2**	-	-	-	-
M[j-1]	0	-	-	-	-

choose one

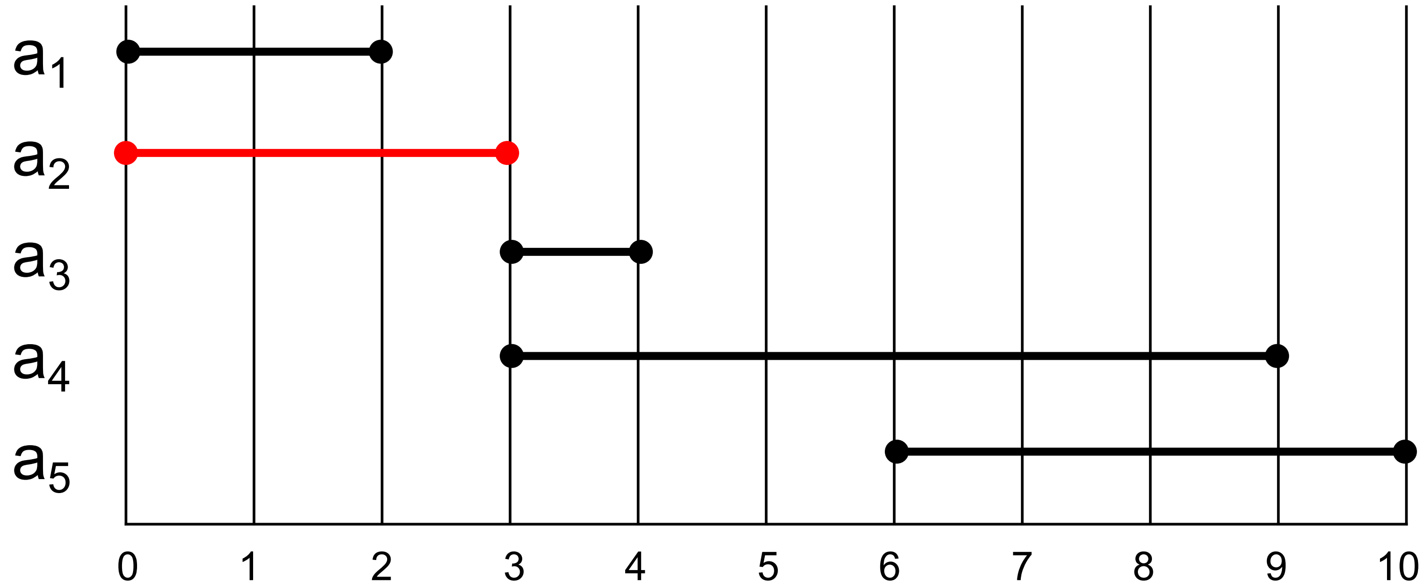
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



weighted interval scheduling

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	-	-	-
V_j+M[p(j)]	2	**3**	-	-	-
M[j-1]	0	2	-	-	-

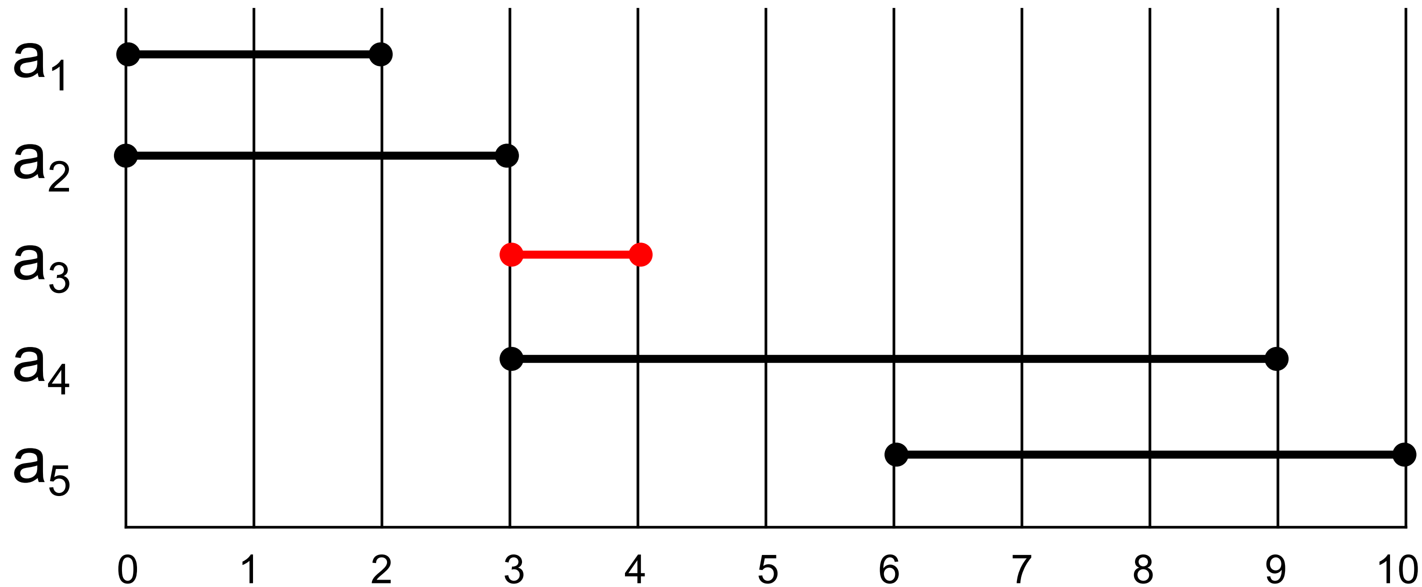
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



weighted interval scheduling

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4 4vs 3	-	-
V_j+M[p(j)]	2	3	**4**	-	-
M[j-1]	0	2	3	-	-

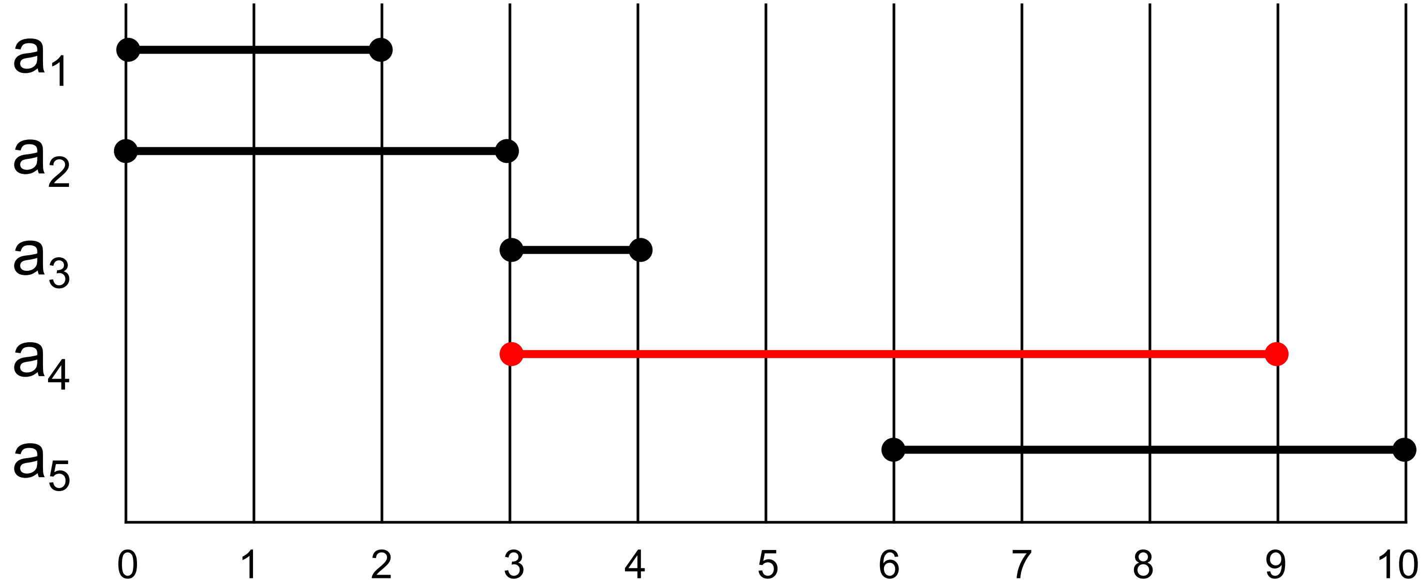
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



weighted interval scheduling

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	-
V_j+M[p(j)]	2	3	4	**9**	-
M[j-1]	0	2	3	4	-

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



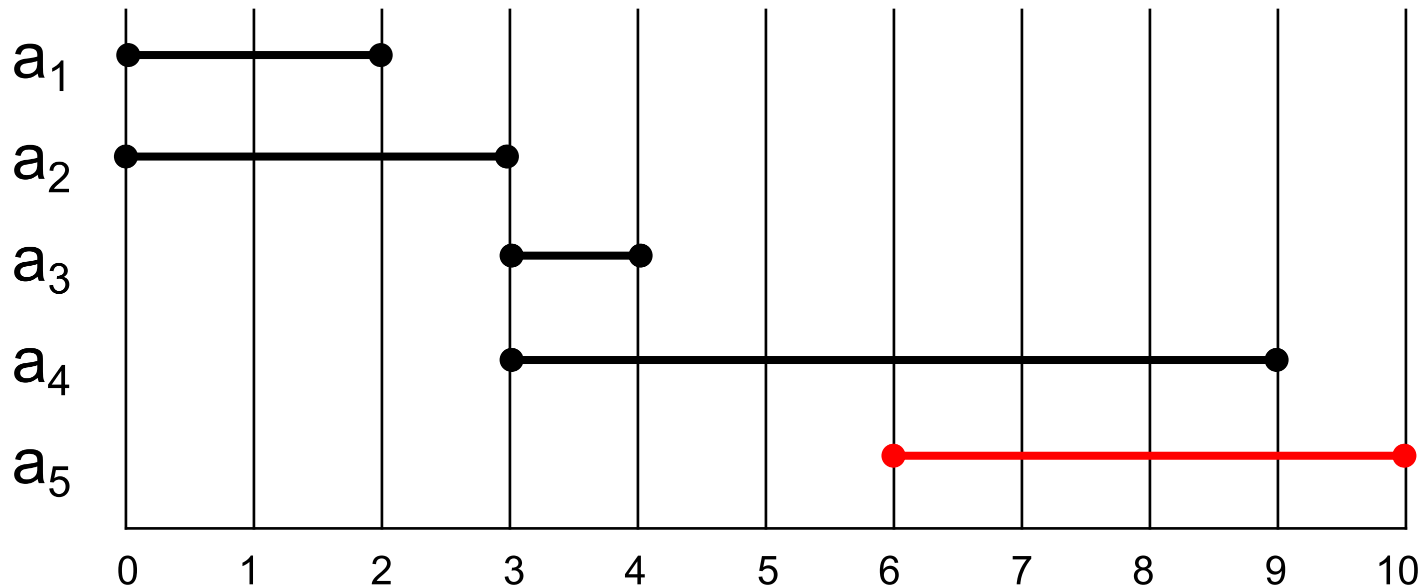
weighted interval scheduling

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
V_j+M[p(j)]	2	3	4	9	8
M[j-1]	0	2	3	4	9

Your solution

max 9, 8

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



weighted interval scheduling – finding tasks

Dyn. Prog. algorithm computes optimal value.

Q: How to find solution itself?
A: Bactrack!

```
Find-Solution(j)
if j = 0
     return Ø.
else if (v[j] + M[p[j]] > M[j–1])
     return { j } ∪ Find-Solution(p[j])
else
     return Find-Solution(j–1).
```

Analysis. # of recursive calls ≤ n ⇒ $O(n)$. – worst case

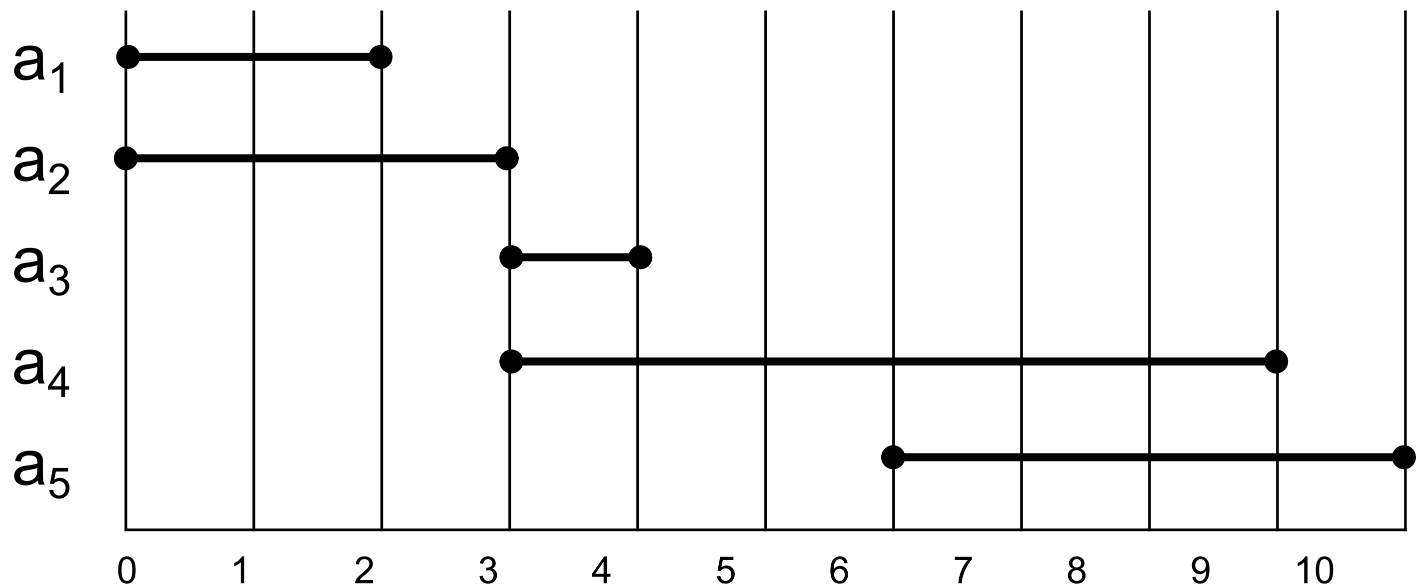
weighted interval scheduling - reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
V_j+M[p(j)]	2	3	4	9	8
M[j-1]	0	2	3	4	9

max(8,9)

didn't select 5

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.

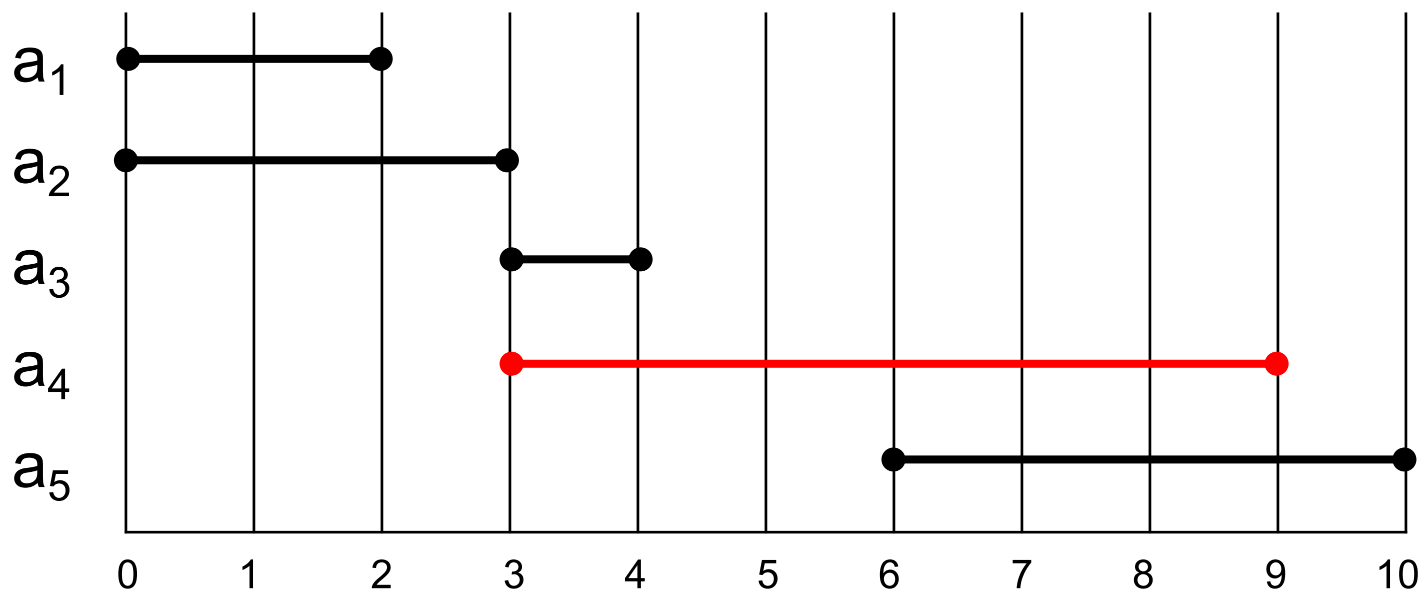


weighted interval scheduling - reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
V_j+M[p(j)]	2	3	4	9	8
M[j-1]	0	2	3	~~4~~	9

max 9,4

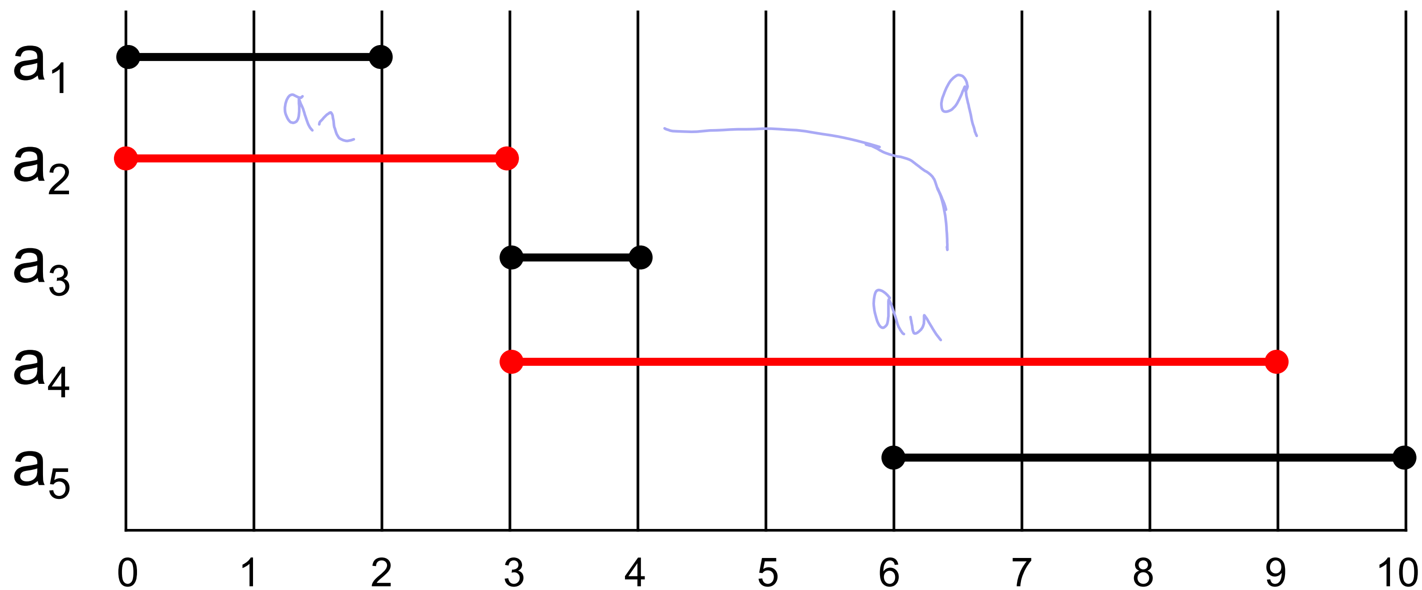
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



weighted interval scheduling - reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
V_j+M[p(j)]	2	3	4	9	8
M[j-1]	0	2	3	4	9

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



weighted interval scheduling – running time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via binary search

- M-COMPUTE-OPT(j): each invocation takes $O(1)$ time and either
  - (i) returns an existing value M[j]
  - (ii) fills in one new entry M[j] and makes two recursive calls

- Progress measure $\Phi = \#$ nonempty entries of M[].
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by $1 \Rightarrow$ at most $2n$ recursive calls.

- Overall running time of M-COMPUTE-OPT(n) is $O(n)$. ▪

Remark. $O(n)$ if jobs are presorted by start and finish times.

Outline

- Complete Search
- Divide and Conquer.
- Dynamic Programming.
    - Introduction.
    - Examples.
- Greedy.


time for questions