Dynamic Programming

Tiago Januario – Boston University

Agenda

- Contextualization of Dynamic Programming
- Main features
  - Subproblem overlapping
  - Principle of optimality
- Approaches
  - Memoization (Top-Down)
  - Tab (Bottom-up)

Context
Dynamic programming

- It is a powerful algorithm design technique

Context
Dynamic programming

- It is a powerful algorithm design technique
- Two perspectives on PD:
  - DP ≈ "careful brute force"
  - Using intelligently, one can reduce "exponential" problems to polynomials

Context
Dynamic programming

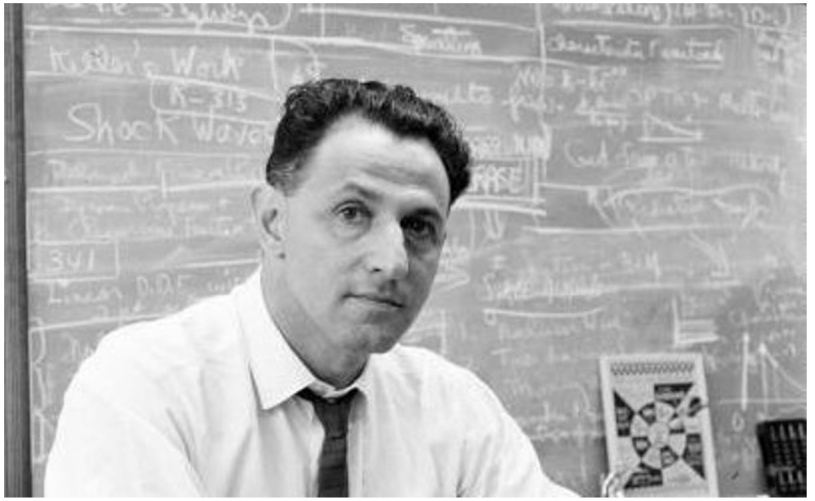
- It is a powerful algorithm design technique
- Two perspectives on PD:
  - DP ≈ "careful brute force”
  - Using intelligently, one can reduce "exponential" problems to polynomials
  - DP ≈ Recursion + "reuse"
  - We will be more precise throughout the class

Context
Dynamic programming

- Dynamic Programming?

Bellman, (1984) p. 159 explained that he invented the name “dynamic programming” to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who “had a pathological fear and hatred of the term, research.” He settled on “dynamic programming” because it would be difficult give it a “pejorative meaning” and because “It was something not even a Congressman could object to.

[John Rust 2006]
[https://editorialexpress.com/jrust/research/papers/dp.pdf]



Dr Richard Bellman

IEEE 1979 Medal

Contexto

Programação dinâmica

Dynamic Programming (DP)

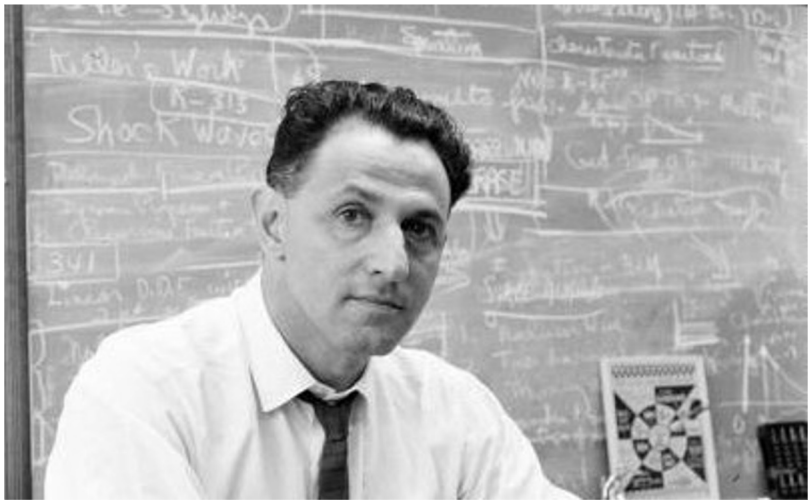
- Dynamic Programming?

Something related to optimization

Something that won't give you problems

Bellman, (1984) p. 159 explained that he invented the name “dynamic programming” to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who “had a pathological fear and hatred of the term, research.” He settled on “dynamic programming” because it would be difficult give it a “pejorative meaning” and because “It was something not even a Congressman could object to.

[John Rust 2006]
[https://editorialexpress.com/jrust/research/papers/dp.pdf]



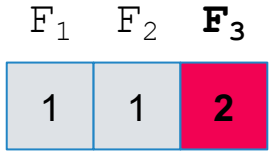
Dr Richard Bellman

Dynamic programming

- Features
  - Overlapping problems (??)
  - Principle of optimality (??)

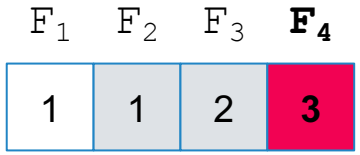
Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$



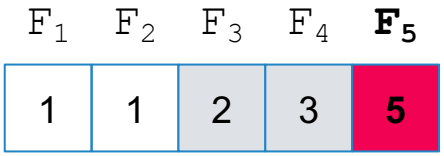
Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$



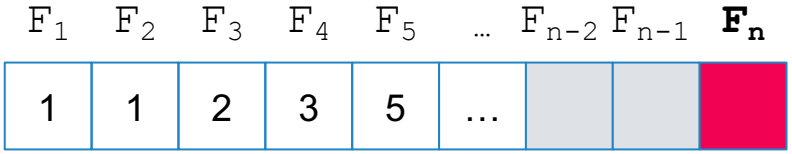
Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$



Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$
- Goal:
  - Compute F_n



Fibonacci sequence
Naive solution

```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

Fibonacci sequence
Naive solution

```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

Fibonacci sequence
Naive solution

```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

Fibonacci sequence
Naive solution

```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

- Does the algorithm work?
- Is it a good algorithm?

Fibonacci sequence
Naive solution

```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

- Does the algorithm work?
  - Yes!
- Is it a good algorithm?
  - No!
  - Exponential time!!!

Fibonacci sequence
Naive solution

```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Fibonacci sequence
Naive solution

```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

$$T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n$$

Fibonacci sequence
Naive solution

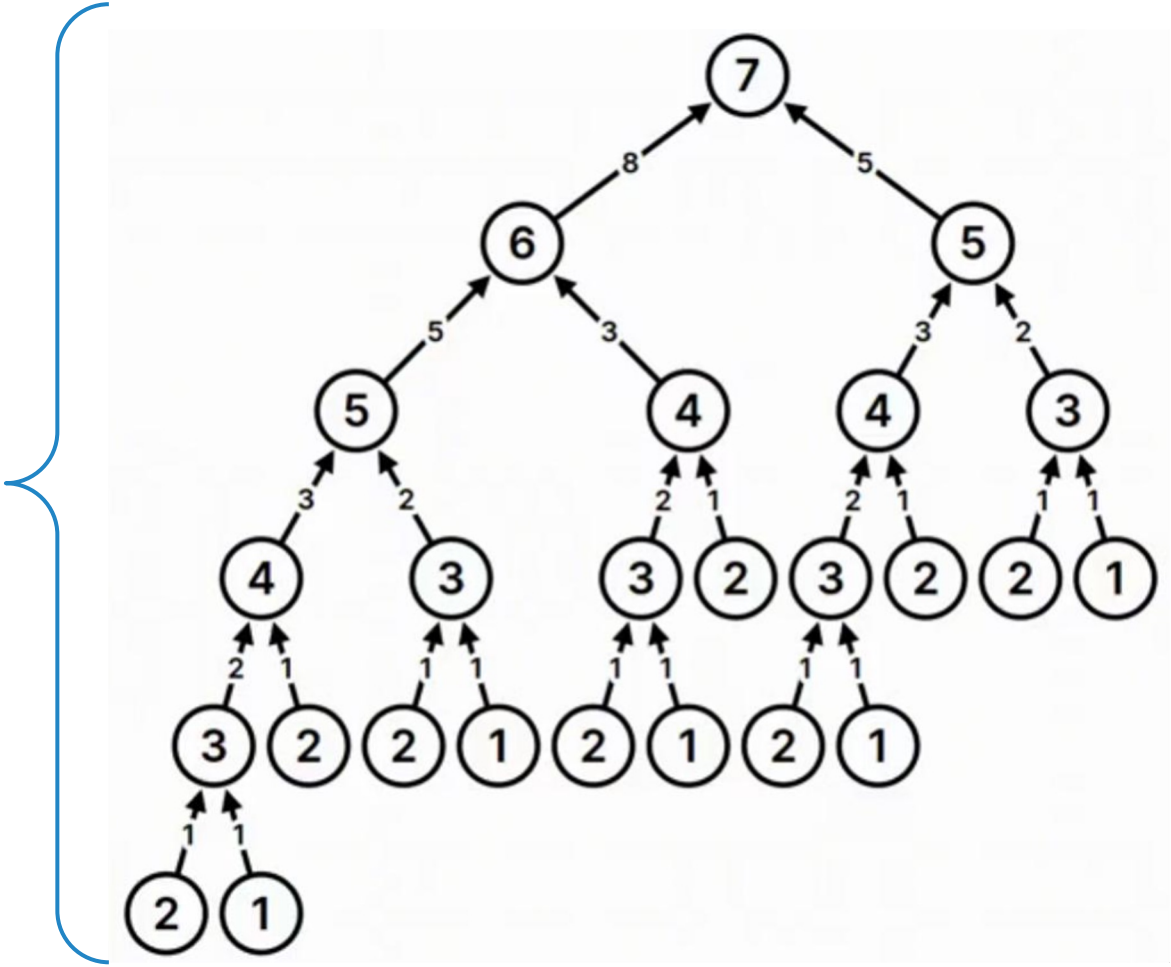
```
1.def fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + fib(n-2)
6.   return f
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n \\
&\geq 2T(n-2) + O(1) \\
&\geq 2^{n/2}
\end{aligned}
$$

« ‹ ›

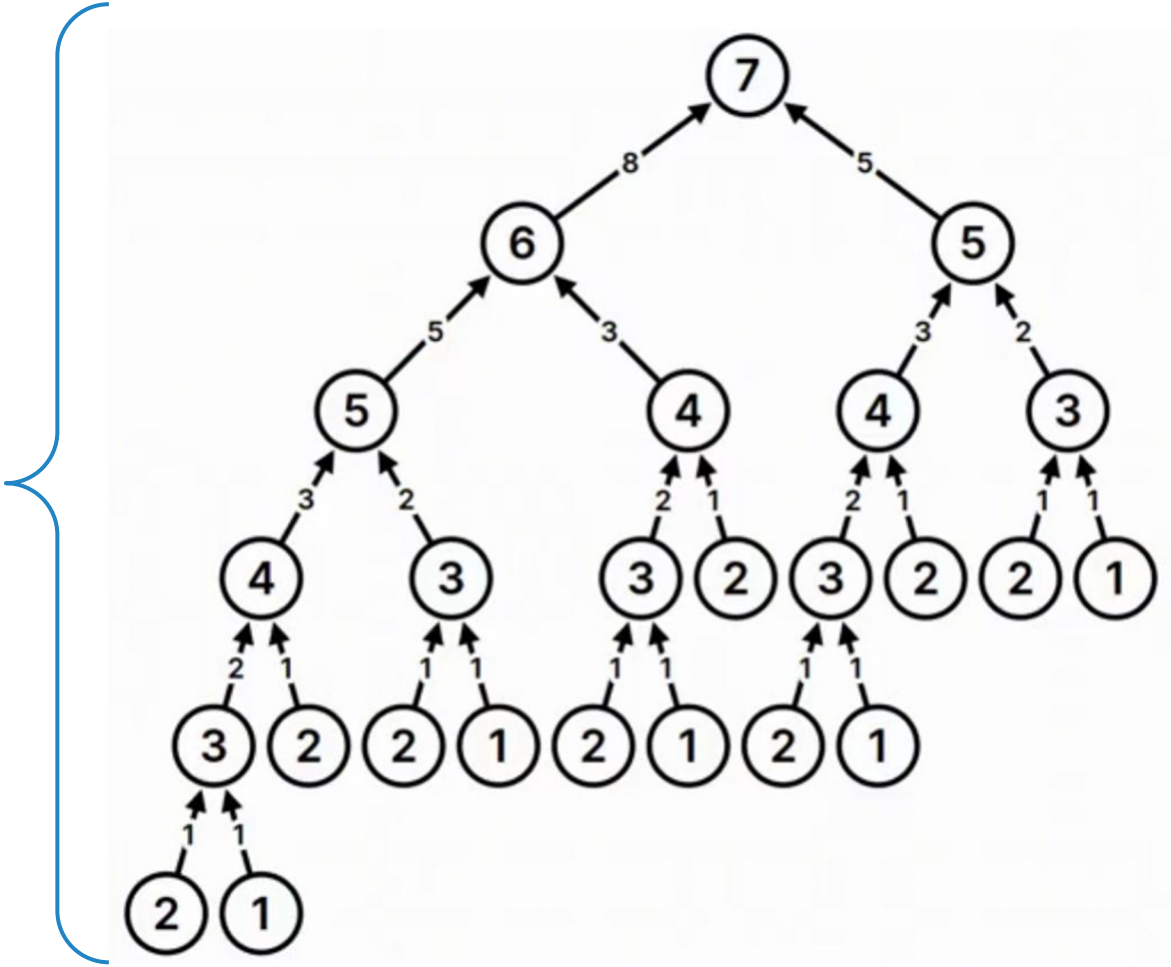
fn(7) starts running

Time ≈ # calls ≈ nodes



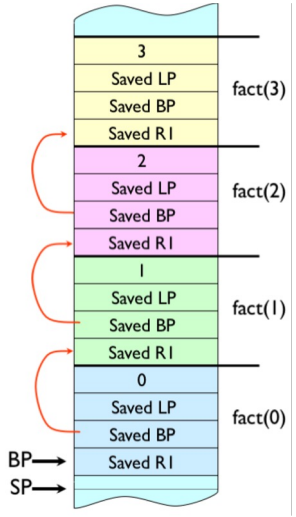
Time ≈ # calls ≈ nodes

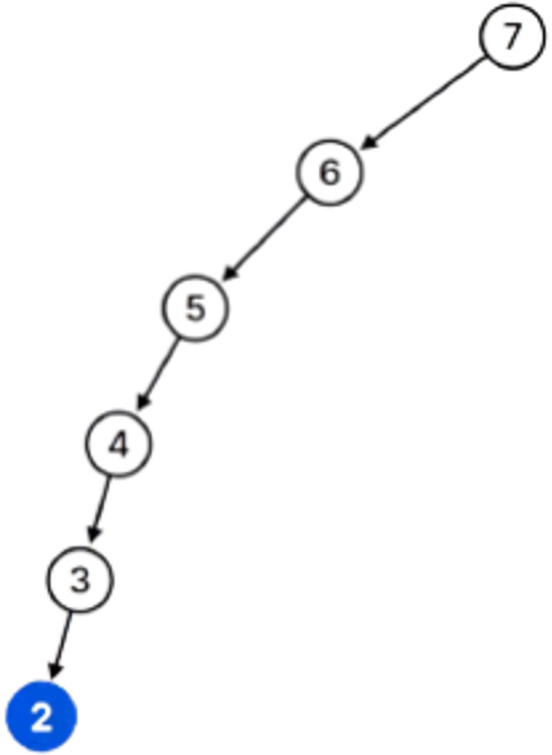
Space ≈ size of the longest path (root, leaf)



Fibonacci sequence
Naive solution

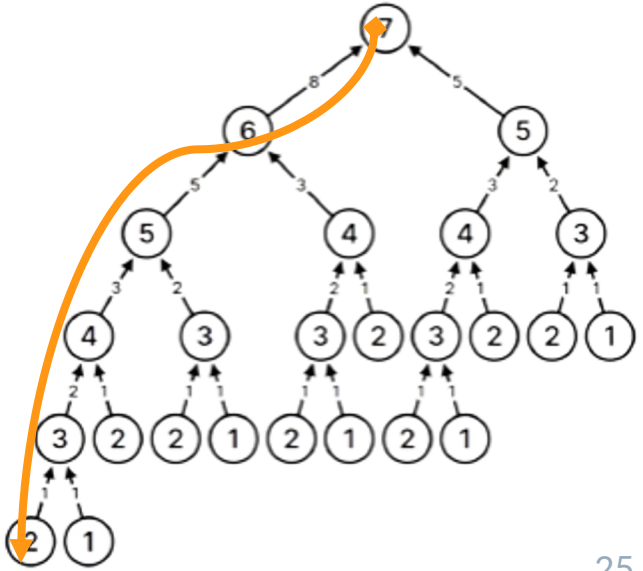
- We make n calls
- Calls are stored in the activation stack





Fibonacci sequence
Naive solution

```
1.fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + f(n-2)
6.   return f
```



Fibonacci sequence
Naive solution

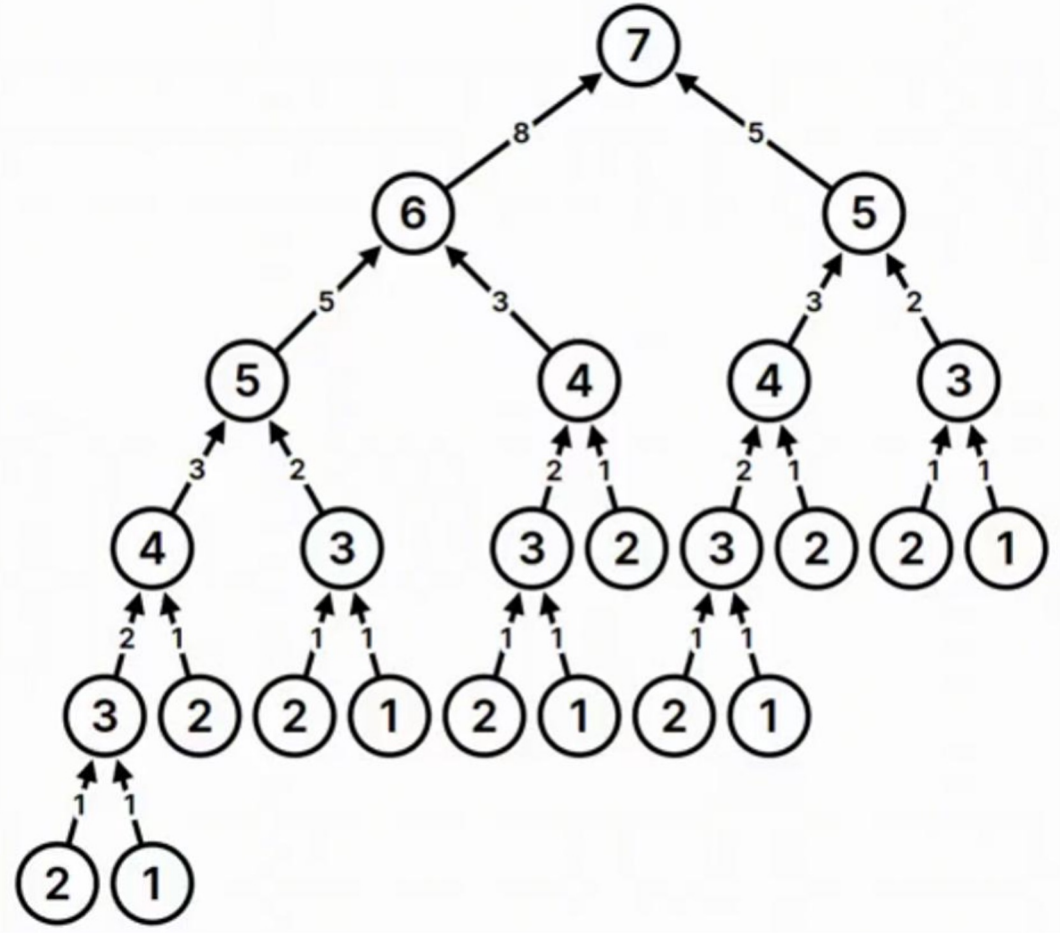
```
1.fib(n):
2.   if n <= 2:
3.       f = 1
4.   else:
5.       f = fib(n-1) + f(n-2)
6.   return f
```

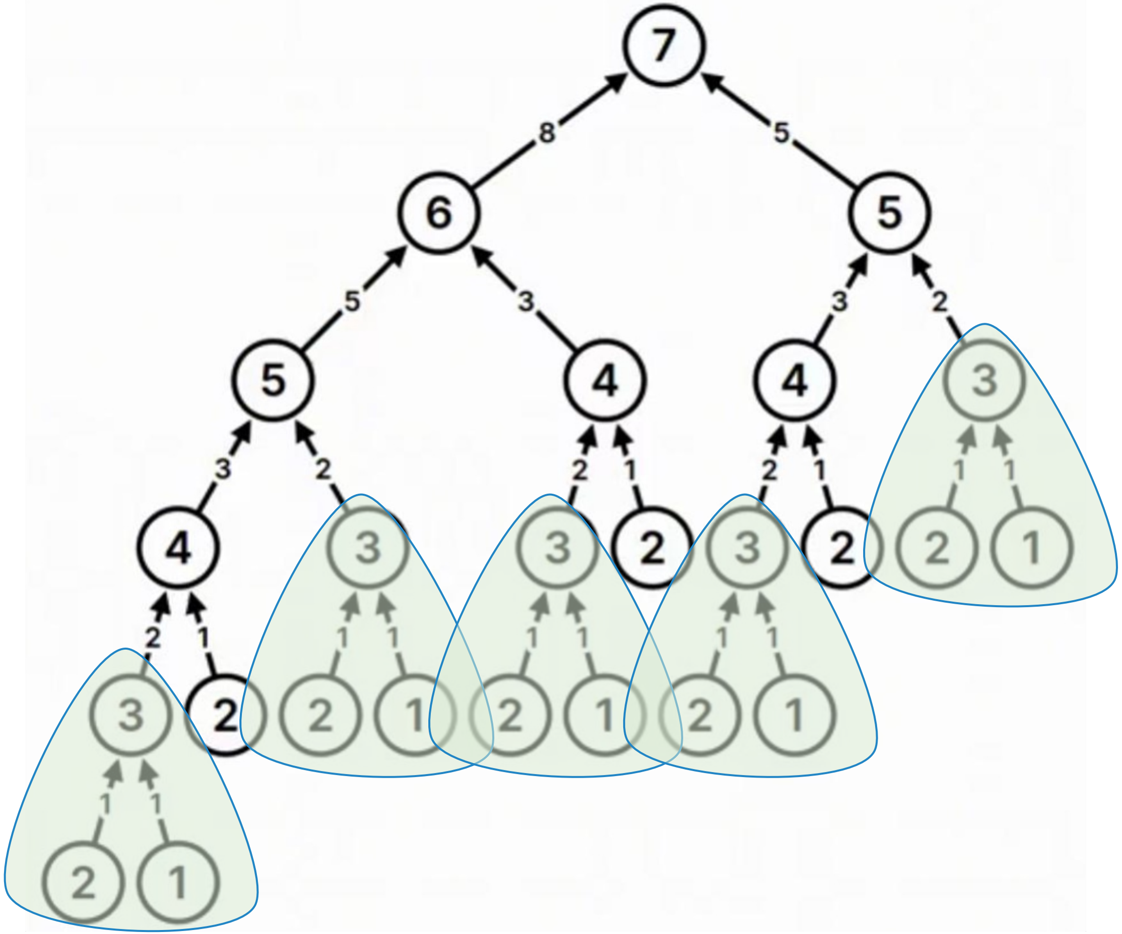
Time $O(2^{n/2})$

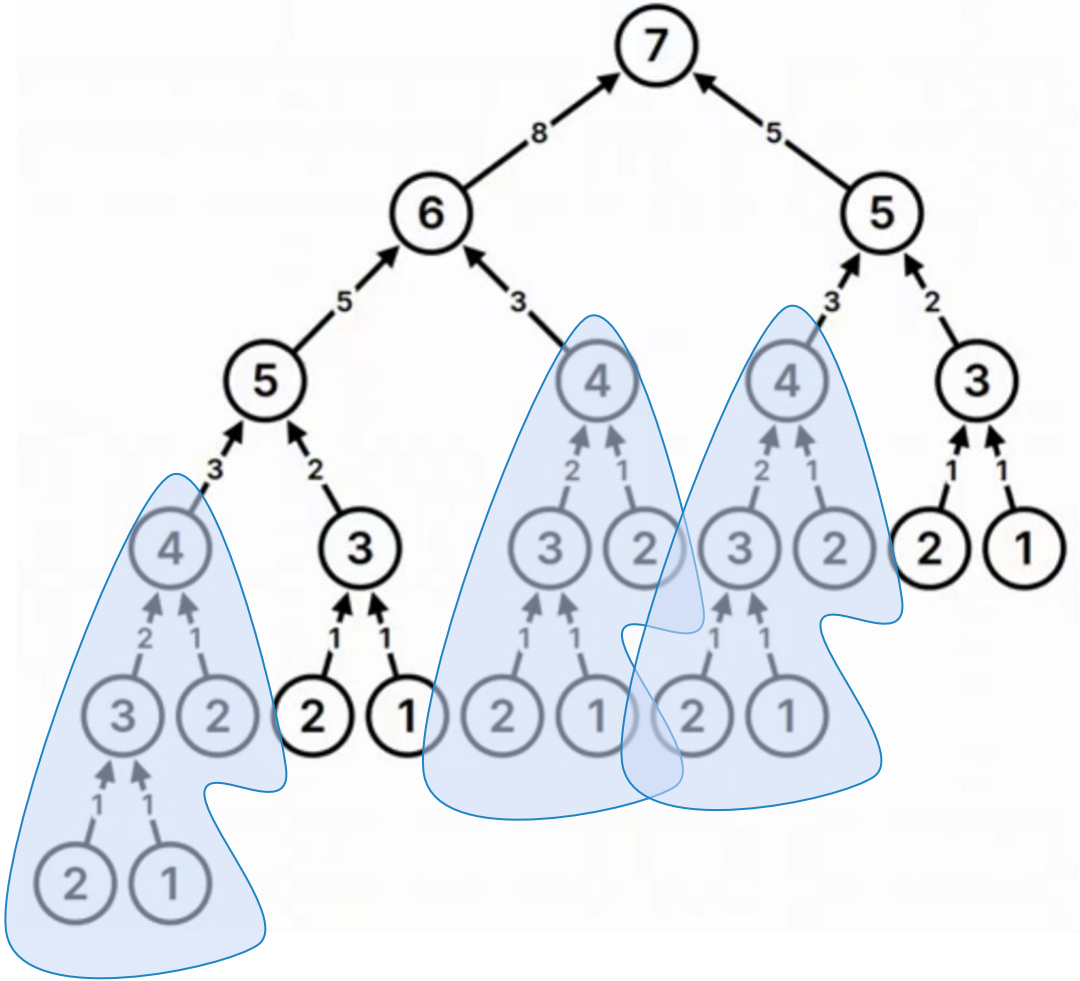
fib(50) ≈ 2^{50} steps

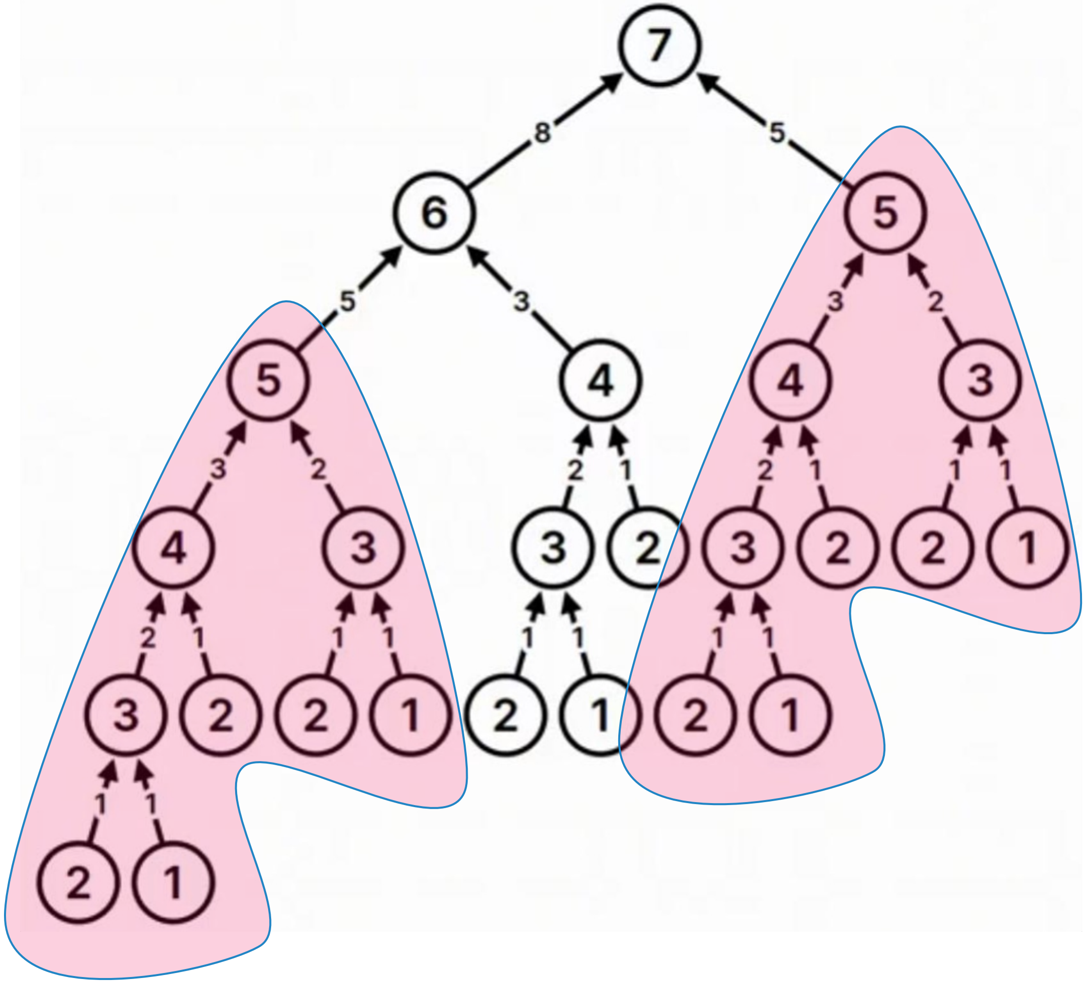
1.12e+15 =

1.125.899.906.842.624


7
8
5
6
5
5
3
3
2
5
4
4
3
3
2
1
2
1
2
1
4
3
3
2
3
2
2
1
2
1
1
1
1
1
1
1
1
1
3
2
2
1
2
1
2
1
1
1
1
1
2
1



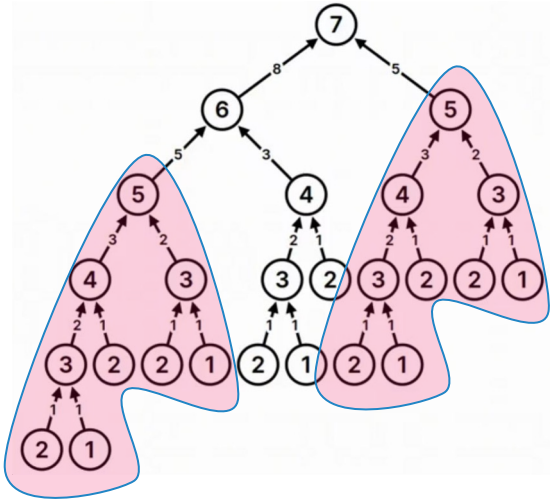

7
8
5
6
5
5
3
3
2
4
3
4
3
3
2
2
1
4
3
2
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1



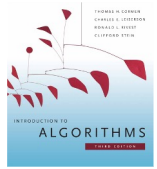
7
8
5
6
5
5
3
3
2
4
4
3
3
2
2
1
1
4
3
3
2
2
1
2
1
1
1
1
1
1
1
1
1
1
1
3
2
2
1
3
2
2
1
2
1
2
1
2
1

Dynamic programming

- Features
- Overlapping problems (✅)
- Principle of optimality (??)

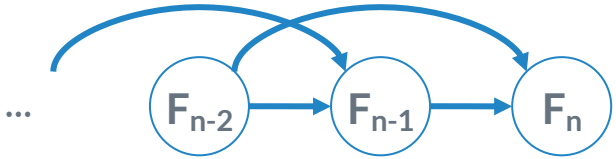


The principle of optimality



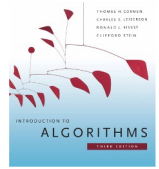
§15 e 27

- Optimal substructure:
  - "A problem has optimal substructure if the optimal solution can be built from optimal solutions to its subproblems."



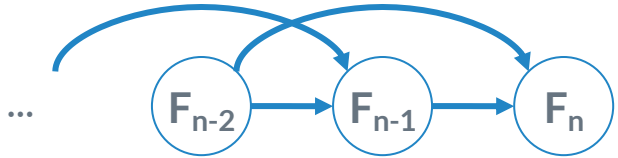
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

The principle of optimality



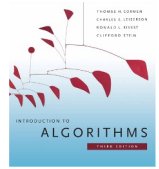
§15 e 27

- Optimal substructure:
  - "A problem has optimal substructure if the optimal solution can be built from optimal solutions to its subproblems."
- In other words:
  - We can solve bigger problems using smaller instance solutions of the same problem!



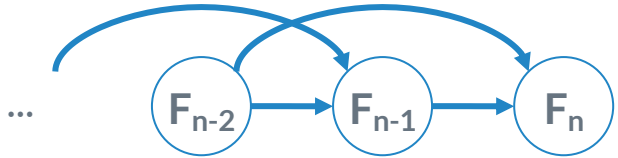
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

The principle of optimality



§15 e 27

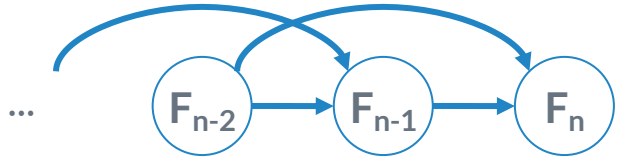
- Dependence on subproblems
  - Must form DAG (Directed Acyclic Graph)
  - If it has cycles, the PD algorithm can execute infinitely



Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

Dynamic programming

- Features
- Overlapping problems (✅)
- Principle of optimality (✅)
- The dependencies of the subproblems must be acyclic (DAG!)



Why?

Dynamic programming

- By using smartly one can reduce "exponential" problems to polynomials

How?

Prob. must have 2 characteristics

- *Overlapping problems* (✅)
- *Principle of optimality* (✅)

What?

Fibonacci sequence Problem

- $F_n = F_{n-1} + F_{n-2}$

Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions

Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions
  - Maintains a "dictionary"
  - Subproblems → solutions

```
memo {
 Subp1: val1,
 Subp2: val2,
  ...     : ...
 Subpn: valn
}
```

Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions
  - Maintains a "dictionary"
  - Subproblems → solutions
- Recursive calls either:
  - Return a stored solution or
  - Compute and store a solution

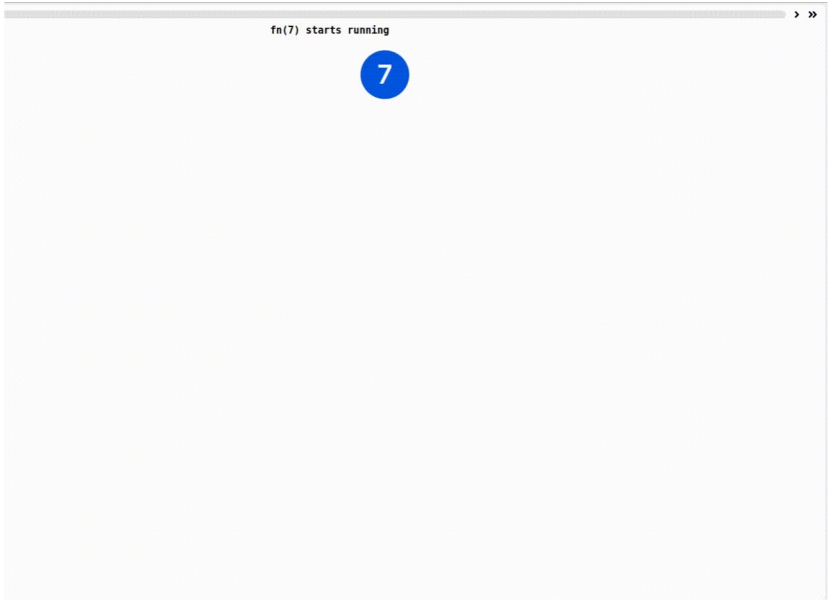
```
memo {
 Subp1: val1,
 Subp2: val2,
 ...    : ...
 Subpn: valn
}
```

Fibonacci sequence
Solution using Memoization

```
1.memo = {}
2.def fib(n):
3.    if n in memo: return memo[n]
4.    if n <= 2:
5.        f = 1
6.    else:
7.        f = fib(n-1) + f(n-2)
8.    memo[n] = f
9.    return f
```

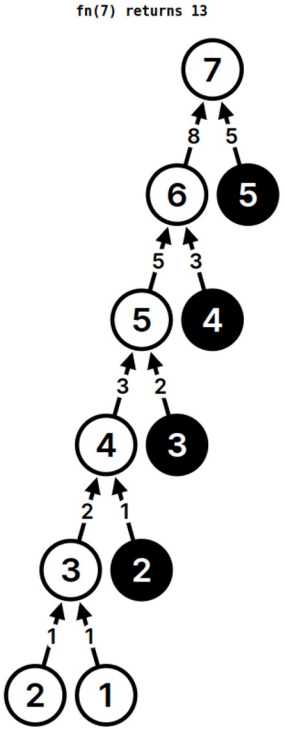
Fibonacci sequence
Solution using Memoization

```
1. memo = {}
2. def fib(n):
3.     if n in memo: return memo[n]
4.     if n <= 2:
5.         f = 1
6.     else:
7.         f = fib(n-1) + f(n-2)
8.     memo[n] = f
9.     return f
```



Fibonacci sequence
Solution using Memoization

```
1. memo = {}
2. def fib(n):
3.     if n in memo: return memo[n]
4.     if n <= 2:
5.         f = 1
6.     else:
7.         f = fib(n-1) + f(n-2)
8.     memo[n] = f
9.     return f
```



Fibonacci sequence
Solution using Memoization

```
1.memo = {}
2.def fib(n):
3.    if n in memo: return memo[n]
4.    if n <= 2:
5.        f = 1
6.    else:
7.        f = fib(n-1) + f(n-2)
8.    memo[n] = f
9.    return f
```

- Does fib(k) once for each k
- Runtime O(n)
  - Only n no 'memorized' calls
  - O(1) time per call
    - Ignore recursion

Memoization
A dynamic programming technique

- **The cost to compute each solution is paid only once**

Memoization
A dynamic programming technique

- **The cost to compute each solution is paid only once**
- **The cost of DP with memoization:**

$$Time \leq \sum_{subproblems} Nonrecursive\ work$$

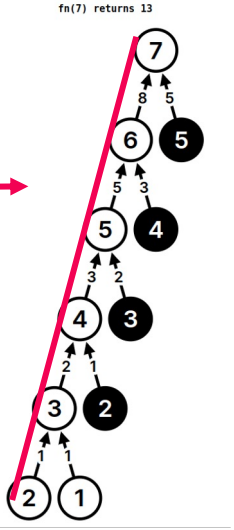
Memoization
A dynamic programming technique

- **The cost to compute each solution is paid only once**
- **The cost of DP with memoization:**

$$Time \leq \sum_{subproblems} \text{Non-recursive work}$$

$$\leq \#\text{subproblems} \times \text{non-recursive work}$$

O(1)



Memoization
A dynamic programming technique

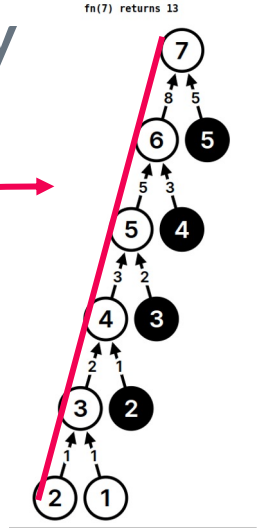
- **The cost to compute each solution is paid only once**
- **The cost of DP with memoization:**

$$Time \leq \sum_{subproblems} \text{Non-recursive work}$$

$$\leq \#\text{subproblems} \times \text{non-recursive work}$$

$$\leq O(n)$$

O(1)



Context
Dynamic programming

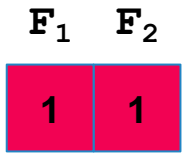
- Second perspective on PD:
  - DP ≈ Recursion + ”recycling"

Context
Dynamic programming

- Second perspective on PD:
  - DP ≈ Recursion + "reuse"
    - Memoization ("remind") & reuse solutions to subproblems that help solve the original problem

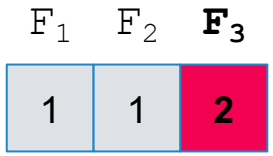
Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_botton_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1):
4.      memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```



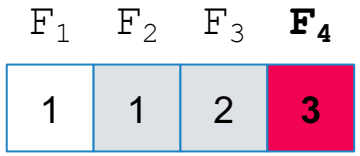
Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```



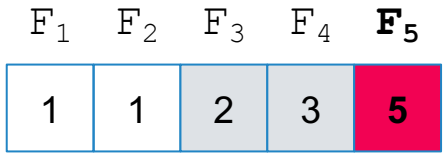
Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```



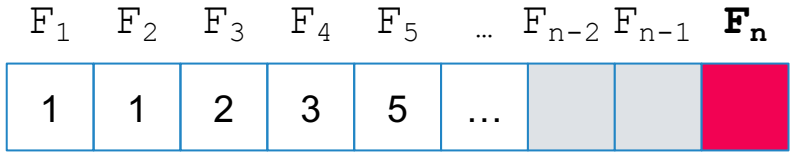
Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```



Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```



Bottom-Up
ANOTHER dynamic programming technique

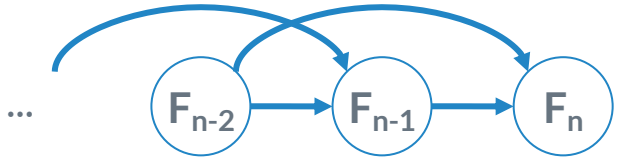
```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```

- Does the same computation as the memoized version

Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```

- Does the same computation as the memoized version
- Topological ordering of subproblem dependencies (form a DAG!)



Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```

} O(n)

- In practice it is faster
  - There is no recursion
- The analysis is more obvious

Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```

- In practice it is faster
  - There is no recursion
- The analysis is more obvious
- Can save space
  - We can remember only the last 2 fibs
    - Space O(1)

Bottom-Up
ANOTHER dynamic programming technique

```
1.def fib_button_up(n):
2.   memo[0] = memo[1] = 1
3.   for i in range(2,n+1)
4.     memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```

- In practice it is faster
  - There is no recursion
- The analysis is more obvious
- Can save space
  - We can remember only the last 2 fibs
    - Space O(1)

There is an implementation of the seq. Time cost Fibonacci O(lg n) via a different technique!

https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

Generic algorithms
Top-Down and Bottom-Up

```
1.memo = {}
2.def fib(n):
3.   if n in memo: return memo[n]
4.   if n <= 2:
5.       f = 1
6.   else:
7.       f = fib(n-1) + f(n-2)
8.   memo[n] = f
9.   return f
```

```
1.memo = {}
2.def f(subprob):
3.   if subprob not in memo:
4.     Memo[subprob] = base or recurrence
5.   return memo[n]
6.
```

Generic algorithms
Top-Down and Bottom-Up

```
1.def fib_button_up(n):
2.   memo[1] = momo[2] = 1
3.   for i in range(2,n+1)
4.      memo[i] = memo[i-1] + memo[i-2]
5.   return memo[n]
```

```
1.def f(subprob):
2.   Base case
3.   for subprob:
4.      memo[subprob] = REC relation.
5.   original return
6.
```

Comparison between PD techniques: memoization (top-down) and tabulation (bottom-up)

	Tabulation (bottom-up)	Memoization (Top-Down)
Speed	Fast. Directly accesses dependent solutions directly from the table	Slow. Due to multiple recursive calls and returns
Solution for subprob.	If all subproblems must be solved at least once, DP using Bottom-up usually performs better than top-down DP	If not all subproblems in the subproblem space need to be solved, the solution using memoization has the advantage of solving only the necessary subproblems
Memo filling	Starts from the first entry. The other entries are filled in one by one.	The table is populated on demand, that is, not all entries are necessarily populated.
Code	It can become complex when you have multiple conditions	Typically less complicated and drawn directly from recurrence.

Algorithmic paradigms so far

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Recursive/ Divide-and-conquer. Break up a problem into *independent* subproblems, solve each subproblem, and combine solutions to form solution to original problem.

- subproblems are defined by their smaller size
- the input of the subproblems is not the same, only the size

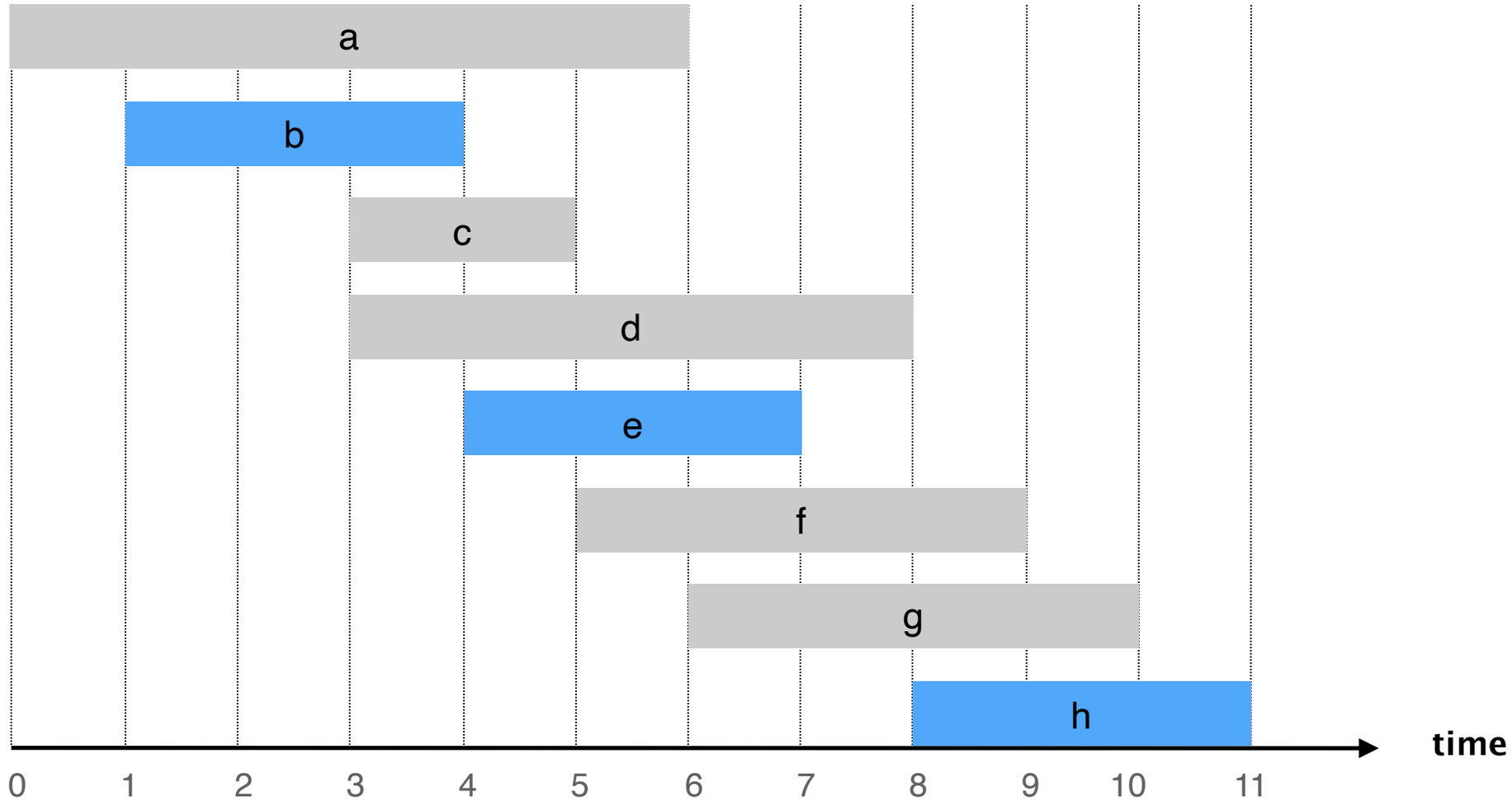
Dynamic programming. Break problem into a series of *reusable* subproblems, and build up solutions to larger and larger subproblems.

- subproblems are defined both by size and content
  - the outcome of each subproblem (as specified by their input) is reused multiple times

Weighted interval scheduling

~~Weighted~~ Interval Scheduling problem.

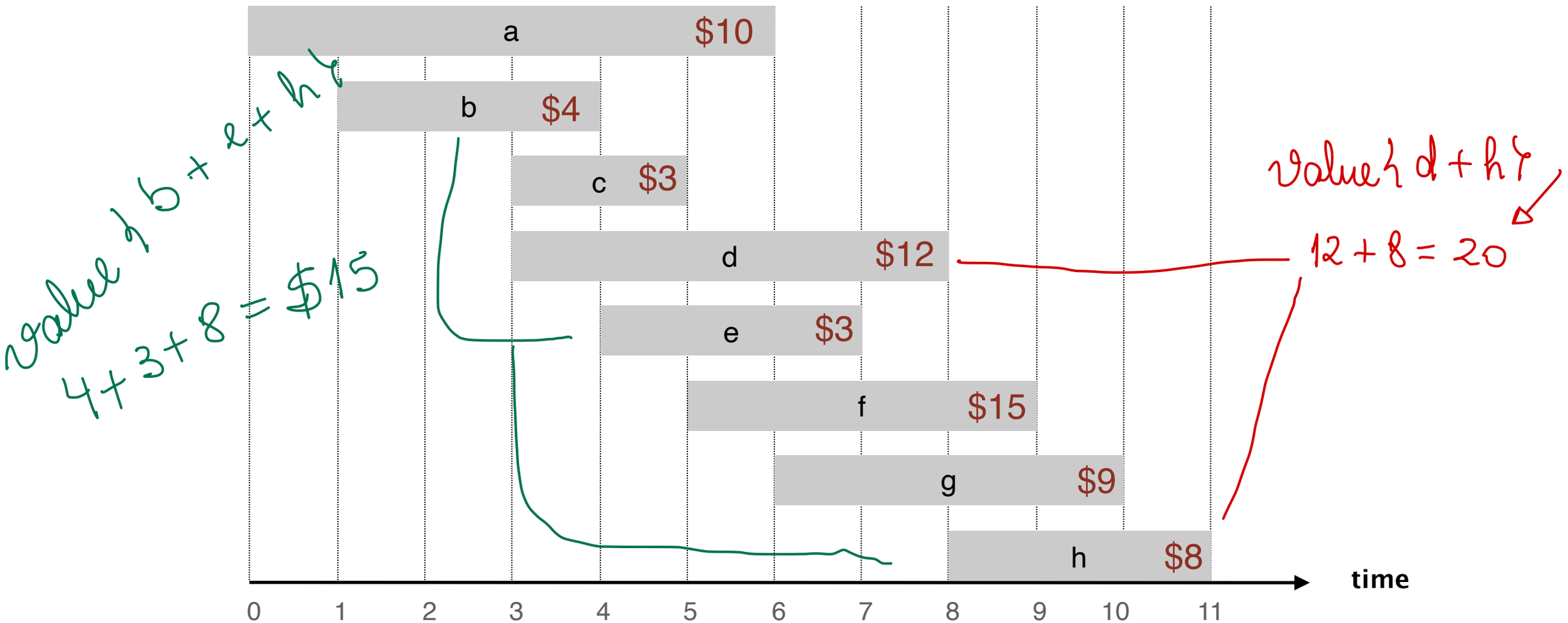
- Job j starts at s_j, finishes at f_j, and has ~~*weight* or *value* v_j.~~
- Two are jobs *compatible* if they don't overlap.
- Goal: find *maximum* subset of mutually compatible jobs.



Weighted interval scheduling

Weighted Interval Scheduling (WIS) problem.

- Job j starts at s_j, finishes at f_j, and has *weight* or *value* v_j.
- Two are jobs *compatible* if they don't overlap.
- Goal: find *maximum-weight/ max-value* subset of mutually compatible jobs.



Earliest-finish-time first algorithm

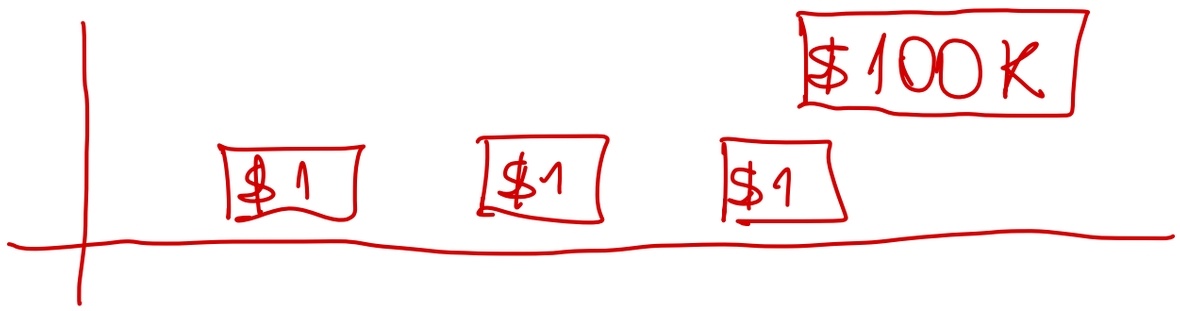
Earliest finish-time first.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Greedy algorithm is correct if all weights are 1.

Is it optimal when there are weights? No.

We can easily create one example with the most valuable job finishing last



TopHat Question

Brute force algorithm for WIS:

- For every subset I of intervals: $O(2^n)$
  - check if any intervals in I overlap $O(n^2)$
  - If not, then compute their total value $O(n)$
  - if better than previous best solution, store *max_value = {I, value(I)}* $O(n)$
- Return the tuple stored in max_value

Feasible solution = Set of compatible jobs

What is the running time of the algorithm in terms of the number of intervals n?

A. $\Theta(n^2)$

B. $\Theta(n^3)$

C. $\Theta(2^n)$

D. $\Theta(2^n n^2)$

E. none of the above

# of iterations => # of subsets in a set of size $n = 2^n$

systematic approach to WIS

Consider all combination of jobs (inefficient), but go through them in some sensible way (hopefully more efficient).
We will do this recursively.

1. order jobs by increasing finish time.
2. focus on the last job j_n and decide whether it should be included.

Weighted interval Scheduling

input: n jobs, for each of them start time s_j, finish time f_j, value v_j.

Objective: find a set of compatible jobs with maximum total value.

Question: Should we choose 8 as part of the solution?

Observation:

- if job 8 *is* selected, then the only jobs available are 1,…,5 and the maximum value we can earn is OPT(5) + $8
- If 8 is *not* selected, then we can choose from jobs 1,2,…,7

How we decide which one is better?

Try both!

Today we will learn a "careful brute force" method

Recursive subproblems

two cases:

- j_n *is part* of the optimal schedule O
  - recurse on the last job compatible to j_n
- j_n *is not part* of O
  - recurse on job j_{n-1}

We will explore these two options to find the full solution

The recursive step corresponds to solving a subproblem:

- a problem considering fewer jobs
- note that the subset of jobs is sequential — it contains *all* jobs before a certain index.

Compatibility

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

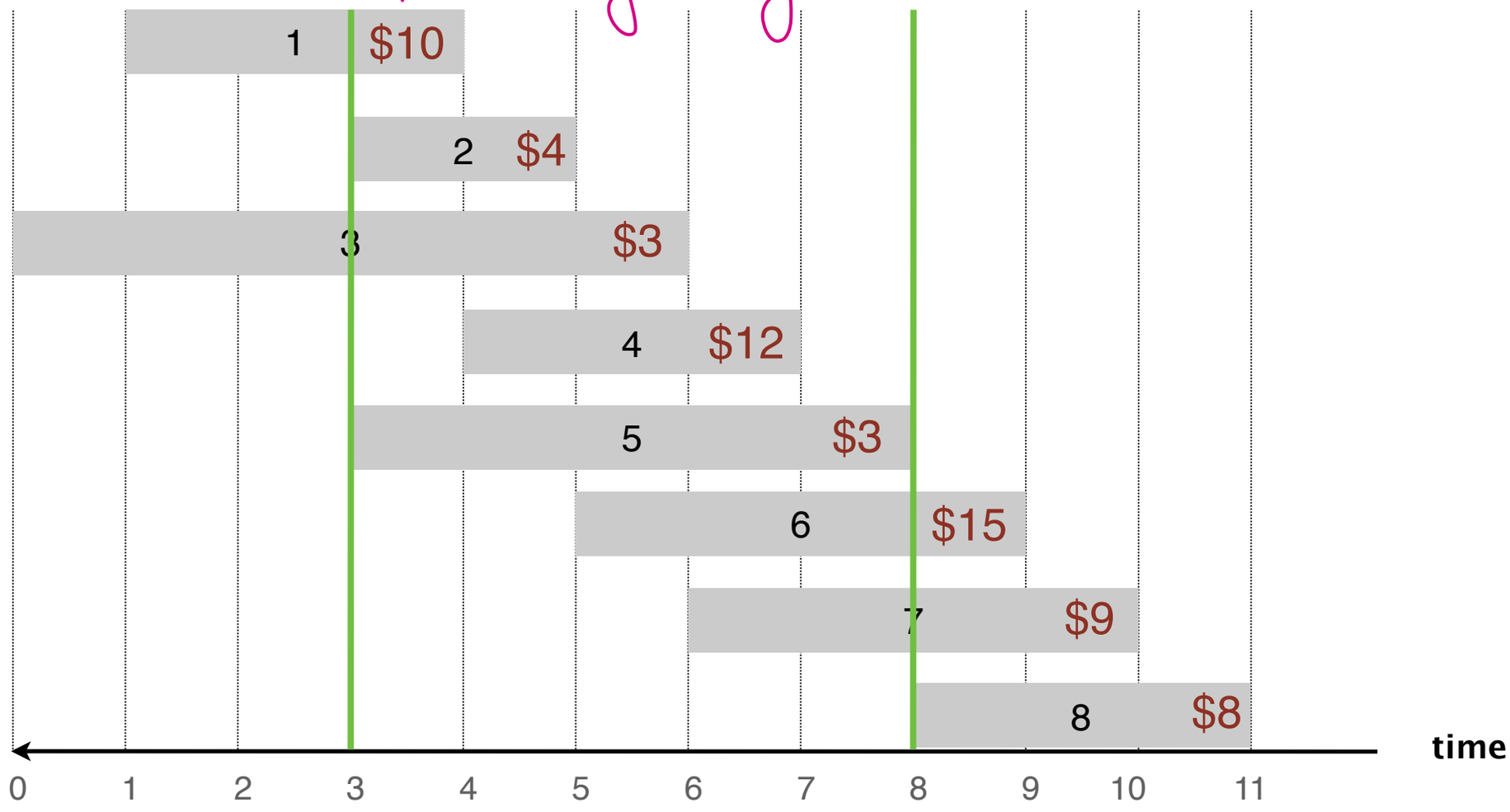
Definition. OPT(j) = maximum total value selection from jobs *1, 2, …, j*

Definition. p(j) = max {i: i <j and job i is compatible with j}

example: p(8) = 5, p(5) = 0

= highest indexed job before j that is compatible with j

not compatible with anything



TopHat Question

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

Definition. OPT(j) = maximum total value selection from jobs *1, 2, …, j*

Definition. p(j) = max {i: i <j and job i is compatible with j}

example: p(8) = 5, p(5) = 0

Question: What is p(7)?

A. 6

B. 5

C. 4

D. 3

E. 2

F. 1

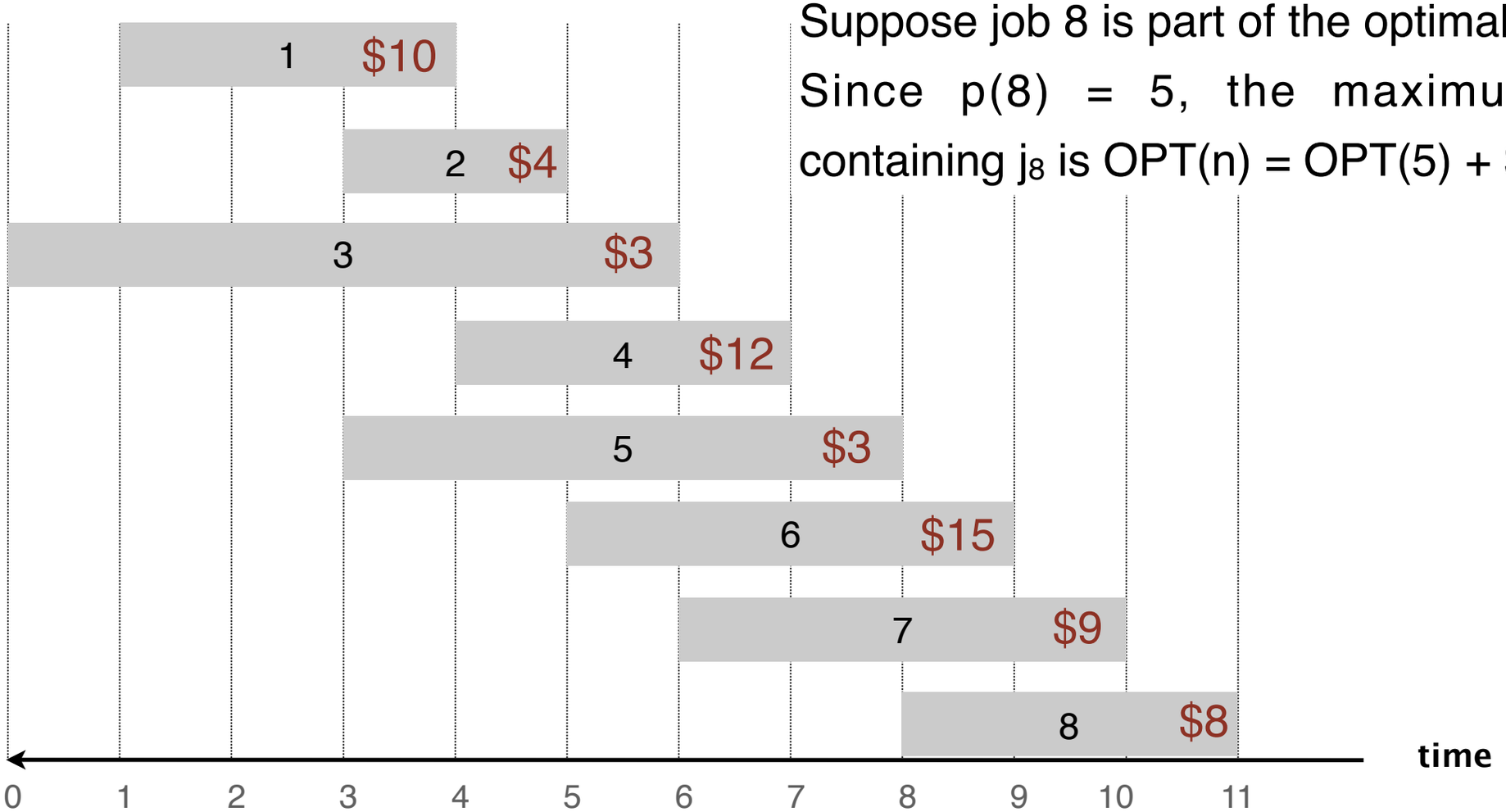
WIS – notation for compatibility

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

Def. $p(j)$ = largest index $i < j$ s.t. job i is compatible with j. (if none, then p(j) = 0)

Ex. $p(8) = 5, p(7) = 3, p(2) = 0.$

OPT(i) = maximum total value selection from jobs *1*, *2*, …, *i*



Observation:

Suppose job 8 is part of the optimal solution.

Since p(8) = 5, the maximum value containing j_8 is OPT(n) = OPT(5) + $8

DP for WIS: recursive formula

Notation. $OPT(j)$ = opt solution, i.e. max total value selection from jobs $1, 2, ..., j$.

$OPT(n)$ = value of optimal solution to the original problem.

j is part of the schedule

Case 1. $OPT(j)$ selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{ p(j)+1, p(j)+2, ..., j-1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$. This is *OPT(p(j))*. recurse on jobs $p(j), \ldots, 2, 1$

Total revenue = $v_j + OPT(p(j))$

Case 2. $OPT(j)$ does not select job j. = don't earn anything from j

- Must include optimal solution to problem consisting of remaining jobs $1, 2, ..., j-1$. This is *OPT(j-1)*. recurse on jobs $j-1, j-2, \ldots, 2, 1$

Total revenue = $OPT(j-1)$

Maximum revenue = $\max(v_j + OPT(p(j)), OPT(j-1))$

DP for WIS: recursive formula

Notation. $OPT(j)$ = opt solution, i.e. max total value selection from jobs $1, 2, ..., j$.

$OPT(n)$ = value of optimal solution to the original problem.

Case 1. $OPT(j)$ selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{ p(j)+1, p(j)+2, ..., j-1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$. This is *OPT(p(j)).*

Case 2. $OPT(j)$ does not select job j.

- Must include optimal solution to problem consisting of remaining jobs $1, 2, ..., j-1$. This is *OPT(j-1).*

Recursive formula : Choose the better from Case 1 and 2

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Correctness of the recursive formula

Claim. Computing the recursive formula yields the weight (i.e. maximum revenue) of the maximum schedule in WIS.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

proof. By induction on n.

base case: j = 0 , clearly no revenue

inductive assumption: Assume that the the formula OPT(j) yields the maximum value for every $j < k$

Prove for j:

By the inductive assumption we know that OPT(p(j)) and OPT(j-1) are optimal, since the index is < k. There are only two possible options, either j is part of the schedule or not. The max formula — that is using the correct values OPT(p(j)) and OPT(j-1) — compares these two cases.

WIS: exponential recursive algorithm

Algorithm 1: NaiveRecursiveWIS(n jobs: s_i, f_i, v_i)

```
1 sorted ← sort jobs by increasing finish time f_1 ≺ ... ≺ f_n;   O(n log n)
2 Compute p(1), p(2), ..., p(n)/* can be done in O(n)            */
3 return RecOpt(n)
```

Algorithm 1: RecOpt(job index j)

```
1 if j == 0 then                                   } Base case
2 |   return 0                                     }
3 else
4 |   Opt(j) ← max{v_j + RecOpt(p(j)); RecOpt(j − 1)};   } recursive formula (See previous slide)
5 |   return Opt(j)
```

Running time: $\Omega(2^{\frac{n}{2}})$

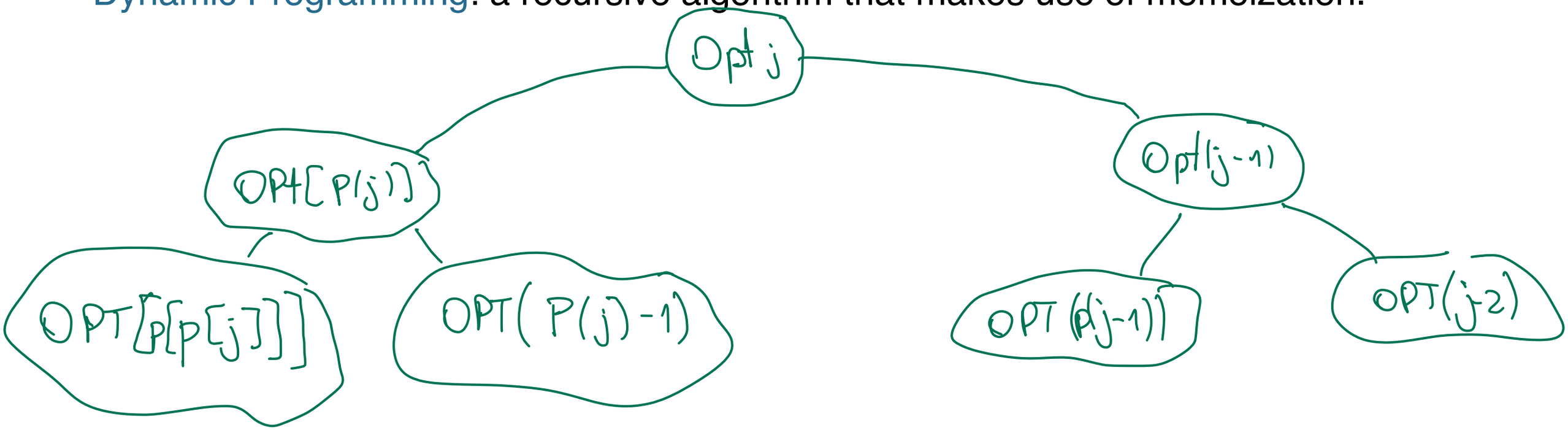
Exercise. Write the procedure to compute every p(j) in time O(n) (once jobs are sorted).

Memoization

Memoization: cache computed values

Instead of recomputing OPT(i) in every recursive call, we only compute it once and store the results. The next time we need it we simply look it up.

Dynamic Programming: a recursive algorithm that makes use of memoization.



Exponential amount of problems to solve

WIS: exponential recursive algorithm

Algorithm 1: NaiveRecursiveWIS(n jobs: s_i, f_i, v_i)

```
1 sorted ← sort jobs by increasing finish time f_1 ≺ ... ≺ f_n;
2 Compute p(1), p(2), ..., p(n)/* can be done in O(n)        */
3 return RecOpt(n)
```

memo = { }

Algorithm 1: RecOpt(job index j)

```
1 if j == 0 then
2 |   return 0
3 else
4 |   Opt(j) ← max{v_j + RecOpt(p(j)); RecOpt(j − 1)};
5 |   return Opt(j)
```

if j in memo: return memo[j]

memo[j] = OPT(j)

instead of recursive function call, use memoization.

Running time: $\Omega(2^{\frac{n}{2}})$

WIS – DP algorithm (recursive)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Memoization table M:

M[j] = OPT(j), array that contains the max value for jobs 0,1…,j

Sorting is the most expensive part

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

1 Sort jobs by finish time $f_1 \leq \ldots \leq f_n$; O(n log n)
2 Compute $p(1), p(2), \ldots p(n)$;
3 $M \leftarrow array(n+1)$// Empty array of size n+1, indexed 0...n
4 $M[0] \leftarrow 0$// no jobs selected
5 **return** $WISCompute(n)$;

Algorithm 2: WISCompute(j)

1 **if** $M[j]$ *is empty* **then**
2 $\quad M[j] \leftarrow \max\{v_j + WISCompute(p(j)) + WISCompute(j-1)\}$;
3 **return** $M[j]$;

O(n) calls ! one for each subproblem.

WIS – DP algorithm (bottom-up)

bottom-up algorithm to compute the optimal solution for WIS

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

```
1 Sort jobs by finish time f_1 ≤ ... ≤ f_n;
2 Compute p(1), p(2) ... p(n);
3 M ← array(n + 1)// empty array fo size n+1, indexed 0...n
4 M[0] ← 0;
5 for j = 1 to n do
6 |   M[j] ← max{v_j + M[p(j)]; M[j − 1]};
7 return M[n];
```

O(n log n)

O(n)

O(n)

Running time?

Recursive or bottom-up implementation have the same efficiency.

Note: M stores the values we can earn. We still don't know the set of jobs.

Use backtracking to find out!

WIS – DP algorithm – how to write a complete solution

1.: clearly define the subproblems, with proper indexing

OPT(j) = maximum value selection from job requests 1,..,j

2.: write the recursive formula:

p(j) = max {i: i <j and job i is compatible with j} = highest index of a job that doesn't overlap with j.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

3.: bottom-up algorithm to compute the optimal solution

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

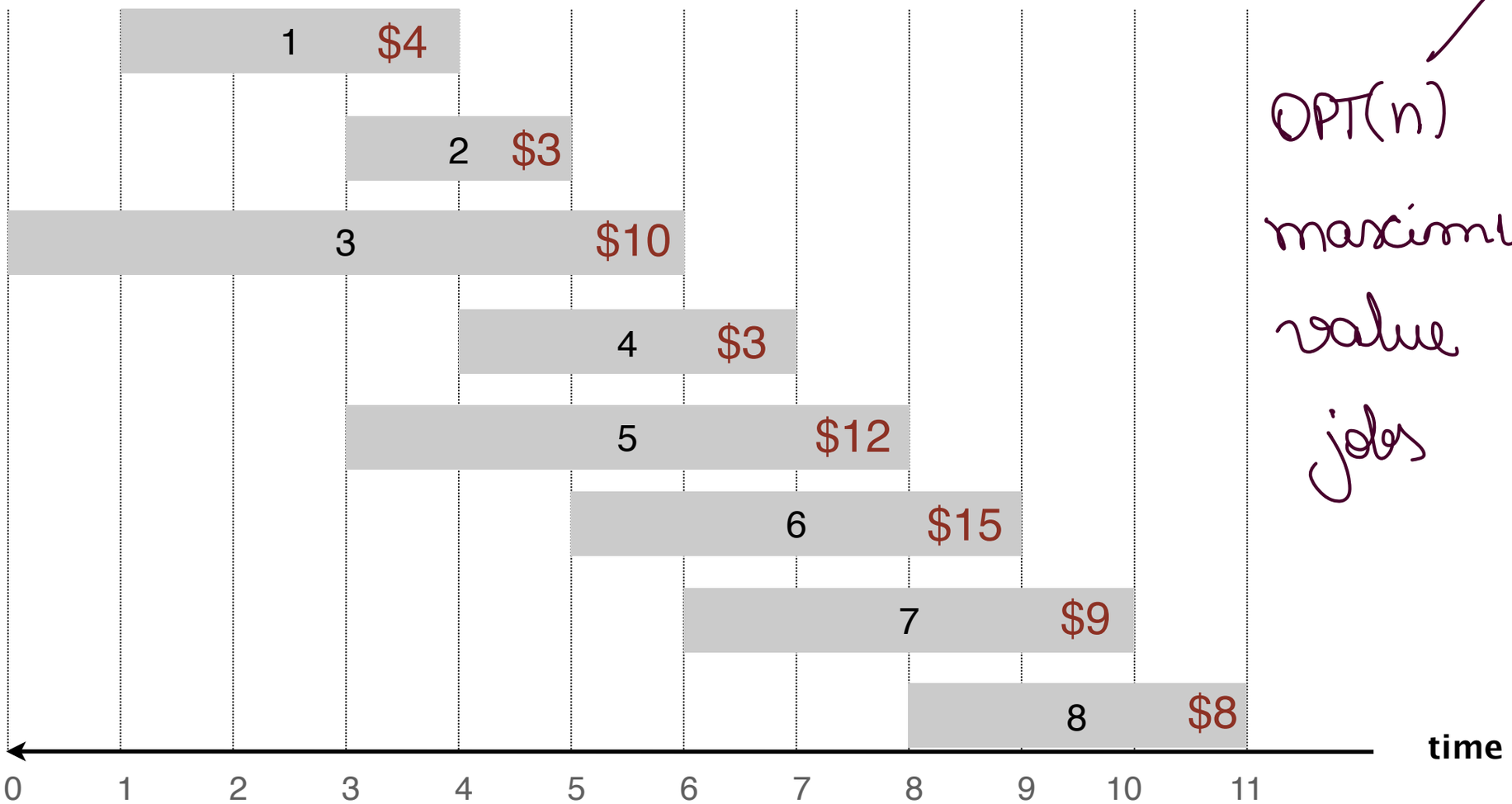
```
1 Sort jobs by finish time f_1 ≤ ... ≤ f_n;
2 Compute p(1), p(2) ... p(n);
3 M ← array(n + 1)// empty array fo size n+1, indexed 0...n
4 M[0] ← 0;
5 for j = 1 to n do
6 |   M[j] ← max{v_j + M[p(j)]; M[j − 1]};
7 return M[n];
```

4.: use backtracking to find set of jobs in optimal solution

WIS example – finding OPT(n)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

j	1	2	3	4	5	6	7	8
OPT(j)	0	$4	$4	$10	$12	$19	$19	$20



OPT(n) maximum value on all jobs

Finding the set of optimal jobs – backtracking

A dynamic programming algorithm computes the optimal value.

How to find the solution itself?

We can reconstruct it from the table.

- backtrack based on the memoization table without explicitly storing values (by checking which case was chosen)

Algorithm 1: FindSolution(j)

```
1 if j == 0 then
2 |   return ∅;
3 else if v_j + M[p(j)] > M[j − 1] then
4 |   return {j} ∪ FindSolution(p(j));
5 else
6 |   return FindSolution(j − 1);
```

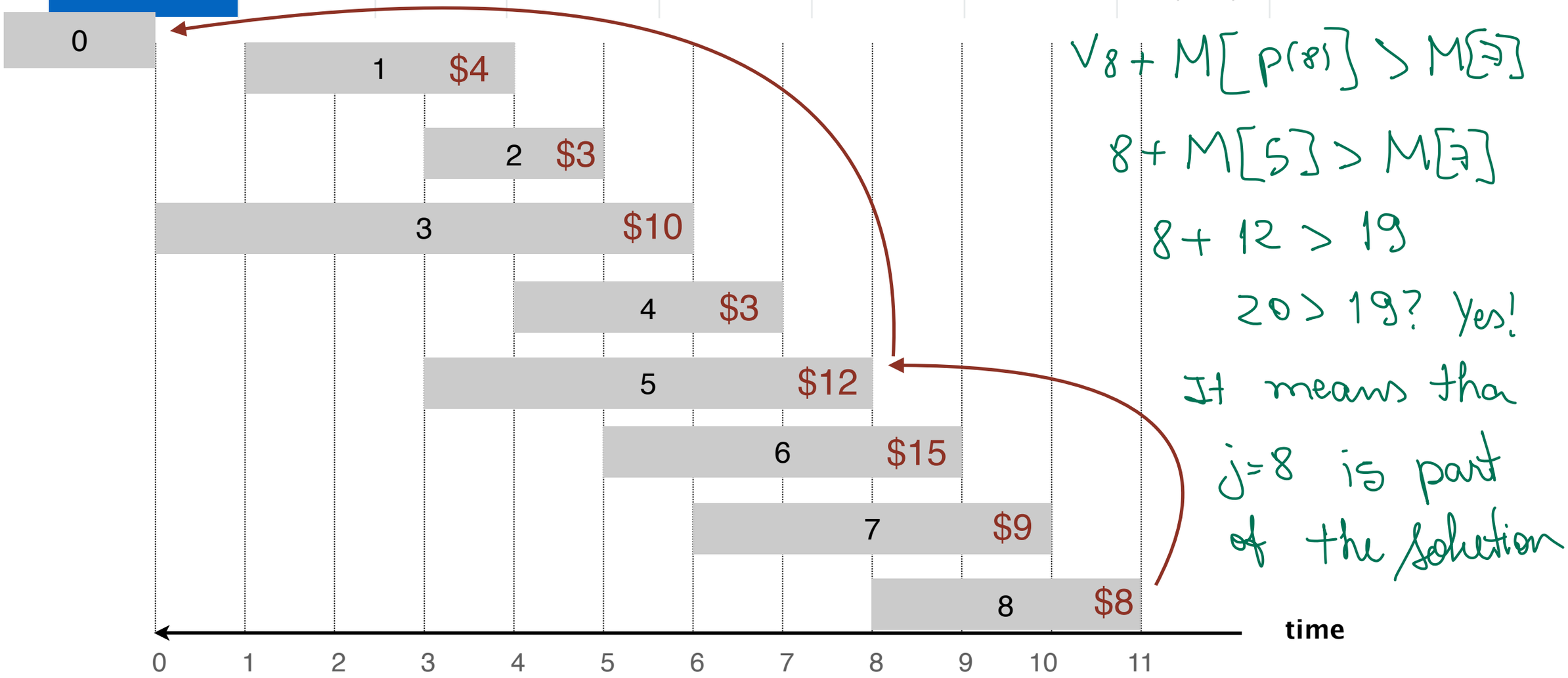
if this is true, it means that job j was select as part of the solution

Find the rest of the solution by looking at the previous compatible jobs.

Finding the set of optimal jobs – backtracking

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

j	1	2	3	4	5	6	7	8
OPT(j)	$4	$3	$10	$10	$12	$19	$19	$20
predecessor	0	0	0	3	0	1	3 (or 6)	5



$v_8 + M[p(8)] > M[7]$

$8 + M[5] > M[7]$

$8 + 12 > 19$

$20 > 19$? Yes!

It means that $j=8$ is part of the solution

WIS – DP algorithm (bottom-up/iterative)

Weighted Interval Scheduling: given n jobs, each with start time s_j, finish time f_j and value v_j find the compatible schedule with maximum total value.

OPT(j) = optimal solution for jobs (0),1,2,…,n

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Memoization table M:

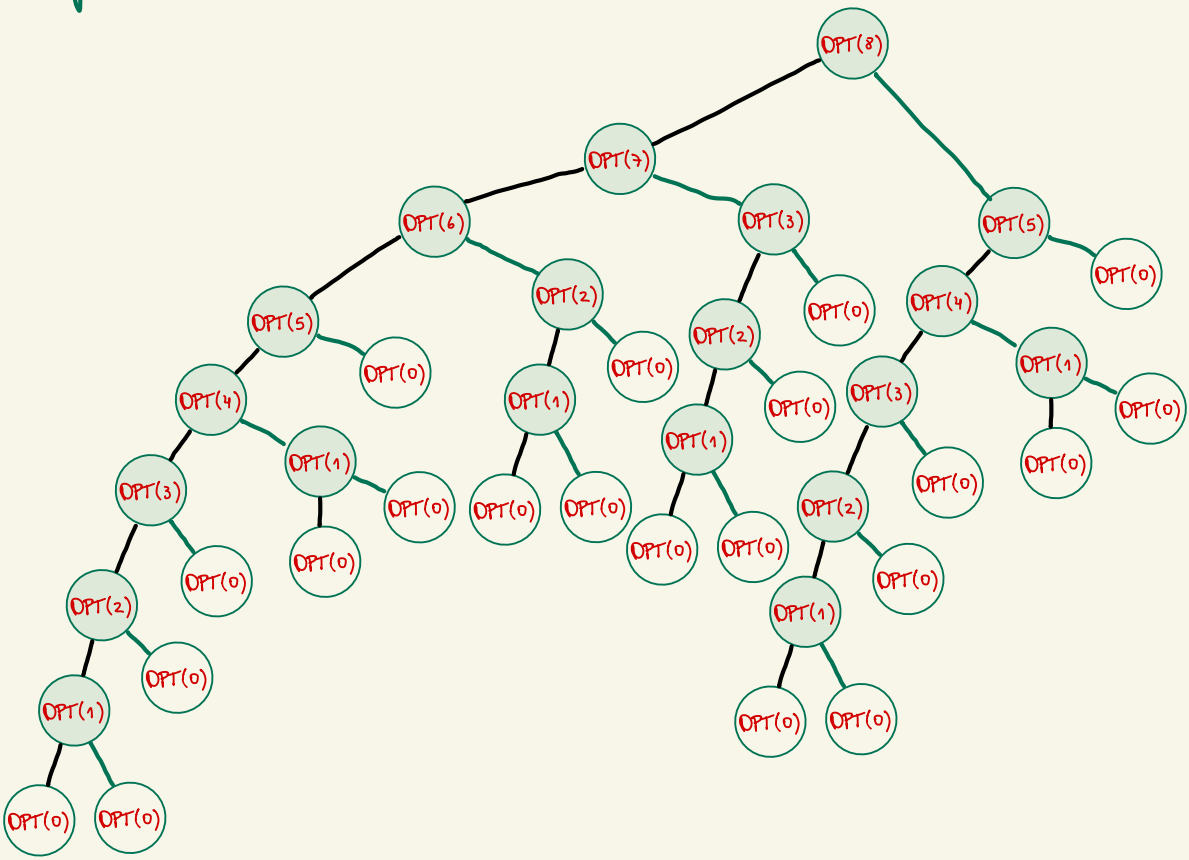
M[j] = OPT(j), array that contains the max value for jobs 0,1…,j

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

1 Sort jobs by finish time $f_1 \leq \ldots \leq f_n$;
2 Compute $p(1), p(2) \ldots p(n)$;
3 $M \leftarrow array(n+1)$// empty array fo size n+1, indexed 0...n
4 $M[0] \leftarrow 0$;
5 **for** $j = 1$ *to* n **do**
6 $\quad M[j] \leftarrow \max\{v_j + M[p(j)]; M[j-1]\}$;
7 **return** $M[n]$;

look straight into the memo table!

Complete tree with all subproblems from the example on the slides



Dynamic programming: adding a new variable

Knapsack Problem

Def. $OPT(i, w)$ = max-profit on items $1, \ldots, i$ with weight limit w.

Goal. $OPT(n, W)$.

↳ maximum capacity

Case 1. $OPT(i, w)$ does not select item i.

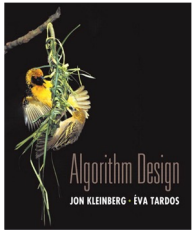
- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit w.

Case 2. $OPT(i, w)$ selects item i.

- Collect value v_i.
- New weight limit = $w - w_i$.
- $OPT(i, w\text{-}w_i)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ using this new weight limit.

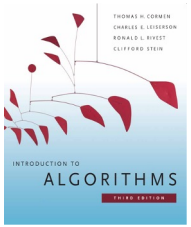
$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \} & \text{otherwise} \end{cases}$$

Access to information



§ 4 e 6

§ 4.3



§15 e 16

- Reference books:
  - Algorithms Theory and Practice [CLRS]
  - Algorithm Design [Jon Keiberg, Eva Tardos]
  - Algorithms [Robert Sedgewick]
    - https://algs4.cs.princeton.edu/home/