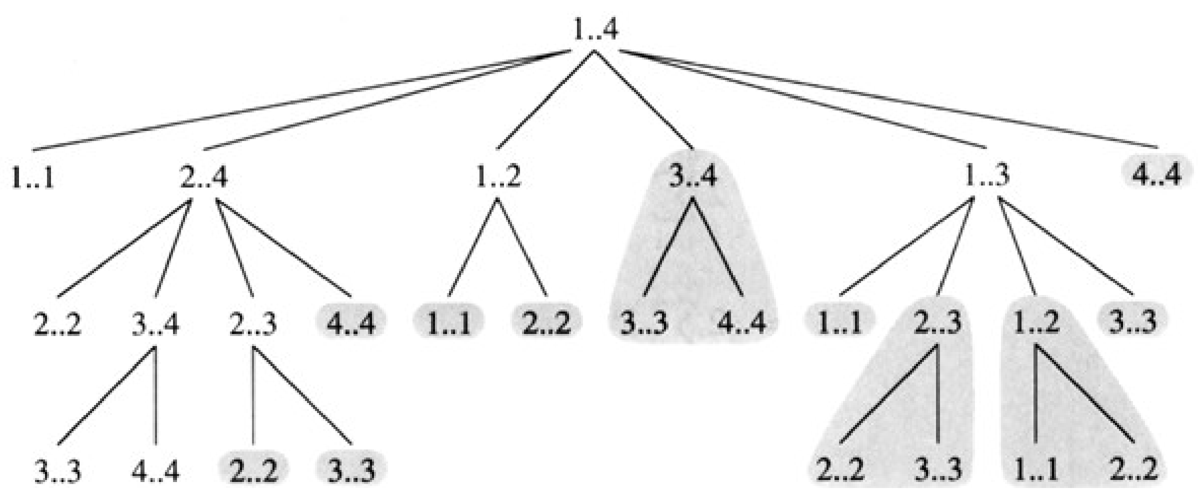
6. Dynamic programming

Algorithmic Paradigms

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

Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.



recursive matrix chain optimal multiplication order (Cormen et al., p.345)

Dynamic Programming History

Bellman (1920-1984). Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.

Dynamic programming = planning over time.

Secretary of Defense was hostile to mathematical research.

Bellman sought an impressive name to avoid confrontation.

- "it's impossible to use dynamic in a pejorative sense"
- "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Some famous dynamic programming algorithms.

- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

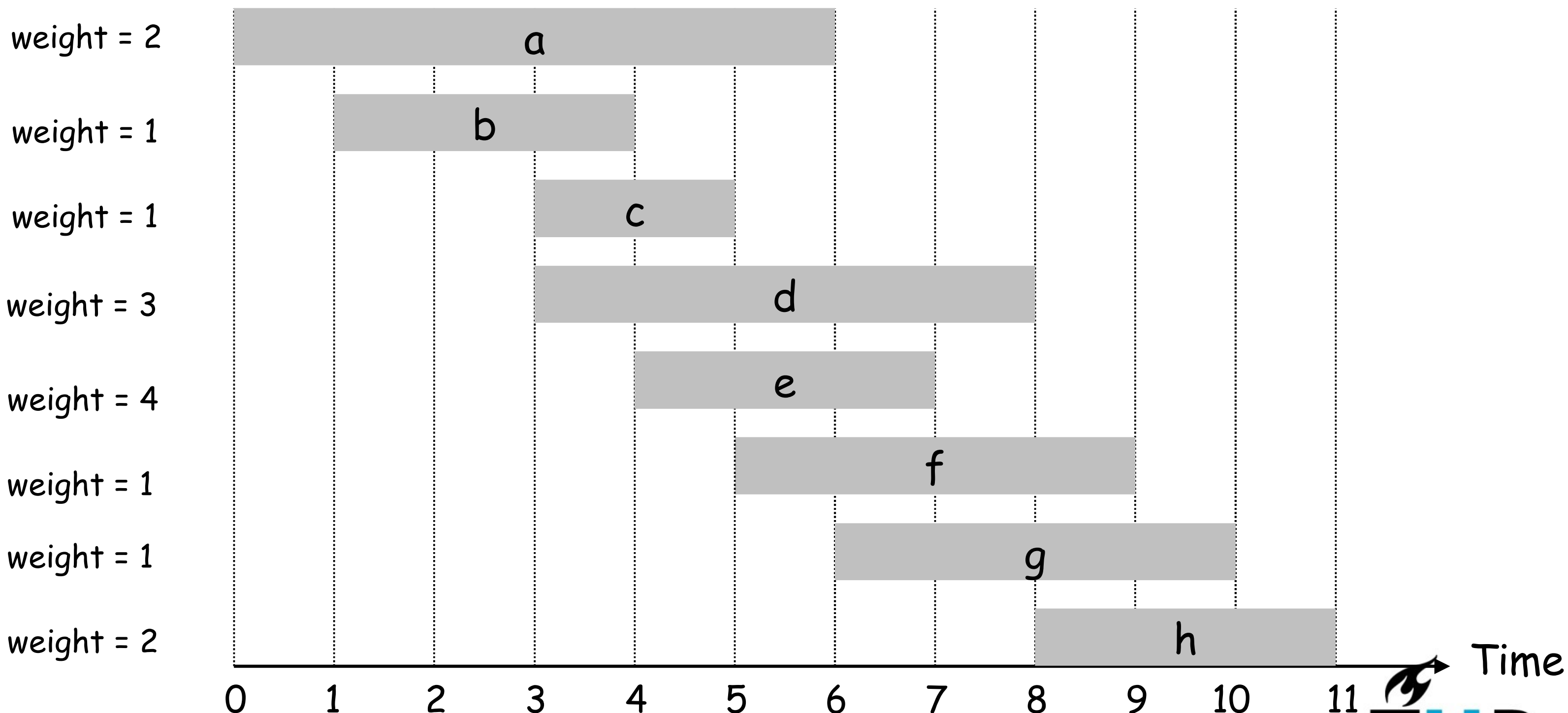
6.1 Weighted Interval Scheduling

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 jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

Q. Give an algorithm to solve this problem. (1 min)



TU Delft

Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

Consider jobs in ascending order of finish time.

Add job to subset if it is *compatible* with previously chosen jobs.

Q. What can happen if we apply the greedy algorithm for interval scheduling to weighted interval scheduling?

Unweighted Interval Scheduling Review

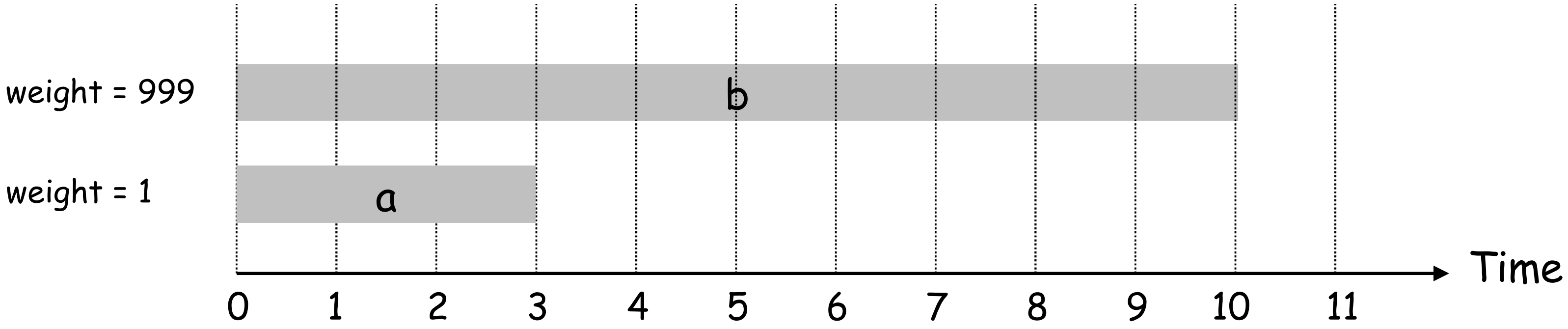
Recall. Greedy algorithm works if all weights are 1.

Consider jobs in ascending order of finish time.

Add job to subset if it is *compatible* with previously chosen jobs.

Q. What can happen if we apply the greedy algorithm for interval scheduling to weighted interval scheduling?

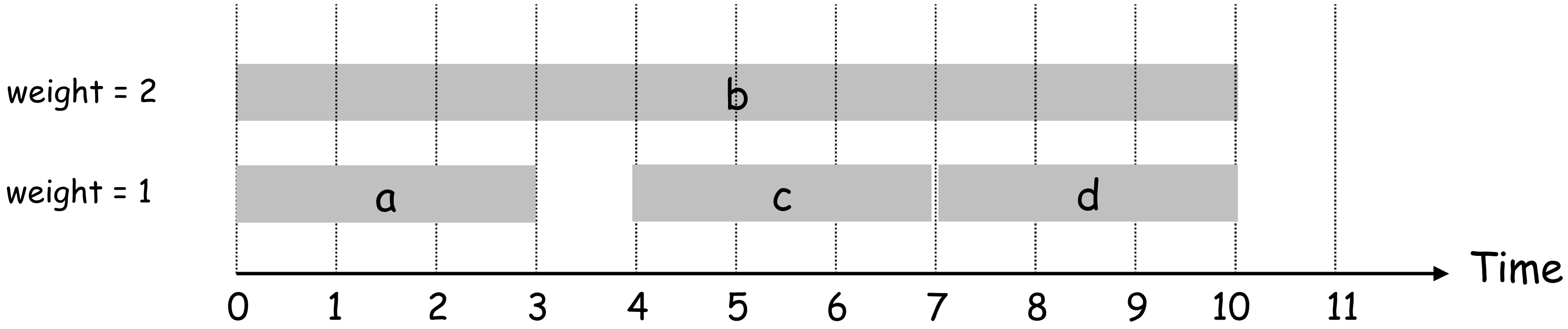
A. It can fail spectacularly.



Weighted Interval Scheduling: Greedy

Q. What can happen if we greedily sort on weight?

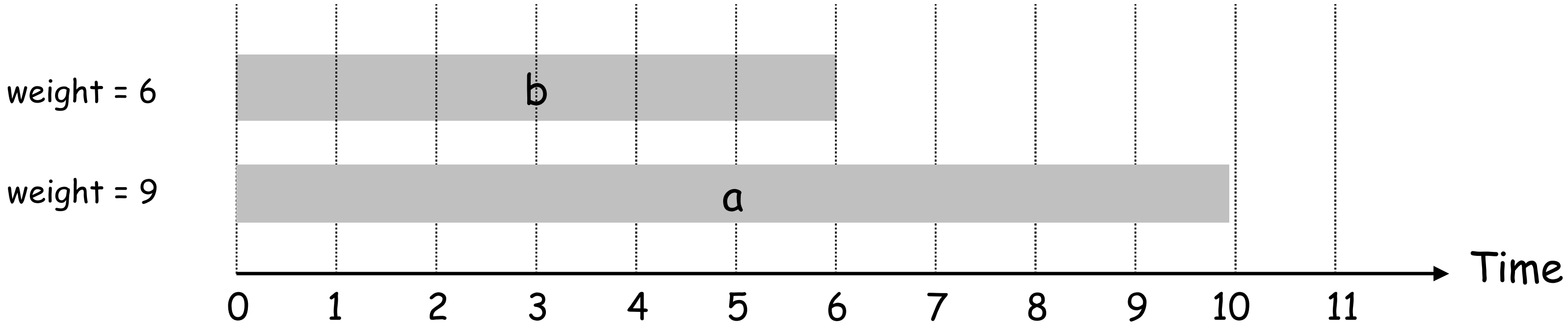
A. It can also fail.



Weighted Interval Scheduling: Greedy

Q. What can happen if we greedily sort on weight per time unit?

A. It can also fail (max. by a factor 2).



Weighted Interval Scheduling: Brute Force

Q. Maybe we need to consider all possibilities. How would you do that?
A. use back-tracking

Q. How many possible selections of jobs are there at most? (n^2, n^3, 2^n, $n!$)
A. Worst-case $O(2^n)$.

We'll now try to improve our brute-force algorithm a bit…

Weighted Interval Scheduling: Brute Force

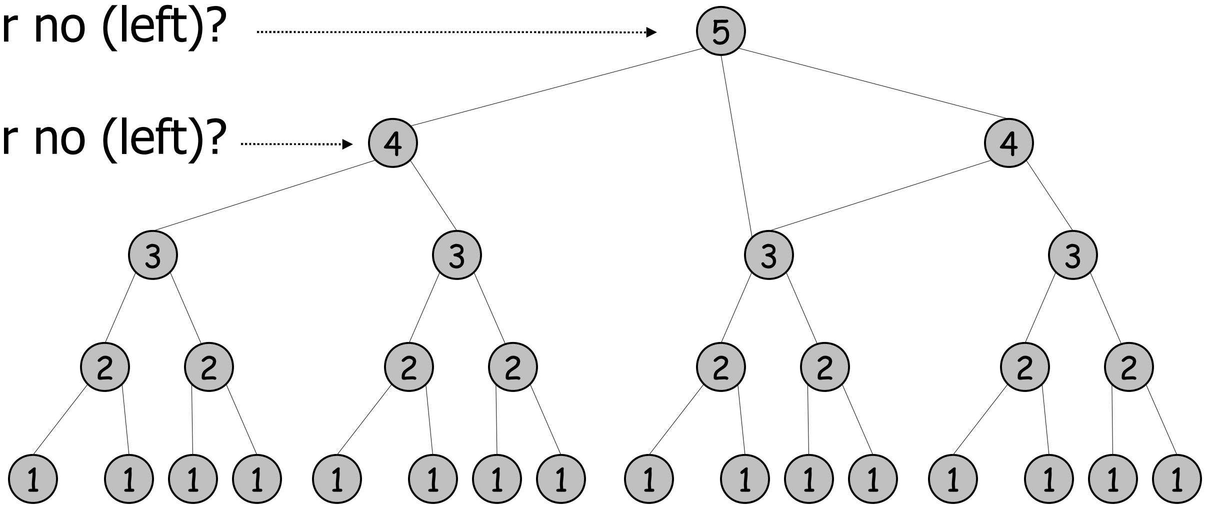
include 5? yes (right) or no (left)?

include 4? yes (right) or no (left)?

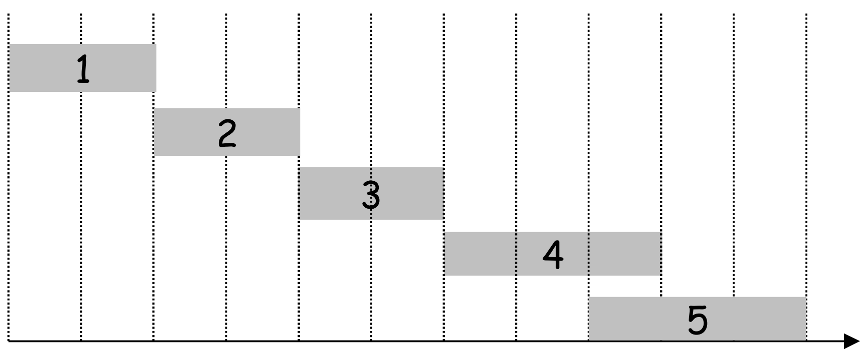
include 3?

include 2?

include 1?

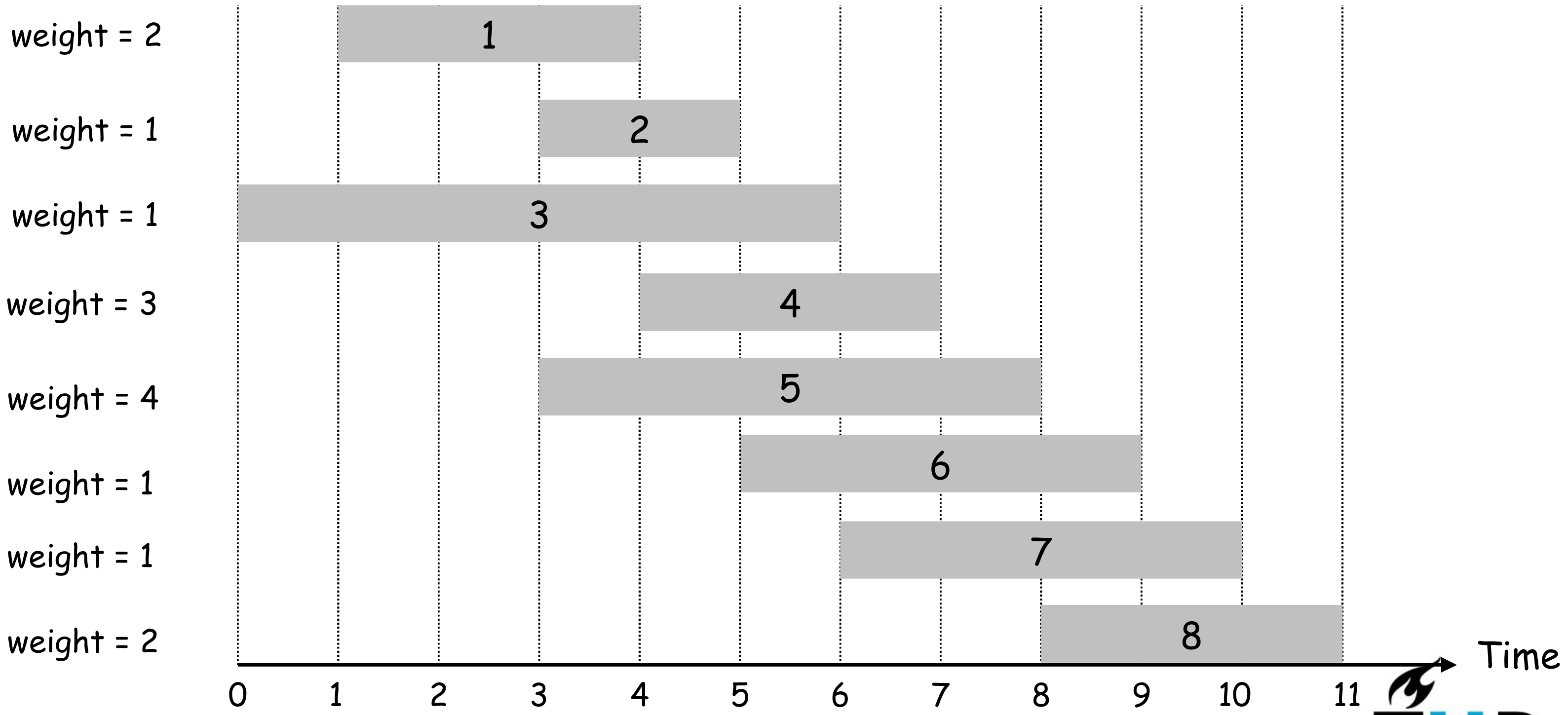


Note: recursion! (Is common with back-tracking).
Some combinations can be infeasible…



Weighted Interval Scheduling: Brute Force

Q. How to generalize this idea of skipping incompatible jobs (and implement this efficiently)?

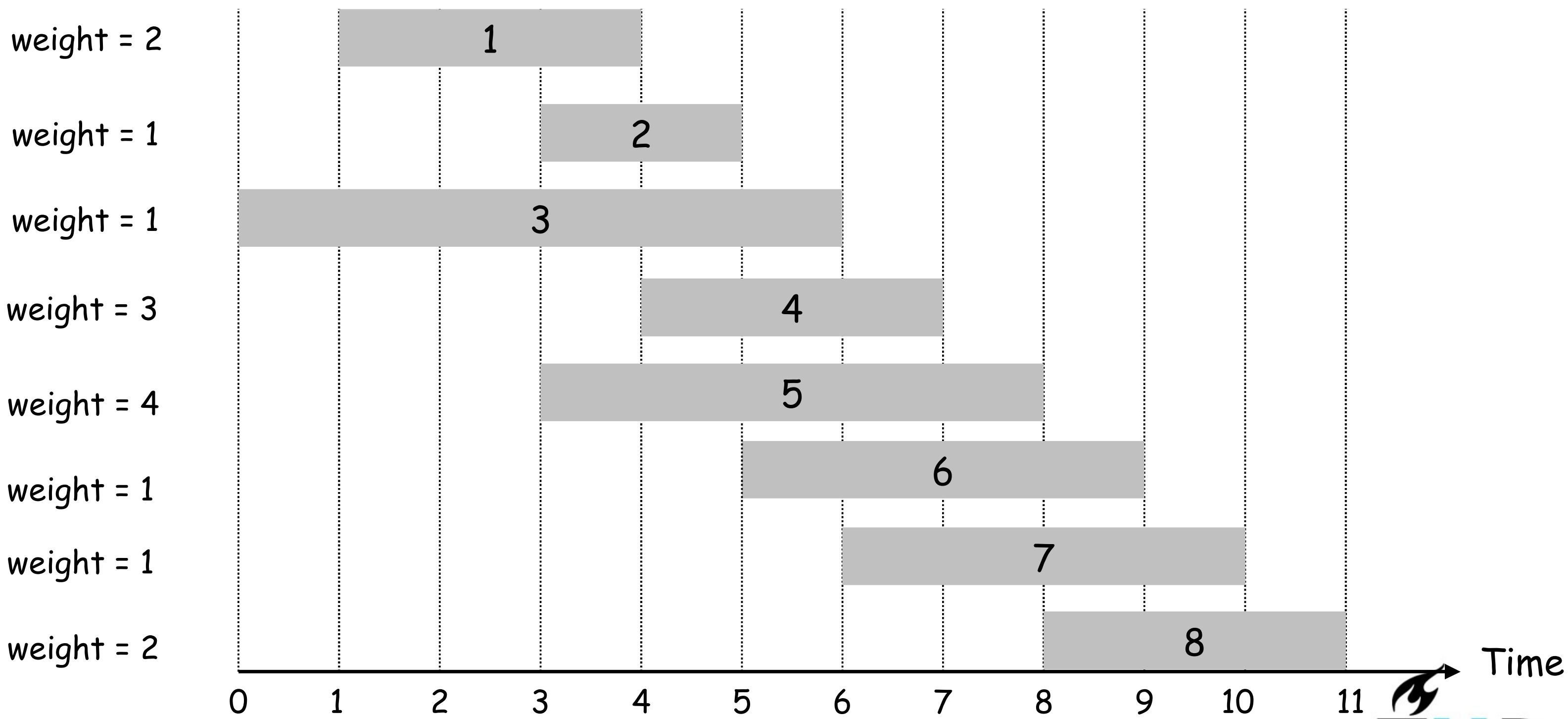


Weighted Interval Scheduling

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

Def. p(j) = largest index i < j such that job i is compatible with j. (predecessor)

Q. p(8) = ?, p(7) = ?, p(2) = ?.



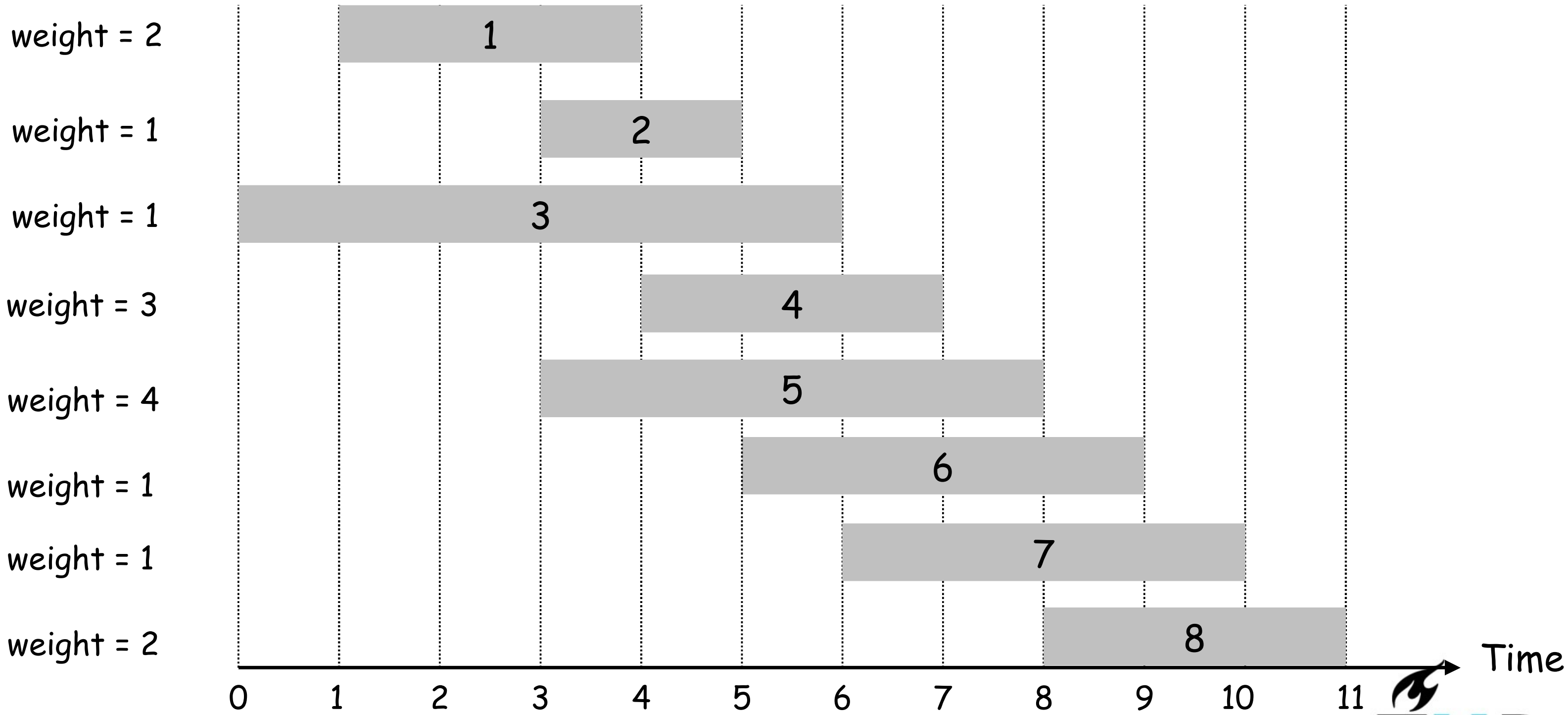
Weighted Interval Scheduling

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

Def. p(j) = largest index i < j such that job i is compatible with j. (predecessor)

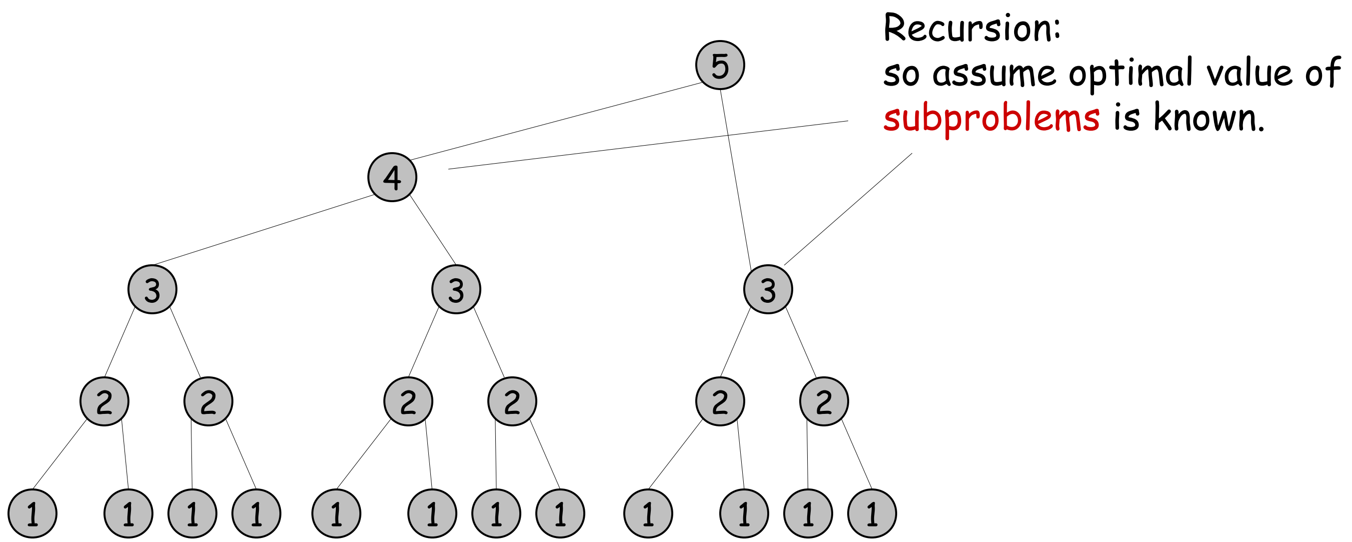
Q. p(8) = ?, p(7) = ?, p(2) = ?.

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



Weighted Interval Scheduling: Brute Force'

Q*. Precisely describe the computation in each recursive call?
(first: to find the maximum weight possible, later: find the schedule with the maximum weight)?



Weighted Interval Scheduling: Brute Force'

Notation. OPT(j) = *value* of optimal solution to the problem consisting of job requests 1, 2, ..., j (ordered by finishing time).

Case 1: OPT selects job 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)

optimal substructure

Case 2: OPT does not select job j.

- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

TU Delft

Weighted Interval Scheduling: Brute Force'

Notation. OPT(j) = *value* of optimal solution to the problem consisting of job requests 1, 2, ..., j (ordered by finishing time).

Case 1: OPT selects job 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)

optimal substructure

Case 2: OPT does not select job j.

- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ \underbrace{v_j + OPT(p(j))}_{\text{Case 1}},\ \underbrace{OPT(j-1)}_{\text{Case 2}} \} & \text{otherwise} \end{cases}$$

TU Delft

Weighted Interval Scheduling: Brute Force'

Brute force algorithm (with smart skipping of predecessors).

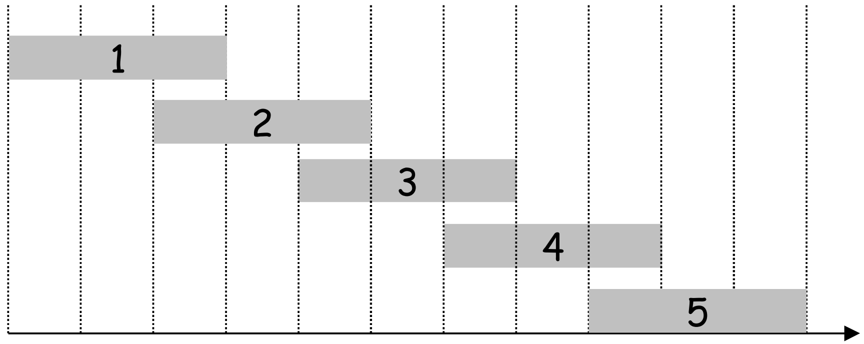
```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

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

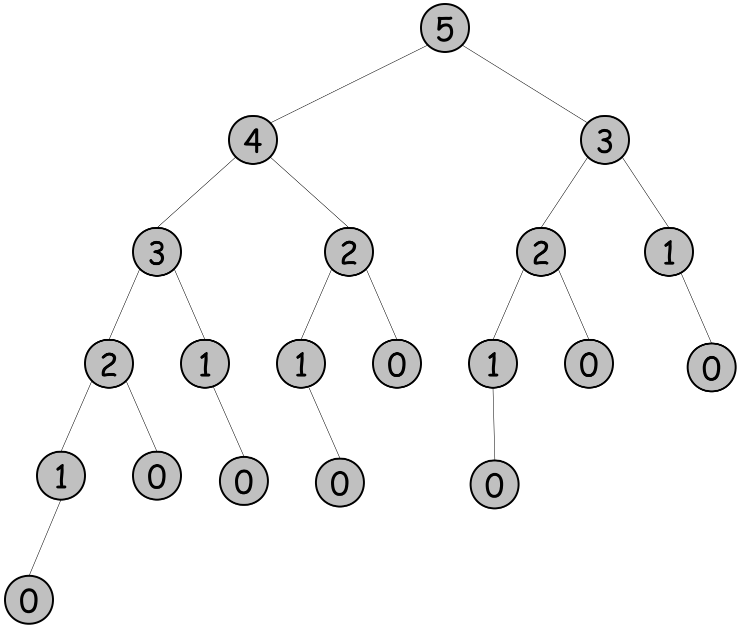
TU Delft

Weighted Interval Scheduling: Brute Force'

Q. Given n jobs, what is the run-time complexity on this problem instance?



p(1) = 0, p(j) = j-2

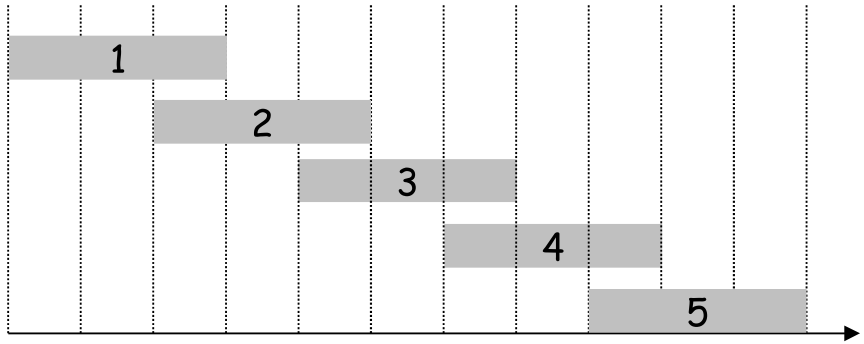



```
return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
```

Weighted Interval Scheduling: Brute Force'

Q. Given n jobs, what is the run-time complexity on this problem instance?
A. $T(0)=O(1)$ and $T(n) = T(n-1) + T(n-2) + O(1)$

Observation. Number of recursive calls grow like Fibonacci sequence $\Rightarrow$ exponential.
Observation. Recursive algorithm has many (redundant) sub-problems.
Q. How can we again improve our algorithm?



$p(1) = 0$, $p(j) = j-2$

```
return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
```

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty       ← global array
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty      ← global array
M[0] = 0

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]
}
```

Q. What is the run-time complexity of this algorithm with memoization? (1 min)

Weighted Interval Scheduling: Running Time

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

Proof.

Q. How many iterations in initialization?

Q. How many iterations in one invocation?

Q. How many invocations?

Weighted Interval Scheduling: Running Time

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

Proof.

Q. How many iterations in initialization?

- Sort by finish time: O(n log n).
- Computing $p(\cdot)$: O(n) by decreasing start time

Q. How many iterations in one invocation?

Q. How many invocations?

Weighted Interval Scheduling: Running Time

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

Proof.

Q. How many iterations in initialization?

Sort by finish time: O(n log n).

Computing p(·) : O(n) by decreasing start time

Q. How many iterations in one invocation?

`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

Q. How many invocations?

Weighted Interval Scheduling: Running Time

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

Proof.

Q. How many iterations in initialization?

Sort by finish time: O(n log n).

Computing $p(\cdot)$: O(n) by decreasing start time

Q. How many iterations in one invocation?

`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

Q. How many invocations?

Progress measure Φ = # nonempty entries of `M[]`.

- initially $\Phi = 0$, throughout $\Phi \leq n$.
- (ii) increases Φ by 1 and only then at most 2 recursive calls.

Overall running time (without init) of `M-Compute-Opt(n)` is O(n). ▪

Remark. O(n) if jobs are pre-sorted by start and finish times.

Automated Memoization

Q. What would the run-time be in a functional programming language?

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp

Automated Memoization

Automated memoization. Some functional programming languages (e.g., Lisp) have built-in support for memoization.

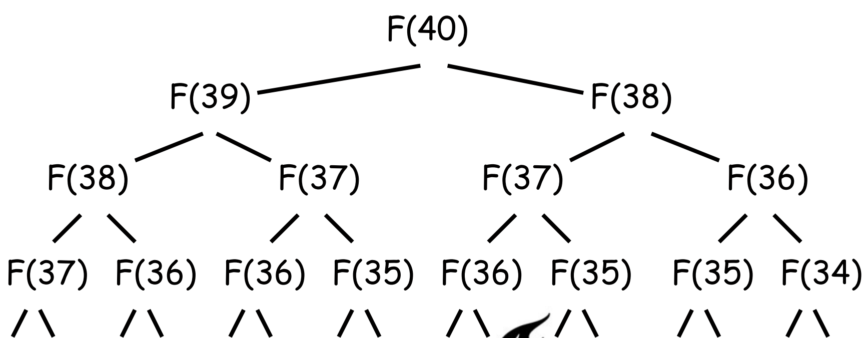
Q. Why not in imperative languages (e.g., Java)?

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficient)

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

Java (exponential)



Automated Memoization

Automated memoization. Some functional programming languages (e.g., Lisp) have built-in support for memoization.

Q. Why not in imperative languages (e.g., Java)?

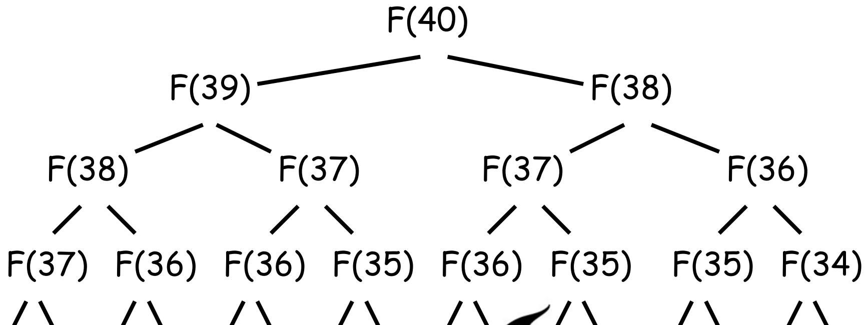
A. Because of side effects (in memory, on screen, etc.): not pure functions

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficient)

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

Java (exponential)



Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing (or store decisions in additional memo.-table).

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

# of recursive calls $\leq n \;\Rightarrow\; O(n)$.

TU Delft

Weighted Interval Scheduling: Bottom-Up

Q. Can this memoization be implemented without recursion?

Weighted Interval Scheduling: Bottom-Up

Q. Can this memoization be implemented without recursion?

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```

Weighted Interval Scheduling: Bottom-Up

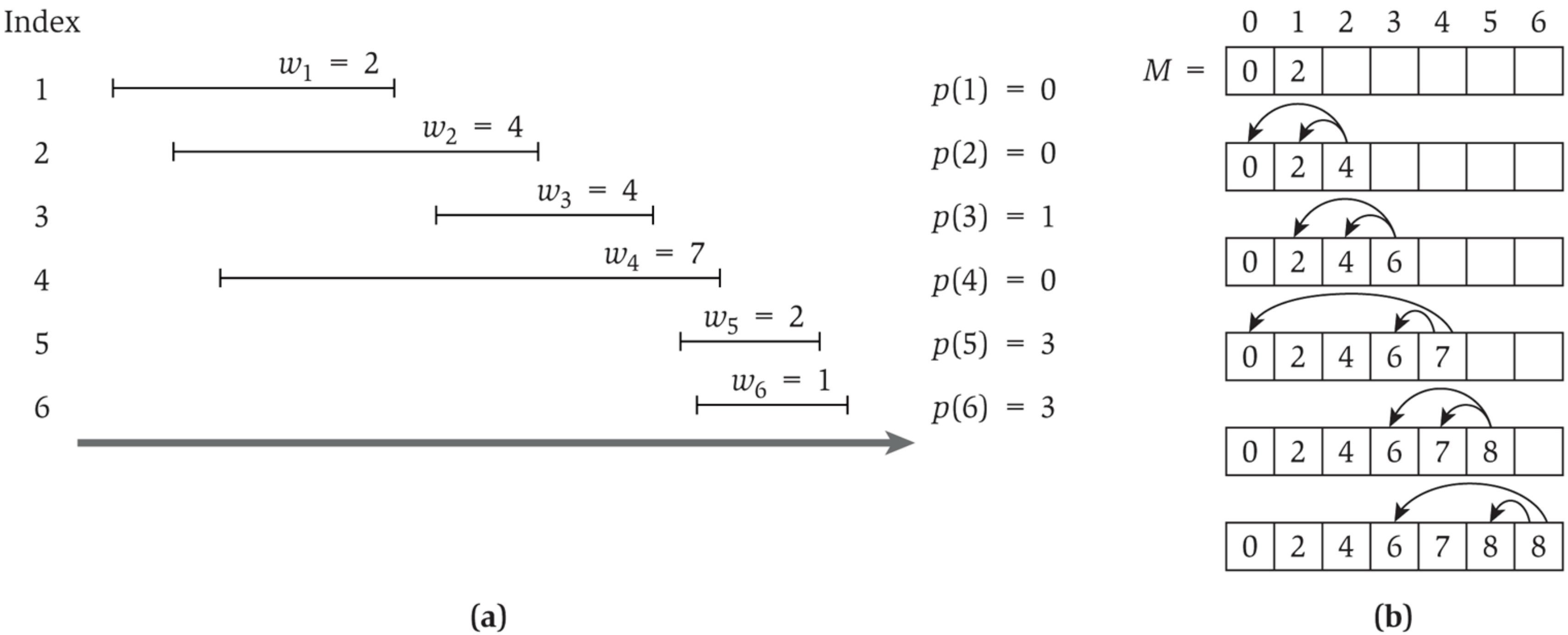


Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).