Algorithm Theory

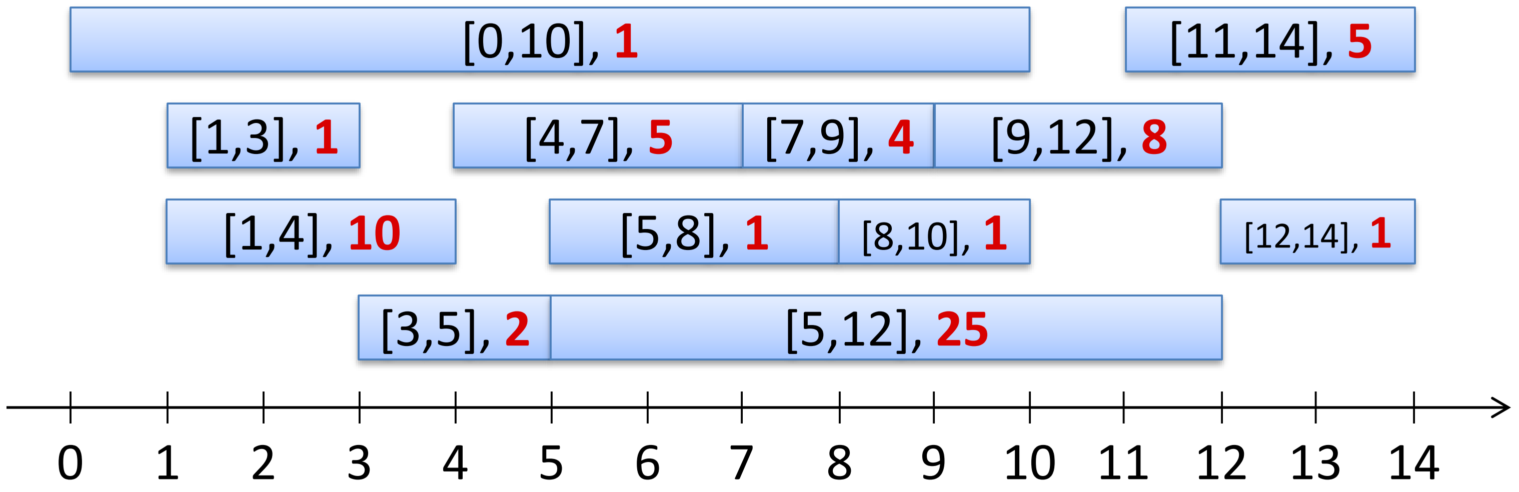
Chapter 3
Dynamic Programming

Part I:
Weighted Interval Scheduling

Fabian Kuhn

Weighted Interval Scheduling

- **Given:** Set of intervals, e.g.
  [0,10],[1,3],[1,4],[3,5],[4,7],[5,8],[5,12],[7,9],[9,12],[8,10],[11,14],[12,14]
- Each interval has a **weight w**

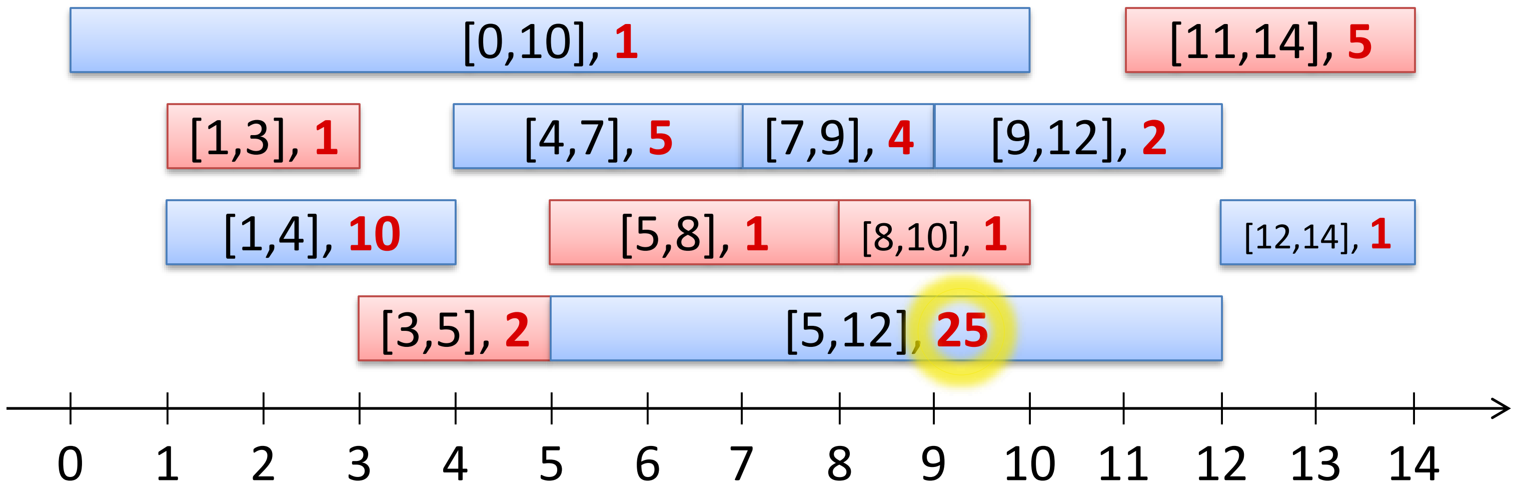


- **Goal:** Non-overlapping set of intervals of largest possible weight
  - Overlap at boundary ok, i.e., [4,7] and [7,9] are non-overlapping

- **Example:** Intervals are room requests of different importance

Greedy Algorithms

Choose available request with earliest finishing time:



- Algorithm is not optimal any more
  - It can even be arbitrarily bad…
- No greedy algorithm known that works

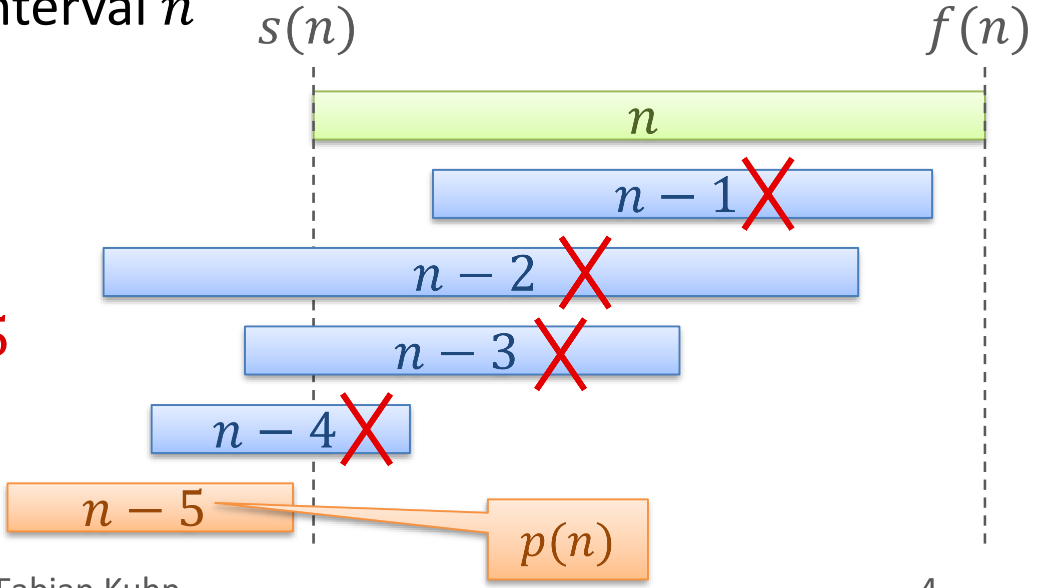
Solving Weighted Interval Scheduling

- **Interval $\boldsymbol{i}$ for $\boldsymbol{i = 1, \dots, n}$:**
  - start time $s(i) \geq 0$, finishing time $f(i) > s(i)$, weight $w(i) \geq 0$
- Assume intervals $1, \dots, n$ are **sorted by increasing $\boldsymbol{f(i)}$**
  - $0 < f(1) \leq f(2) \leq \cdots \leq f(n)$, for convenience: $f(0) = 0$

Simple observation: Opt. solution does or does not contain interval n

Case 1: opt. solution does **not contain** interval n
$\Longrightarrow$ opt. sol. for intervals $1, \dots, n$ = opt. sol. for intervals $1, \dots, n-1$

Case 2: opt. solution **contains** interval n



In example:
Opt. sol. consists of interval n

+

opt. sol. for intervals $1, \dots, n-5$

$p(n)$: first non-confl. Interval
here, $p(n) = n - 5$

Solving Weighted Interval Scheduling

- **Interval $\boldsymbol{i}$ for $\boldsymbol{i = 1, \dots, n}$:**
  - start time $s(i) \geq 0$, finishing time $f(i) > s(i)$, weight $w(i) \geq 0$
- Assume intervals $1, \dots, n$ are **sorted by increasing $\boldsymbol{f(i)}$**
  - $0 < f(1) \leq f(2) \leq \cdots \leq f(n)$, for convenience: $f(0) = 0$

Simple observation: Opt. solution does or does not contain interval n

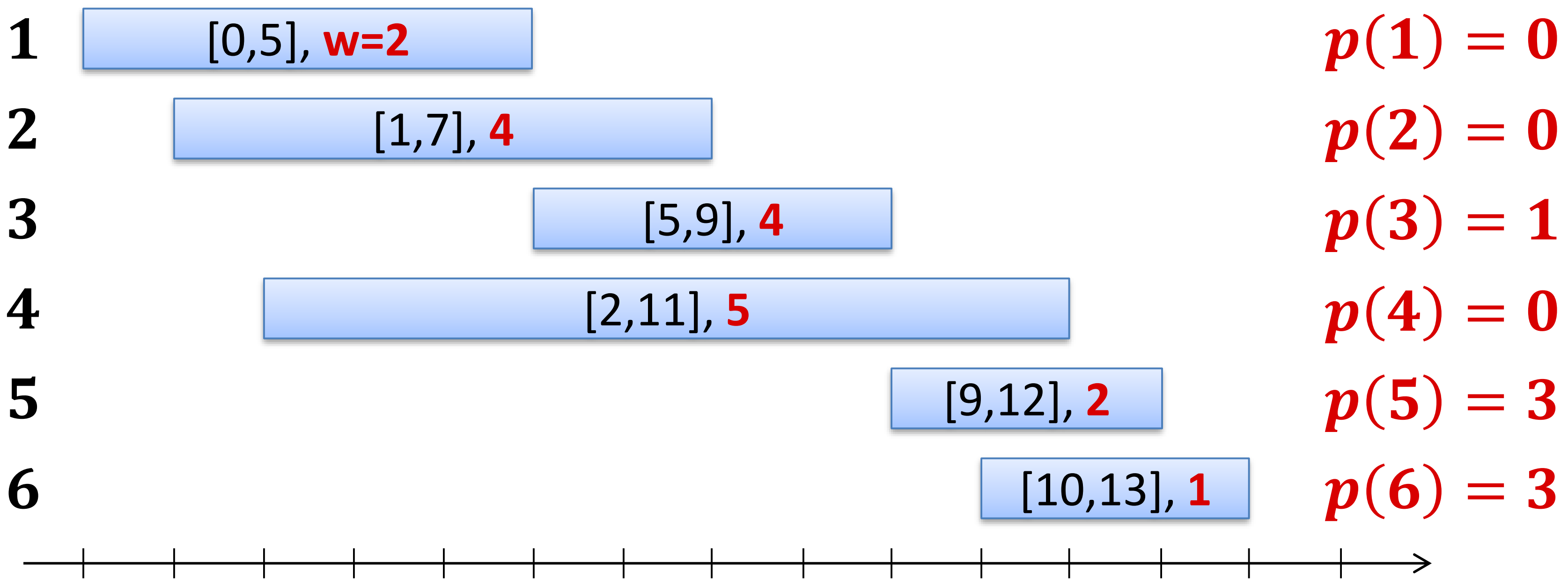
- Define $p(k) \coloneqq \max\{i \in \{0, \dots, k-1\} : f(i) \leq s(k)\}$
- Weight of optimal solution OPT_k for only intervals $1, \dots, k$: $W(k)$

- Solution OPT_k does not contain interval k: $\boldsymbol{W(k) = W(k-1)}$
  - Weight of optimal solution with only first $k-1$ intervals

- Solution OPT_k contains interval k: $\boldsymbol{W(k) = w(k) + W(p(k))}$
  - Weight of interval k plus weight of optimal solution of non-conflicting earlier intervals

Example

Interval:



Time to compute values $p(k)$:

- Assume that intervals are already sorted by finishing time.
- For each k, do a binary search $\Longrightarrow$ time $O(\log k)$
- Overall time: $O(n \log n)$

Recursive Definition of Optimal Solution

- Recall:
  - $W(k)$: weight of optimal solution with intervals $1, \dots, k$
  - $p(k)$: last interval that finishes before interval k starts
    - $p(k) = 0$ if there is no interval that finishes before interval k starts

- Recursive definition of optimal weight:

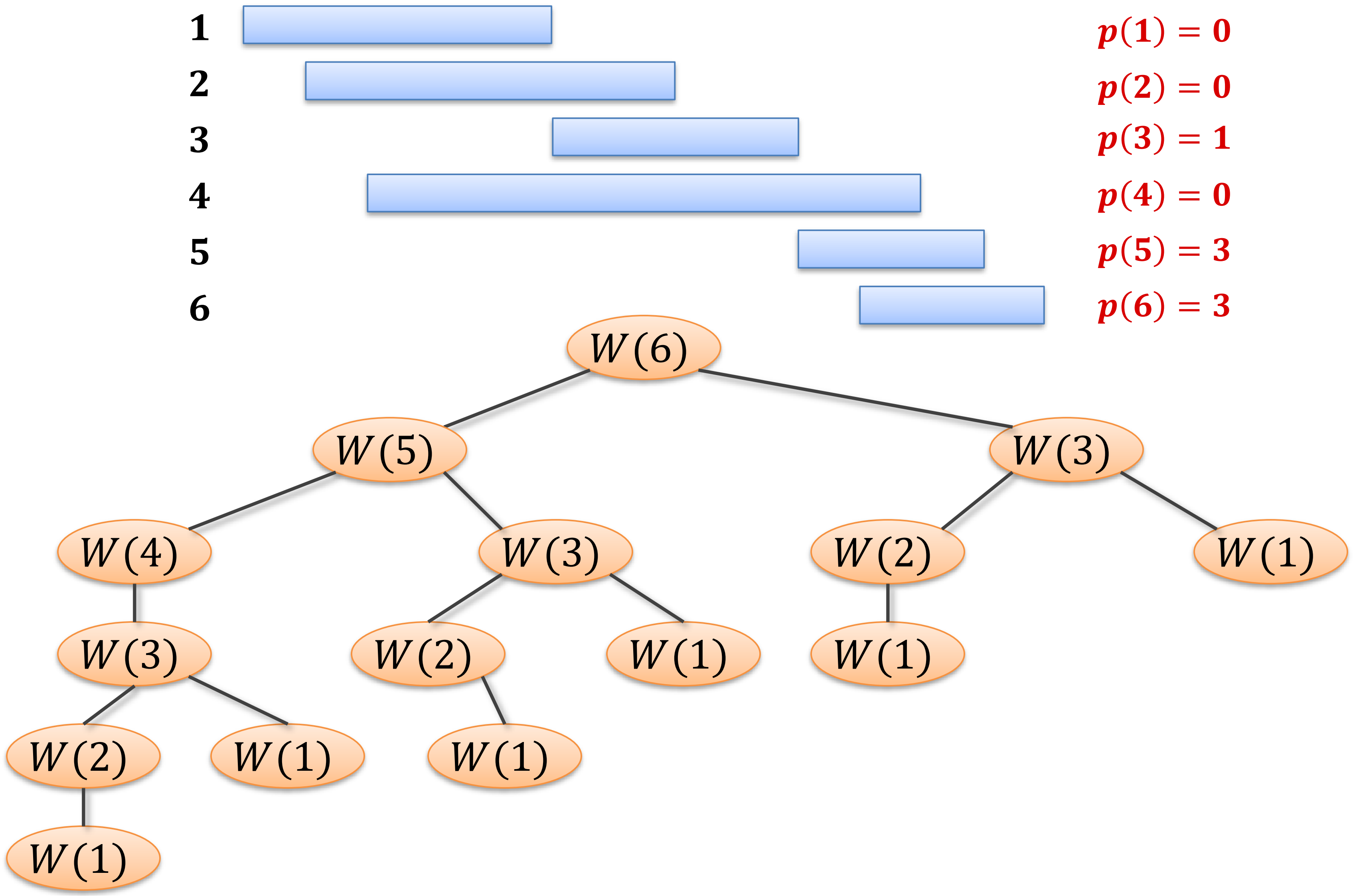
$$\forall k > 1:\ W(k) = \max\{W(k-1), w(k) + W(p(k))\}$$

$$W(0) = 0$$

Immediately gives a simple, recursive algorithm

```
Compute p(k) values for all k
W(k):
    if k == 0:
        x = 0
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    return x
```

Running Time of Recursive Algorithm



Memoizing the Recursion

- Running time of recursive algorithm: exponential!
- But, alg. only solves n different sub-problems: $W(1), \dots, W(n)$
- There is no need to compute them multiple times

Memoization: Store already computed values for future rec. calls

```
Compute p(k) for all k
memo = {};
W(k):
    if k in memo: return memo[k]
    if k == 0:
        x = 0
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    memo[k] = x
    return x
```

Dynamic Programming (DP)

DP $\approx$ Recursion + Memoization

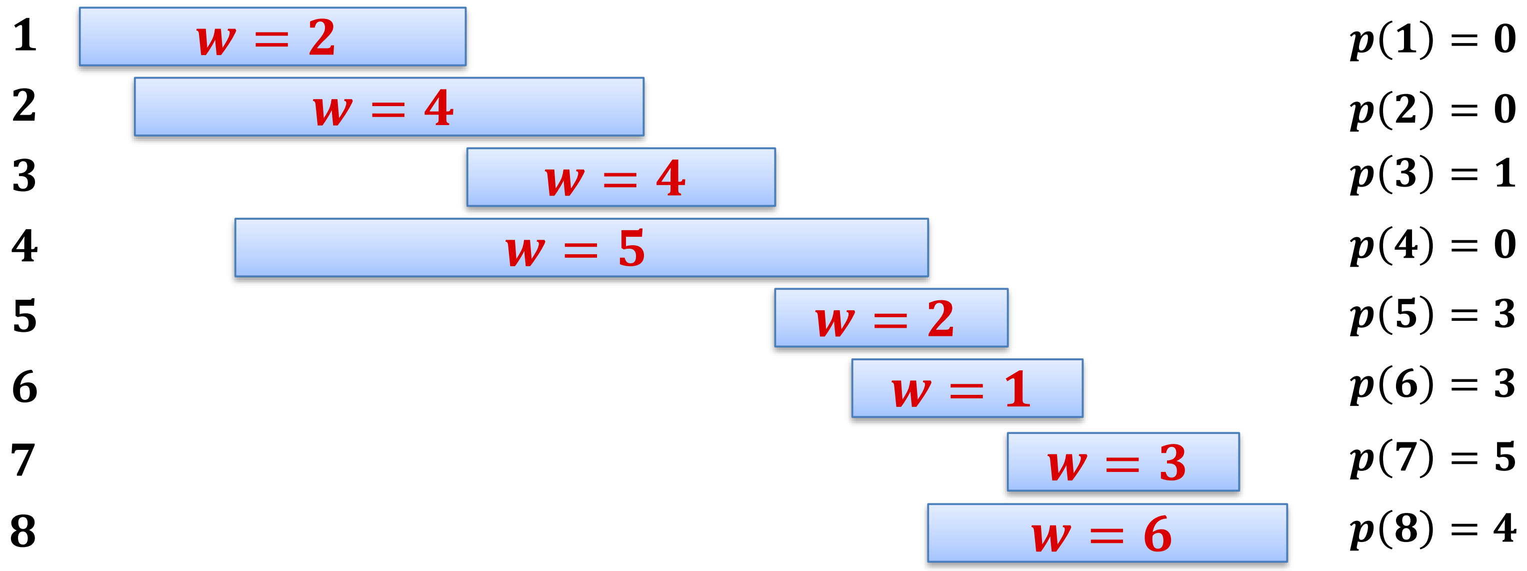
Recursion: Express problem *recursively* in terms of
(a 'small' number of) *subproblems* (of the same kind)

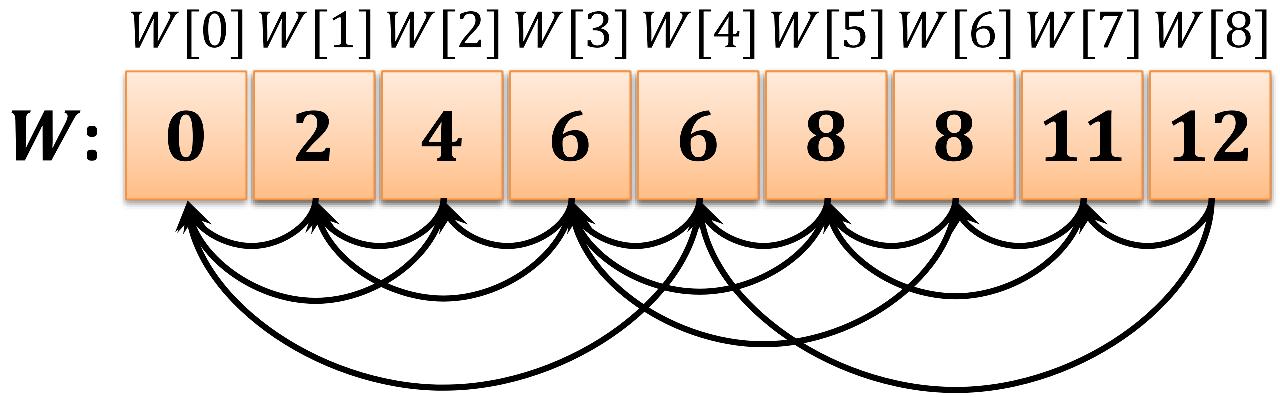
Memoize: *Store* solutions for *subproblems*
reuse the stored solutions if the same subproblems
has to be solved again

Weighted interval scheduling: subproblems $W(1), W(2), W(3)$, …

runtime $=$ #subproblems $\cdot$ time per subproblem

Bottom-Up & Computing the Solution




$W[0]$ $W[1]$ $W[2]$ $W[3]$ $W[4]$ $W[5]$ $W[6]$ $W[7]$ $W[8]$
W: 0 2 4 6 6 8 8 11 12

Computing the schedule: **store where you come from!**

DP: Some History …

- Where das does the name come from?
- DP was developed by Richard E. Bellman in 1940s/1950s.
- In his autobiography, it says:

  "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. … The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. … His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. … Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. … It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. … Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. …"