

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 1

Today's Lecture

Where are we?

- ▶ We've been studying algorithm design.
- ▶ **Greedy algorithms**
 - ▶ Typically fast
 - ▶ But only guaranteed to find optimal answer for a select few problems (e.g., activity scheduling)
- ▶ **Backtracking**
 - ▶ Usually have bad worst case (exponential!)
 - ▶ But are guaranteed to find optimal answer.

Today

- ▶ **Dynamic Programming**: backtracking + memoization.
- ▶ Just as general as backtracking.
- ▶ And for some problems, **massively faster**.
- ▶ A “sledgehammer” of algorithm design.¹

¹Dasgupta, Papadimitriou, Vazirani

Today

- ▶ A new problem: weighted activity scheduling.
- ▶ We'll design a **dynamic programming** solution in steps:
 1. Backtracking solution.
 2. "Nicer" backtracking with repeating subproblems.
 3. Give backtracking algorithm a short-term memory.
- ▶ We'll turn an **exponential** time algorithm to **linear** by adding 2 lines of code.

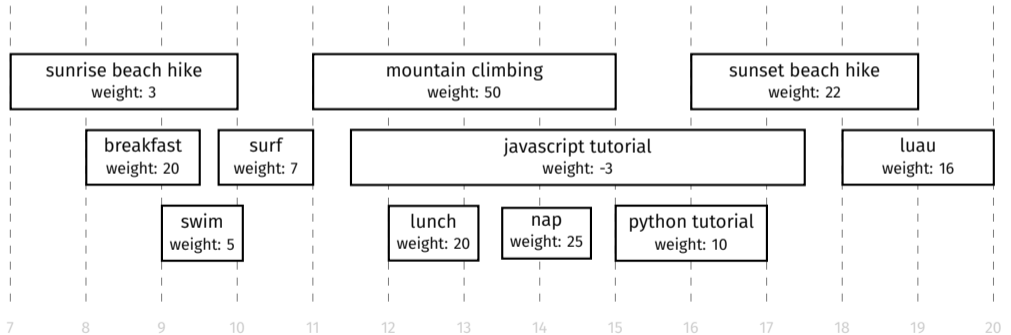
DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 2

Weighted Activity Selection Problem

Vacation Planning



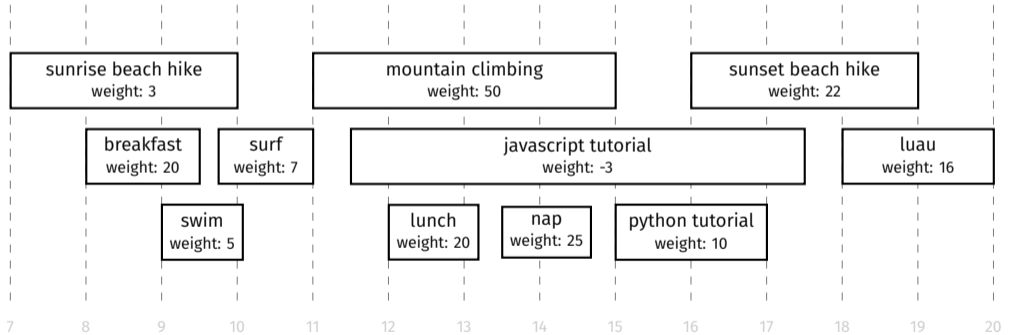
Weighted Activity Selection Problem

- ▶ **Given:** a set of activities each with start, finish, weight.
- ▶ **Goal:** Choose set of compatible activities so as to maximize total weight.

Greedy?

- ▶ Remember the *unweighted* problem: maximize total number of activities.
- ▶ Greedy solution: take compatible activity that finishes earliest, repeat.
- ▶ This was **guaranteed** to find optimal in that problem.
- ▶ It **may not** find optimal for weighted problem.

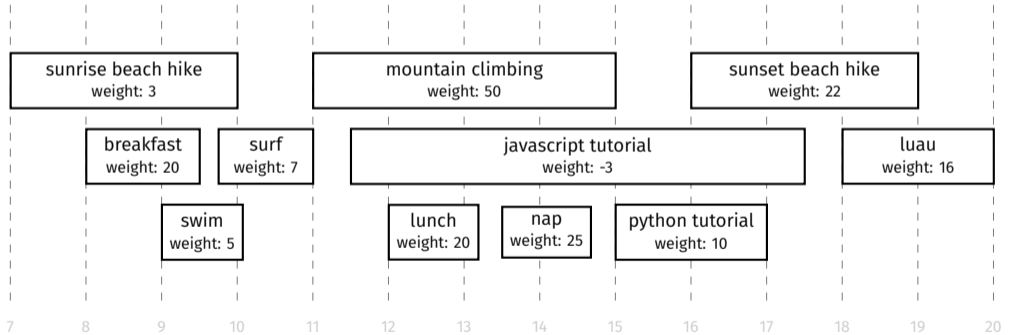
Greedy?



Greedy?

- ▶ Maybe a different greedy approach works?
- ▶ Idea: take compatible activity with **largest weight**.

Greedy?



Don't be greedy!

- ▶ The greedy approach is **not guaranteed** to find best.
- ▶ Note: you might get lucky on a particular instance!

What now?

- ▶ We'll try **backtracking**.
- ▶ It will take **exponential time**.
- ▶ But with a small change, we'll get a **linear time** algorithm that is guaranteed to find the best!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 3

Step 01: Backtracking Solution

Backtracking

- ▶ We'll build up a schedule, one activity at a time.
- ▶ Choose an arbitrary activity, x .
 - ▶ Recursively see what happens if we **do** include x .
 - ▶ Recursively see what happens if we **don't** include x .
- ▶ This will try **all valid schedules**, keep the best.

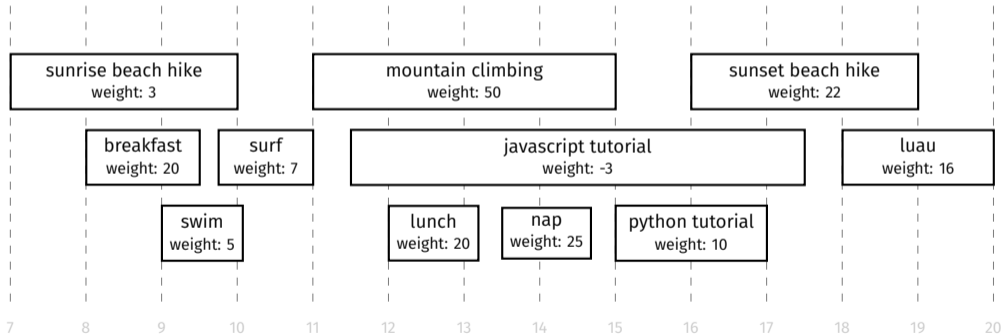
Backtracking

```
def mwsched_bt(activities):  
    if not activities:  
        return 0  
  
    # choose arbitrary activity  
    x = activities.choose_arbitrary()  
  
    # best with x  
    best_with = ...  
  
    # best without x  
    best_without = ...  
  
    return max(best_with, best_without)
```


Recursive Subproblems

- ▶ What is `BEST(activities)` if we assume that `x` **is** in schedule?
- ▶ Imagine choosing `x`.
 - ▶ Your current total weight is `x.weight`.
 - ▶ Activities left to choose from: those **compatible** with `x`.
- ▶ Clearly, you want the best outcome for *new* situation (subproblem).
- ▶ Answer: `x.weight + BEST(activities.compatible_with(x))`

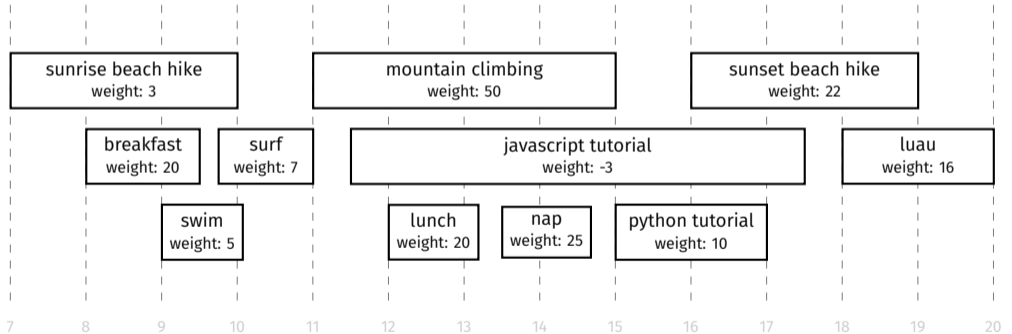
activities.compatible_with(x)



Recursive Subproblems

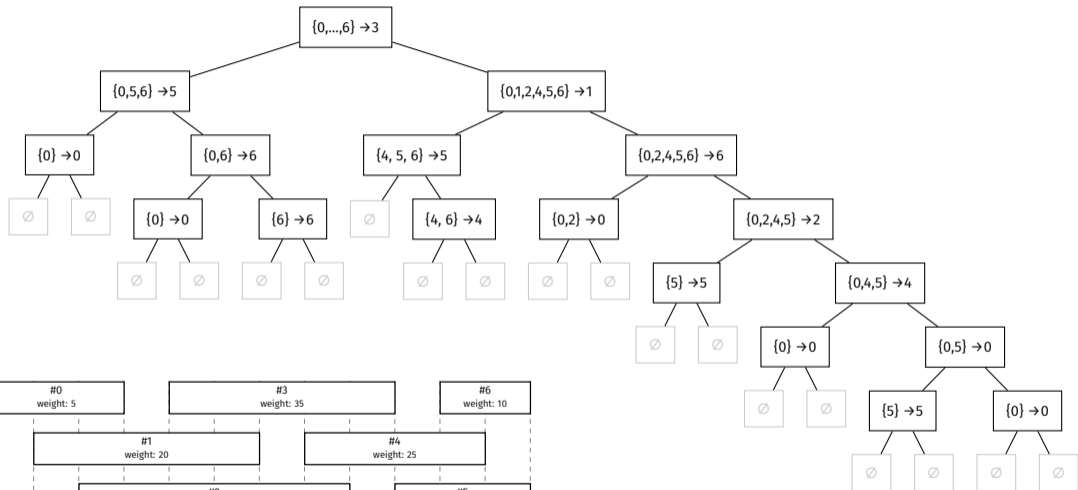
- ▶ What is $\text{BEST}(\text{activities})$ if we assume that x **is not** in schedule?
- ▶ Imagine not choosing x .
 - ▶ Your current total weight is 0 .
 - ▶ Activities left to choose from: all except x .
- ▶ Clearly, you want the best outcome for *new* situation (subproblem).
- ▶ Answer: $\text{BEST}(\text{activities.without}(x))$

activities.without(x)



Backtracking

```
def mwsched_bt(activities):  
    if not activities:  
        return 0  
  
    # choose arbitrary activity  
    x = activities.choose_arbitrary()  
  
    # best with x  
    best_with = x.weight + mwsched_bt(activities.compatible_with(x))  
  
    # best without x  
    best_without = mwsched_bt(activities.without(x))  
  
    return max(best_with, best_without)
```



Efficiency

- ▶ Worst case: recursive calls on problem of size $n - 1$.
- ▶ Recurrence of form $T(n) = 2T(n - 1) + \Theta(\dots)$
- ▶ **Exponential time** in worst case.
- ▶ Could prune, branch & bound, but there's a better way.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 4

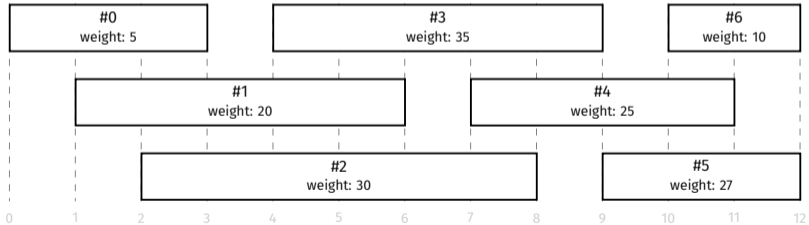
Step 02: A Nicer Backtracking Solution

Arbitrary Choices

- ▶ Our subproblems are arbitrary sets of activities.
 - ▶ E.g., {1, 3, 4, 5, 8, 11, 12}
- ▶ **Now:** If we make choice of next event more carefully, the subproblems look much nicer.
- ▶ Something great happens!

A Nicer Choice

- ▶ Instead of choosing arbitrarily, choose first.



```
def mwsched_bt_nice(activities):
    if not activities:
        return 0

    # choose first activity
    x = activities.choose_first()

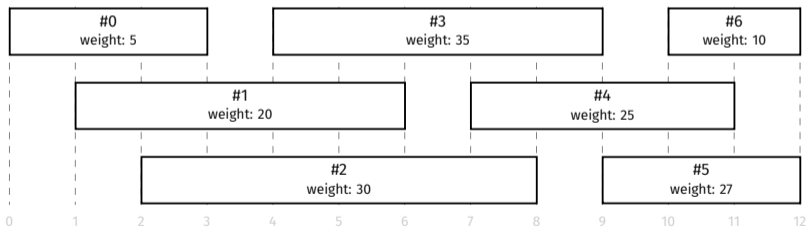
    # best with x
    best_with = x.weight + mwsched_bt(activities.compatible_with(x))

    # best without x
    best_without = mwsched_bt(activities.without(x))

    return max(best_with, best_without)
```

activities.compatible_with(x)

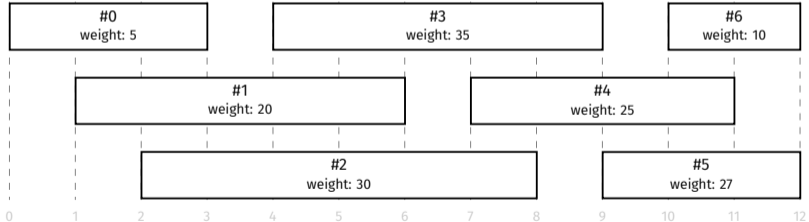
- ▶ Results in a “nice” set of the form $\{i, i + 1, \dots, n - 1\}$ ²



²Assuming x is the activity with first start time.

activities.without(x)

- ▶ Results in a “nice” set of the form $\{i, i + 1, \dots, n - 1\}$ ³



³Assuming x is the activity with first start time.

Representing Remaining Activities

- ▶ Assume events are in sorted order by start time.
- ▶ Subproblems are always of form $\{i, i + 1, i + 2, \dots, n - 1\}$
- ▶ We can specify them with a **single number**, i .

```
def mwsched_bt_nice(activities, first: int=0):
    """Find best schedule using only events in activities[first:]
    Assumes activities sorted by start time.
    """
    if first >= len(activities):
        return 0

    # choose first event
    x = activities[first]

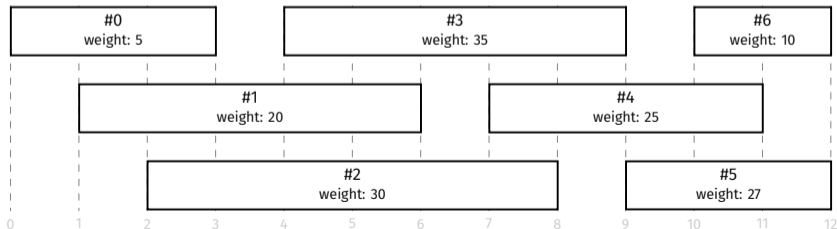
    # best with x
    next_compatible = index_of_next_compatible(activities, after=first)
    best_with = x.weight + mwsched_bt_nice(activities, next_compatible)

    # best without x
    best_without = mwsched_bt_nice(activities, first + 1)

    return max(best_with, best_without)
```

index_of_next_compatible()

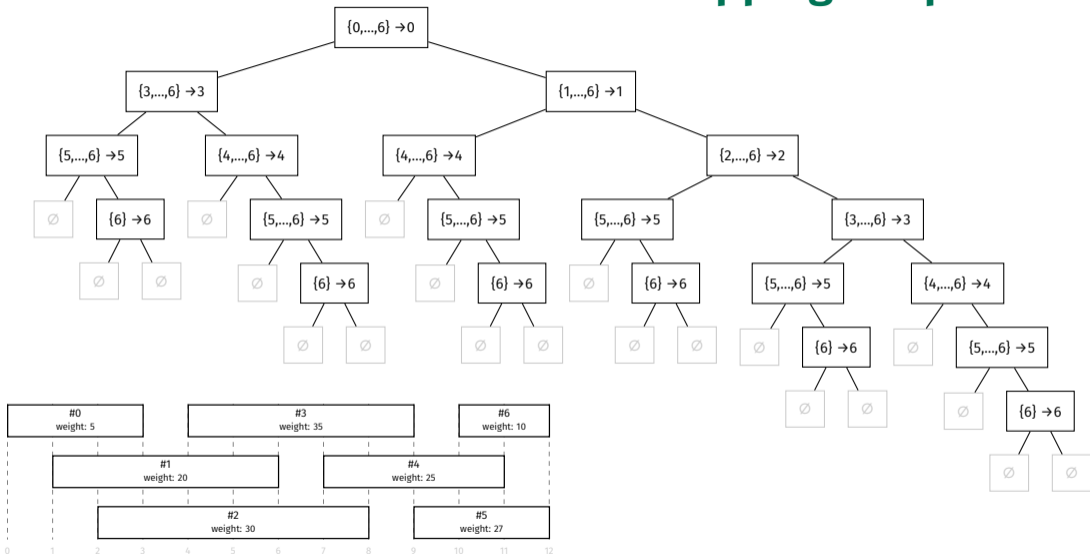
```
def index_of_next_compatible(activities, after: int):  
    """Find index of first event starting after `after` ends.  
    Assumes activities sorted by start time.  
    """  
    for j in range(after + 1, len(activities)):  
        if activities[j].start >= activities[after].finish:  
            return j  
    return len(activities)
```



What did we gain?

- ▶ Can specify subproblems with integers instead of sets.
 - ▶ Saves memory.
- ▶ But there's an **even better** consequence!

Overlapping Subproblems

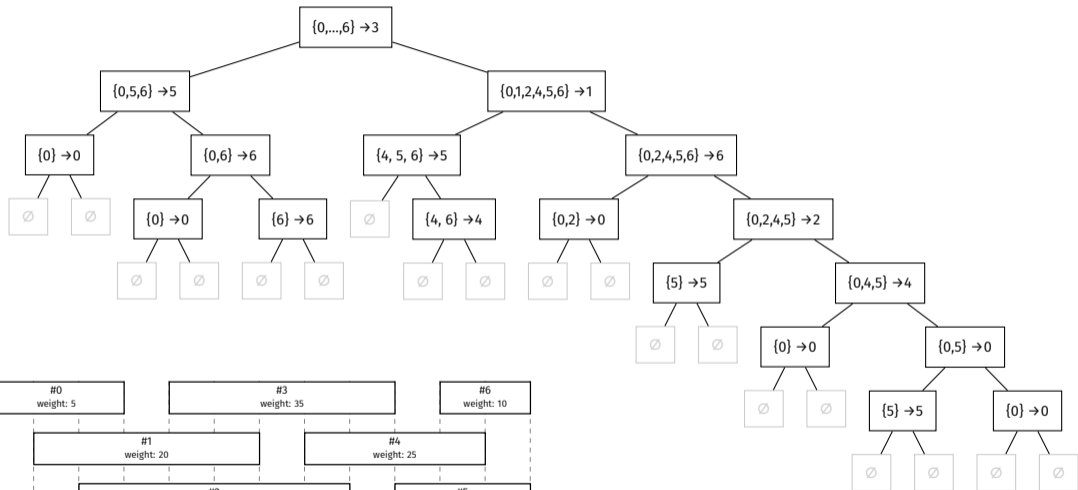


Overlapping Subproblems

- ▶ Backtracking doesn't have a memory.
- ▶ It will happily solve same subproblem over and over, getting same result each time.
- ▶ We'll speed it up by giving it a memory.

Note

- ▶ Overlapping subproblems are a consequence of this more careful choice of event.
- ▶ When we chose arbitrarily, we didn't have (as many) overlapping subproblems.



DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 5

Step 03: Memoization

Backtracking + Memoization

- ▶ By making careful choices, we've found a backtracking solution with many overlapping subproblems.
- ▶ Idea: solve subproblem once, save the result!
- ▶ This is called **memoization**⁴.

⁴Not “memorization”. That would make too much sense.

Memoization

- ▶ Keep a **cache**: dictionary or array mapping subproblems to solutions.
- ▶ Before solving a subproblem, check if already in cache.
- ▶ After solving a subproblem, save result in cache.


```
def mwsched_dp(activities, first: int=0, cache=None):
    """Find best schedule using events in activities[first:].
    Assumes activities sorted by start time."""

    if cache is None: # cache[i] is solution of activities[i:]
        cache = [None] * len(activities)

    if first >= len(activities):
        return 0

    # save some work if we've already computed this
    if cache[first] is not None:
        return cache[first]

    # choose first event
    x = activities[first]

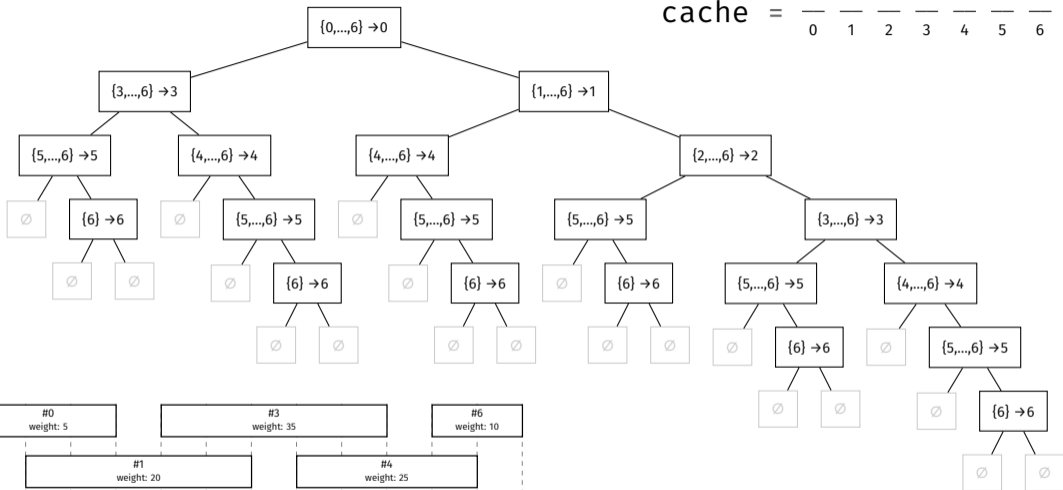
    # best with x
    next_compatible = index_of_next_compatible(activities, after=first)
    best_with = x.weight + mwsched_dp(activities, next_compatible, cache=cache)

    # best without x
    best_without = mwsched_dp(activities, first + 1, cache=cache)

    best = max(best_with, best_without)

    # store result in cache for future reference
    cache[first] = best
    return best
```

cache = $\frac{\quad}{0} \frac{\quad}{1} \frac{\quad}{2} \frac{\quad}{3} \frac{\quad}{4} \frac{\quad}{5} \frac{\quad}{6}$



Time Complexity

- ▶ There are only n subproblems.
 - ▶ $\{0, \dots, n - 1\}, \{1, \dots, n - 1\}, \dots, \{n - 1\}$
- ▶ Solve each one once.
- ▶ The memoized solution takes $\Theta(n)$ time.

Dynamic Programming

- ▶ This approach (backtracking + memoization) is called “top-down” **dynamic programming**.
- ▶ Often reduces time from exponential to polynomial.

DSC 190

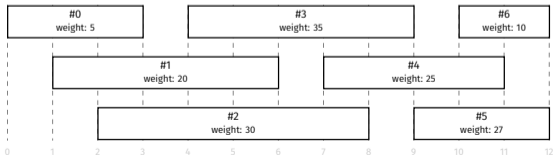
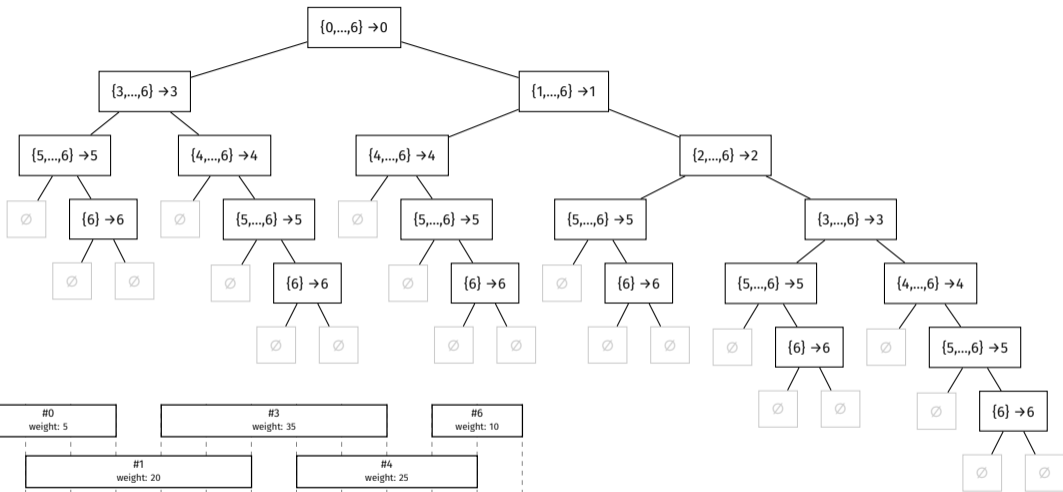
DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 6

Top-Down vs. Bottom-Up

Top-Down

- ▶ Backtracking + memoization is known as “top down” dynamic programming.
- ▶ We start at top level problem, recursively find subproblems.
- ▶ But we can start from bottom-level problems, too.



Bottom-Up

- ▶ The top-down recursive code solves problems in order:
 - ▶ $\{6\}, \{5, 6\}, \{4, \dots, 6\}, \{3, \dots, 6\}, \{2, \dots, 6\}, \{1, \dots, 6\}, \{0, \dots, 6\}$
- ▶ The bottom-up approach starts with easiest subproblem, iteratively solves harder subproblems.
- ▶ Solve $\{6\}$. Use it to solve $\{5, 6\}$. Use this to solve $\{4, \dots, 6\}$, etc.


```

def mwsched_bottom_up(activities):
    """Assumes activities sorted by start time."""
    n = len(activities)

    # best[i] is the weight of the best possible schedule that can be formed
    # using activities[i:]. best[n] is a dummy value; it represents the "base case"
    # solution of zero. best[0] is solution to the full problem.
    best = [None] * (n + 1)
    best[n] = 0

    # solve easiest subproblem: when we have one event, activities[n-1]
    best[n-1] = activities[n-1].weight

    # iteratively solve subproblems from small to big,
    # using solutions of smaller problems in solving big
    for first in reversed(range(n-1)):
        x = activities[first]

        # best with
        next_compatible = index_of_next_compatible(activities, after=first)
        best_with = x.weight + best[next_compatible]

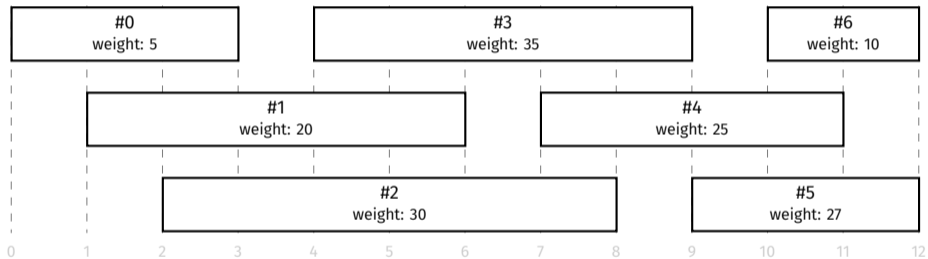
        # best without
        best_without = best[first + 1]

        best[first] = max(best_with, best_without)

    return best[0]

```

Example



best = $\frac{\quad}{0} \frac{\quad}{1} \frac{\quad}{2} \frac{\quad}{3} \frac{\quad}{4} \frac{\quad}{5} \frac{\quad}{6} \frac{\quad}{7}$

Which to use?

- ▶ Bottom-up and top-down will generally have same time complexity.
- ▶ Top-down arguably easier to design.
- ▶ Bottom-up avoids overhead of recursion.
- ▶ But bottom-up may solve unnecessary subproblems.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 7

Dynamic Programming

When can we use it?

- ▶ Memoization can be added to *any* backtracking algorithm.
- ▶ But it is only *useful* if there are **overlapping subproblems**.
- ▶ Not all problems yield overlapping subproblems.

How do we design them?

- ▶ General strategy for top-down:
 1. Write a backtracking solution.
 2. Modify backtracking solution to get overlapping subproblems that are “easy to describe”.⁵
 3. Add memoization.

- ▶ “Expert mode”: identify recursive substructure immediately.

- ▶ Can be tricky; need to be creative.

⁵Easier said than done.

How do we design them?

- ▶ General strategy for bottom-up:
 1. Write a top-down dynamic programming solution.
 2. Analyze the order in which cache is filled in.
 3. Iteratively solve subproblems in this order.

Are they guaranteed to be optimal?

- ▶ **Yes!** Dynamic programming *is* a form of backtracking, so it is guaranteed to find an optimal solution.

Is it at all useful for data science?

- ▶ **Yes!**
- ▶ Next time: the longest common subsequence problem and its applications to “fuzzy” string matching, DNA string comparison.
- ▶ Future (maybe): Hidden Markov Models, All-Pairs Shortest Paths