DSC 190

DATA STRUCTURES & ALGORITHMS

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]

[1] Dasgupta, 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.

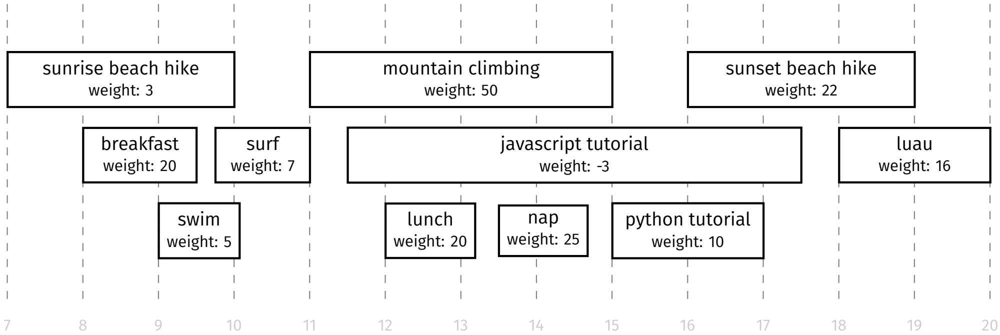
- We'll turn an **exponential** time algorithm to **linear** by adding 2 lines of code.

DSC 190

DATA STRUCTURES & ALGORITHMS

Weighted Activity Selection Problem

Vacation Planning



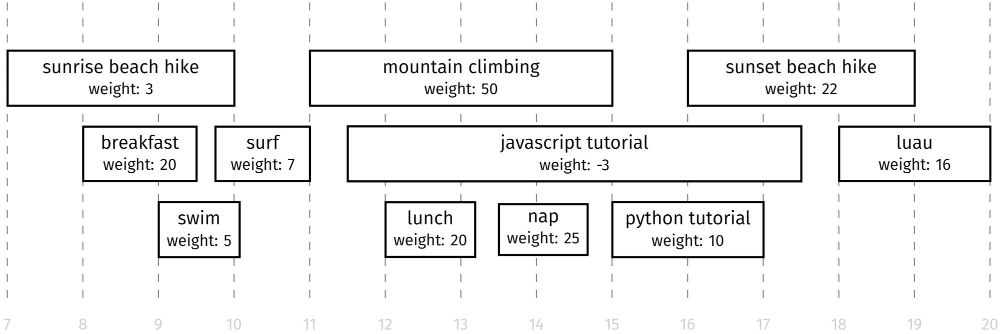
Weighted Activity Selection Problem

- **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.

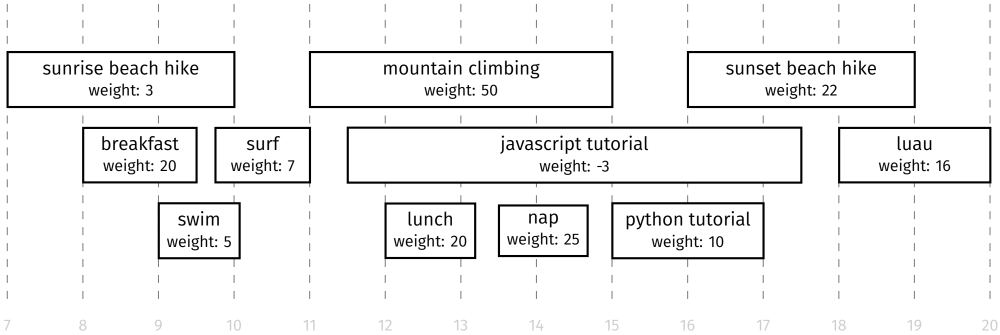
Greedy?


sunrise beach hike
weight: 3
mountain climbing
weight: 50
sunset beach hike
weight: 22
breakfast
weight: 20
surf
weight: 7
javascript tutorial
weight: -3
luau
weight: 16
swim
weight: 5
lunch
weight: 20
nap
weight: 25
python tutorial
weight: 10
7
8
9
10
11
12
13
14
15
16
17
18
19
20

Greedy?

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

Greedy?


sunrise beach hike
weight: 3
mountain climbing
weight: 50
sunset beach hike
weight: 22
breakfast
weight: 20
surf
weight: 7
javascript tutorial
weight: -3
luau
weight: 16
swim
weight: 5
lunch
weight: 20
nap
weight: 25
python tutorial
weight: 10
7
8
9
10
11
12
13
14
15
16
17
18
19
20

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

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.

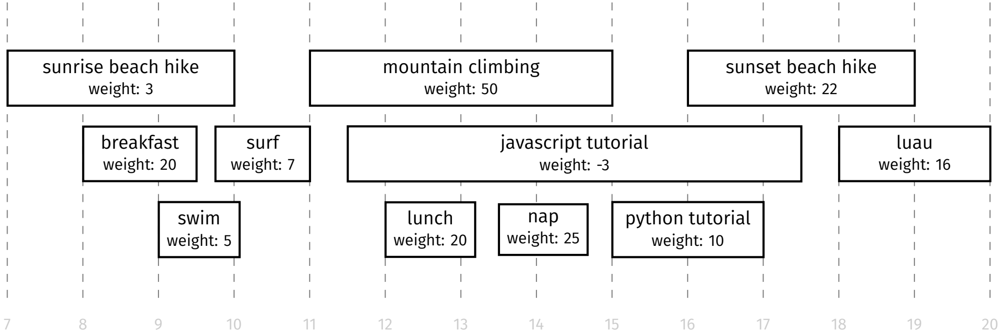
Backtracking

```python
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

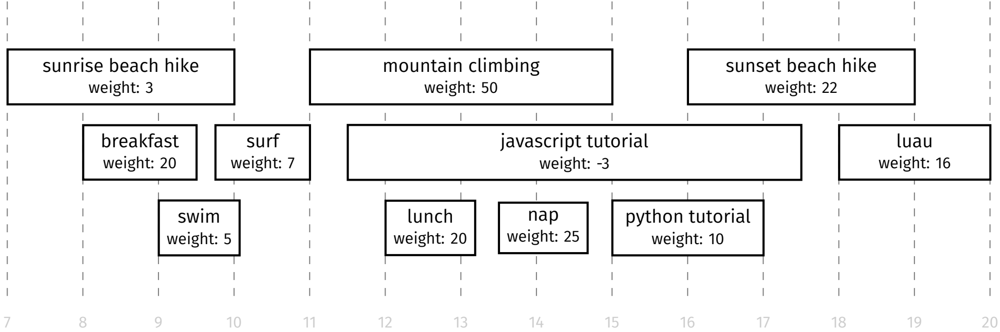
- What is BEST(`activities`) if we assume that `x` **is not** in schedule?

- 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).

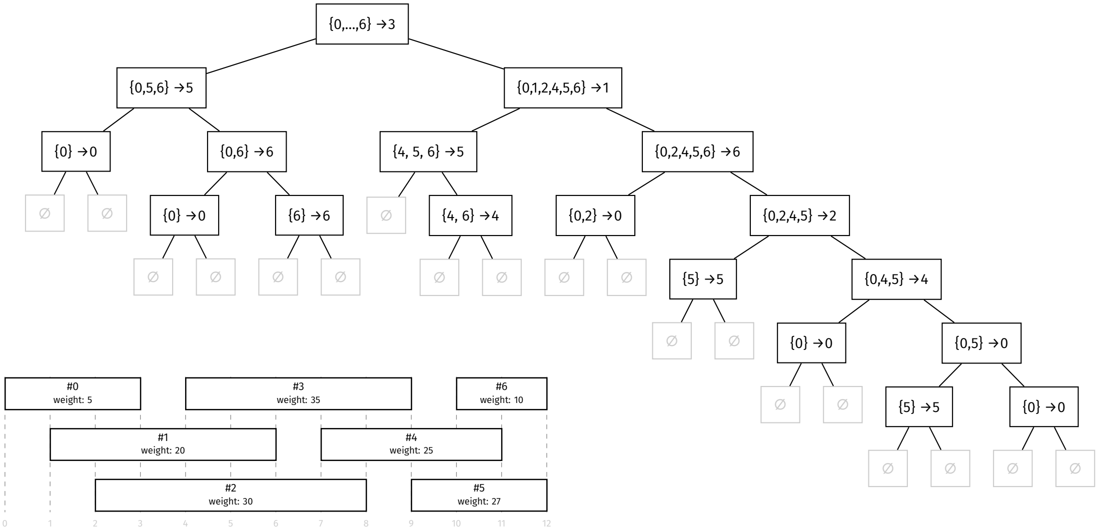
- Answer: BEST(`activities.without(x)`)

activities.without(x)



Backtracking

```python
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)
```


{0,...,6} →3
{0,5,6} →5
{0,1,2,4,5,6} →1
{0} →0
{0,6} →6
{4, 5, 6} →5
{0,2,4,5,6} →6
∅
{0} →0
{6} →6
{4, 6} →4
{0,2} →0
{0,2,4,5} →2
{5} →5
{0,4,5} →4
{0} →0
{0,5} →0
{5} →5
{0} →0
#0
weight: 5
#3
weight: 35
#6
weight: 10
#1
weight: 20
#4
weight: 25
#2
weight: 30
#5
weight: 27
0
1
2
3
4
5
6
7
8
9
10
11
12

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

Step 02: A Nicer Backtracking Solution

Arbitrary Choices

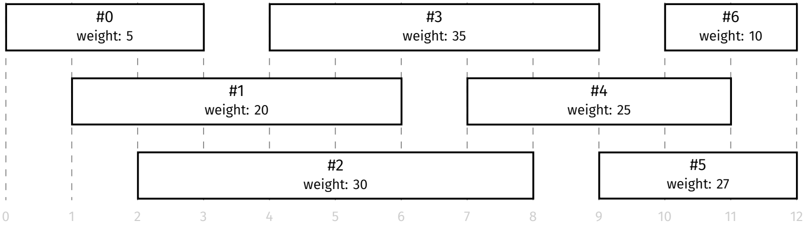
- Our subproblems are arbitrary sets of activities.
  - E.g., {1, 3, 4, 5, 8, 11, 12}

- **Now**: If we make choice of next event more carefully, the subproblems look much nicer.

- Something great happens!

A Nicer Choice

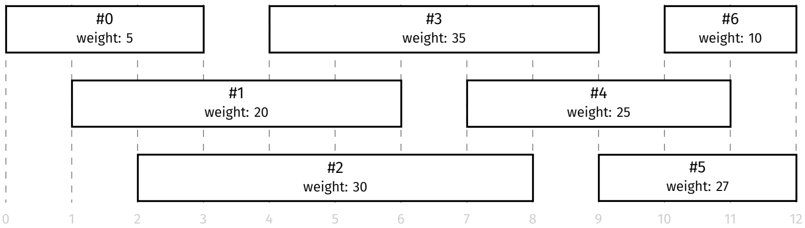
- Instead of choosing arbitrarily, choose first.



```python
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)`

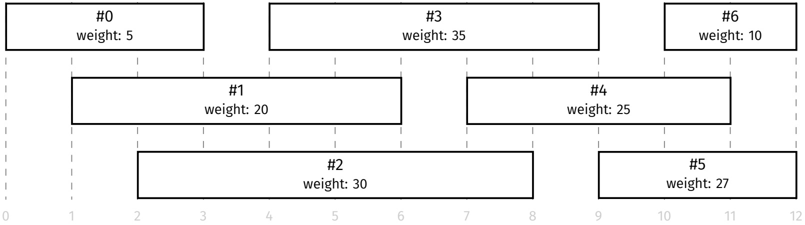
- Results in a "nice" set of the form $\{i, i+1, \ldots, n-1\}$[2]



[2] Assuming x is the activity with first start time.

activities.without(x)

- Results in a "nice" set of the form $\{i, i+1, \ldots, n-1\}$[3]



[3] Assuming x is the activity with first start time.

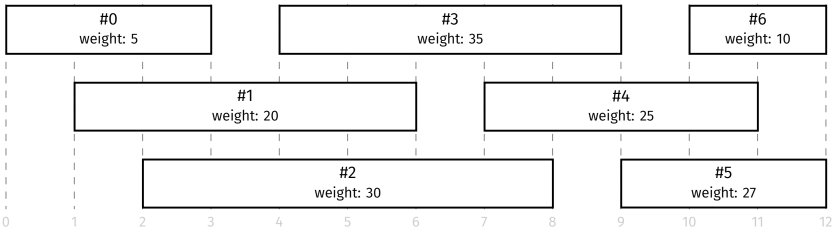
Representing Remaining Activities

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

```python
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()

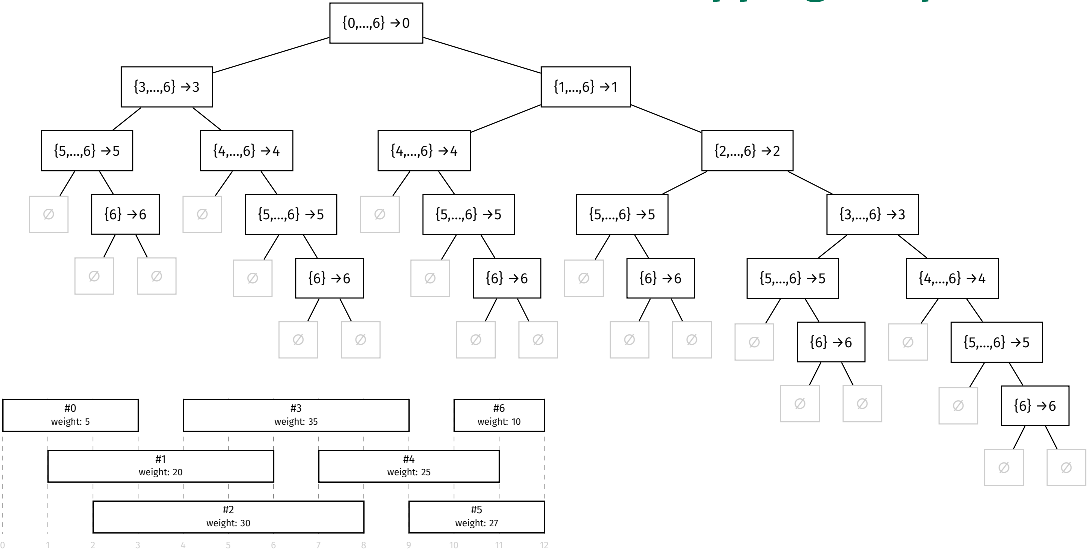
```python
def index_of_next_compatible(activities, after: int):
    """Find index of first event starting after `after` ends.
    Assumes activities sorted by start time.
    """
    for j in range(after + 1, len(activities)):
        if activities[j].start >= activities[after].finish:
            return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



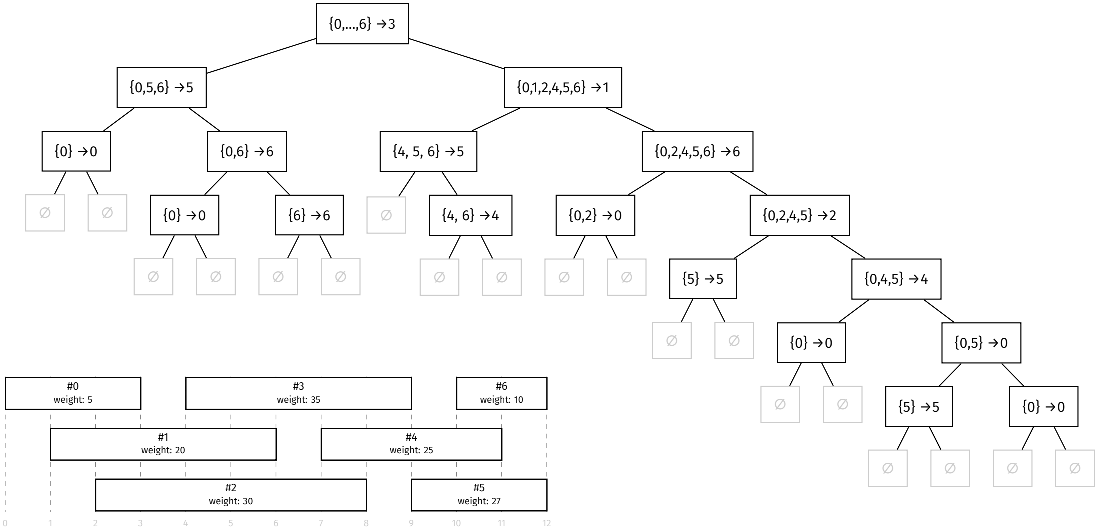
Overlapping Subproblems

- 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.


{0,...,6} →3
{0,5,6} →5
{0,1,2,4,5,6} →1
{0} →0
{0,6} →6
{4, 5, 6} →5
{0,2,4,5,6} →6
{0} →0
{6} →6
{4, 6} →4
{0,2} →0
{0,2,4,5} →2
{5} →5
{0,4,5} →4
{0} →0
{0,5} →0
{5} →5
{0} →0
#0
weight: 5
#3
weight: 35
#6
weight: 10
#1
weight: 20
#4
weight: 25
#2
weight: 30
#5
weight: 27
0 1 2 3 4 5 6 7 8 9 10 11 12

DSC 190

DATA STRUCTURES & ALGORITHMS

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].

[4] Not "memorization". That would make too much sense.

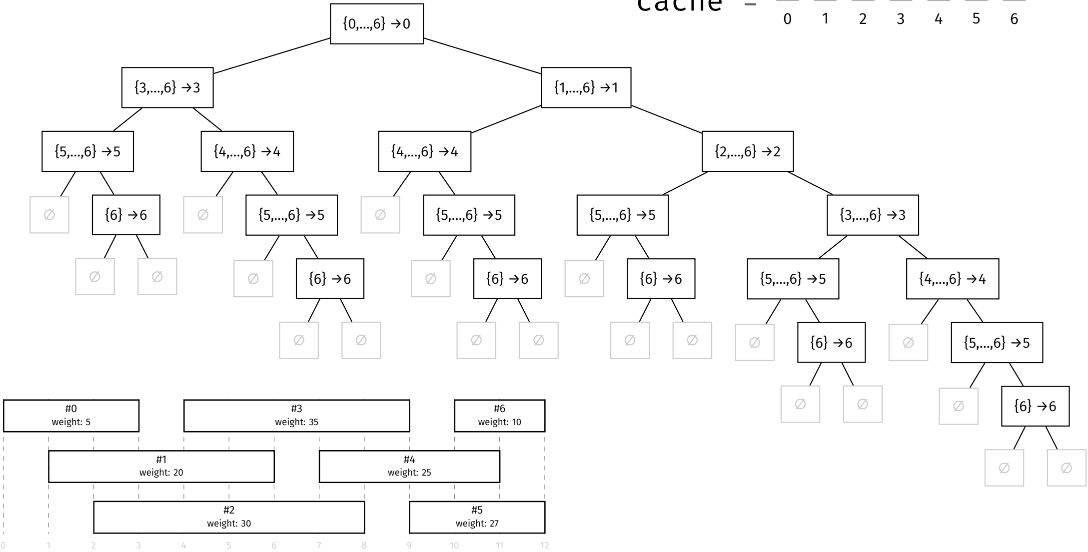
Memoization

- 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.

```python
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_bt_nice(activities, next_compatible)

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

    best = max(best_with, best_without)

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

cache = __ __ __ __ __ __ __

0 1 2 3 4 5 6



Time Complexity

- There are only n subproblems.
  - $\{0, \ldots, n-1\}, \{1, \ldots, n-1\}, \ldots, \{n-1\}$
- 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.

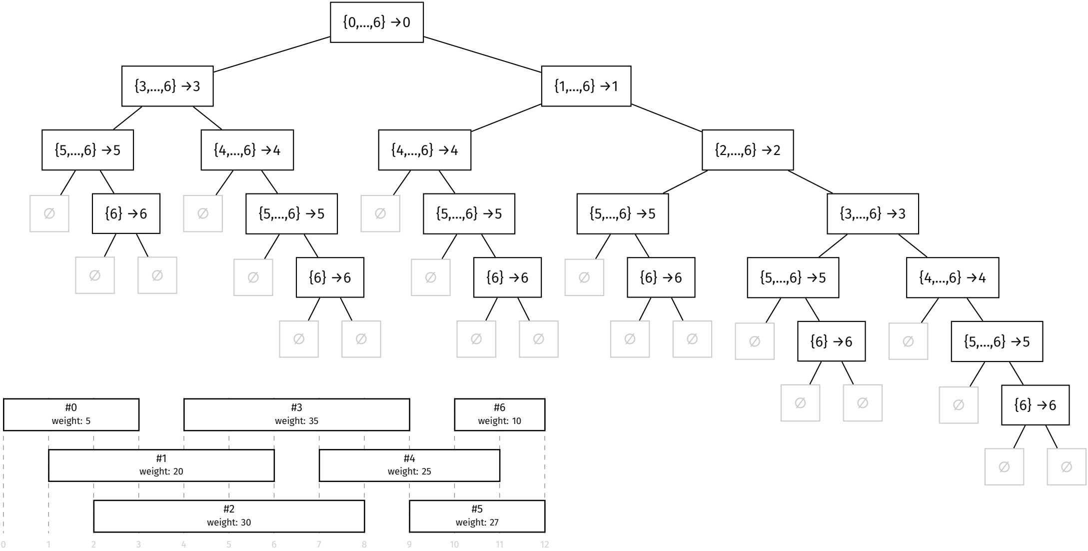
DSC 190

DATA STRUCTURES & ALGORITHMS

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.


{0,...,6} →0
{3,...,6} →3
{1,...,6} →1
{5,...,6} →5
{4,...,6} →4
{4,...,6} →4
{2,...,6} →2
∅
{6} →6
∅
{5,...,6} →5
∅
{5,...,6} →5
{5,...,6} →5
{3,...,6} →3
∅
∅
∅
{6} →6
∅
{6} →6
∅
{6} →6
{5,...,6} →5
{4,...,6} →4
∅
∅
∅
∅
∅
∅
∅
{6} →6
∅
{5,...,6} →5
∅
∅
∅
{6} →6
∅
∅
#0
weight: 5
#3
weight: 35
#6
weight: 10
#1
weight: 20
#4
weight: 25
#2
weight: 30
#5
weight: 27
0
1
2
3
4
5
6
7
8
9
10
11
12

Bottom-Up

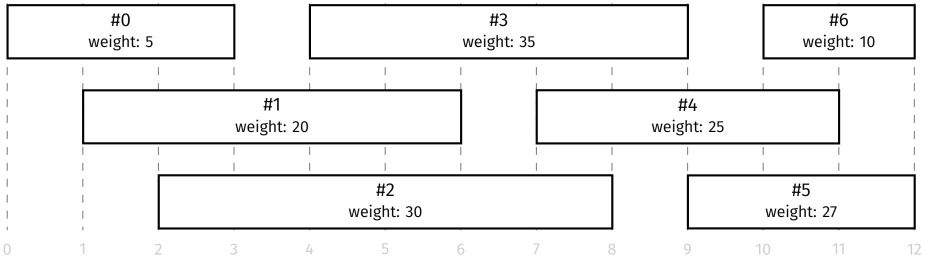
- The top-down recursive code solves problems in order:
  - $\{6\}, \{5, 6\}, \{4, \ldots, 6\}, \{3, \ldots, 6\}, \{2, \ldots, 6\}, \{1, \ldots, 6\}, \{0, \ldots, 6\}$

- The bottom-up approach starts with easiest subproblem, iteratively solves harder subproblems.

- Solve $\{6\}$. Use it to solve $\{5, 6\}$. Use this to solve $\{4, \ldots, 6\}$, etc.

```python
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` = __ __ __ __ __ __ __ __

0 1 2 3 4 5 6 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

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.

[5] Easier 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