Intorduction to Dynamic Programming

About me

- Carlos Gonzalez Oliver
- **carlos@ozeki.io** (mailto:carlos@ozeki.io)
- **carlosoliver.co** (http://carlosoliver.co)

My interests:
- Structural Bioinformatics
- Graph algorithms
- Representation learning

Some questions I ask:
- *How can we efficiently search through graph databases?*
- *What kinds of patterns can we discover in datasets of graphs?*
- *Can we model biological systems using efficient data structures to learn about how they function?*

Dynamic Programming (DP)

	• Roger Bellman, 1950s, working at RAND institute. • Picked the name “Dynamic Programming” to please his boss.

- Method for solving problems by constructing the solution bottom-up.
  Applications
  - Bioinformatics: protein and RNA folding, sequence alignment
  - Speech recognition: Viterbi’s algorithm
  - Time series modeling: Dynamic time warping
  - Search: string matching
  - Scheduling: weighted interval
  - Music: Beat tracking
  - Routing: shortest path problems

Objectives

After this talk we should be able to:

- [ ] Recognize the **ingredients** needed to solve a problem with DP.
- [ ] Write down and execute some simple DP algorithms.
- [ ] Be able to implement a DP algorithm for a real-world problem (homework).

Basic Intuition

- What is:
  - $1 + 1 + 1 + 1 + 1 + 1 = ?$
- Now what is:
  - $1 + 1 + 1 + 1 + 1 + 1 + 1 = ?$

First ingredient: Optimal Substructures

> The optimal structure can be built by combining optimal solutions to smaller problems.

E.g. Fibonacci numbers

- $Fib(0) = 1$
- $Fib(1) = 1$
- $Fib(n) = Fib(n-1) + Fib(n-2)$

Second ingredient: Overlapping Substructures

> The same subproblem's solution is used multiple times.

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Call tree `fib(6)` :

Recursive solution is $\mathcal{O}(2^n)$.

An example *without* overlapping subproblems.

Binary Search

```
def binary_search(A, left, right, query):
    if low > high:
        return None

    mid = (left + right) // 2
    if A[mid] == query:
        return i
    elif A[mid] > query:
        return binary_search(A, left, mid-1, query)
    else:
        return binary_search(A, mid+1, right, query)
```

Call tree `binary_search(A=[ 15, 22, 32, 36, 41, 63, 75], left=0, right=4, query=75)` :

Case study: Weighted Interval Scheduling

- **Input:** n requests, labeled $\{1, .., n\}$. Each request has a start time s_i and an end time f_i, and a **weight** v_i.
- **Output:** a *compatible* subset S of $\{1, .., n\}$ that maximizes $\sum_{i \in S} v_i$.

S is compatible iff all pairs of intervals in S are non-overlapping.

- Uses: resource allocation for computer systems, optimizing course selection.

Example:

False start: greedy approach

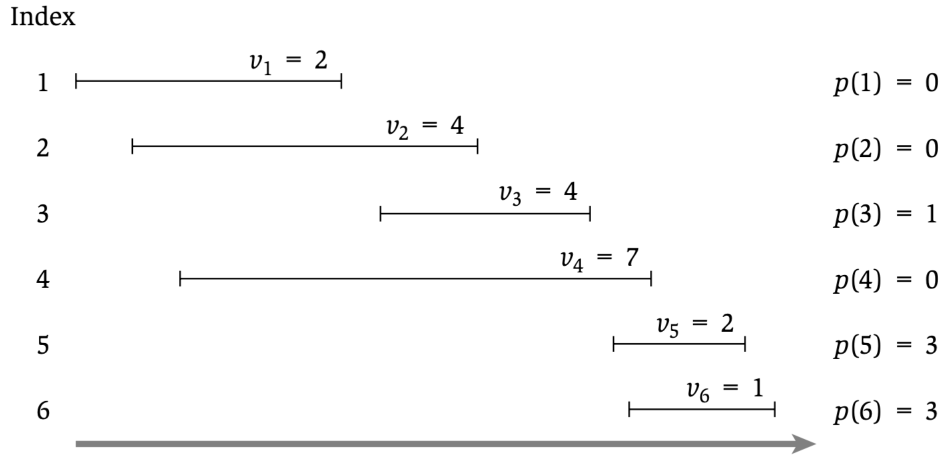
- From "previous" lectures we know the greedy approach to the unweighted IS problem gives the optimal solution (i.e. pick item with earliest ending time)

Counterexample:

The weights force us to consider all possible subproblems (i.e. local choice is not enough).

A helper function

Let us sort all intervals by increasing *finish* time and let $p(j)$ return the latest interval i that is still compatible with j.



First ingredient: Optimal Substructure?

- Consider $\mathcal{O}_n$ to be the *optimal* set of requests over all items n.
- There are two cases for the last request, n.:

  Case 1

  Case 1

  Implication?

Optimal Substructure

Let $\textrm{OPT}(j)$ be the total weight of the optimal over intevals up to j.

We can now write an expression for computing \(\textrm{OPT}(j)\):

Ingredients

- [x] Optimal Substructure
- [ ] Overlapping Subproblems

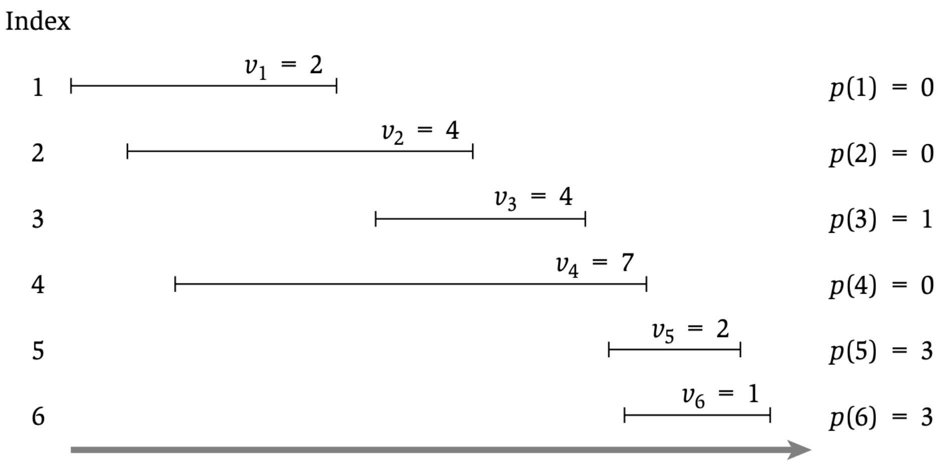
First algorithm

Now we can write down a recursive algorithm that gives us the maximum weight over \(1, ..., n\).

`compute_OPT(j):`

Ingredient 2: Overlapping Substructures

Let's build the execution tree for our recursive algorithm on this example:



Runtime:

Ingredients

- [x] Optimal Substructure
- [x] Overlapping Subproblems

Memoization

- Key idea in DP: remember solutions to sub-problems and iterate such that you already have the solution you need.

New algo: `M_compute_OPT(j)`

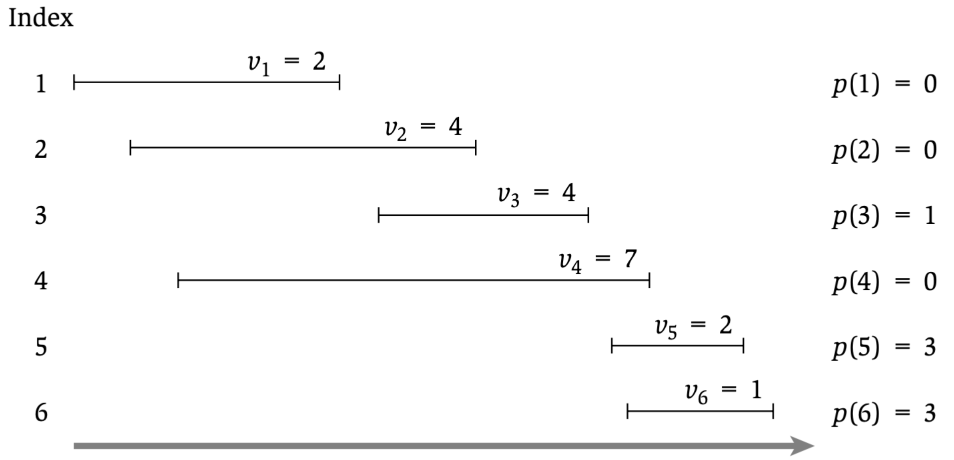
Runtime:

Are we done?

- Recall $\mathrm{OPT}(j)$ is just a number, we want the *set* of intervals with the score $\mathrm{OPT}(j)$... i.e. $\mathcal{O}_j$.
- **Traceback** is a key idea in DP. We reconstruct the solution backwards from the M array using the recurrence.
- **Obervation:** We know that an interval j belongs to $\mathcal{O}_j$ if:

Now we just apply this rule going backwards from $j = n$ and collect optimal intervals until we reach 0.

Full Example



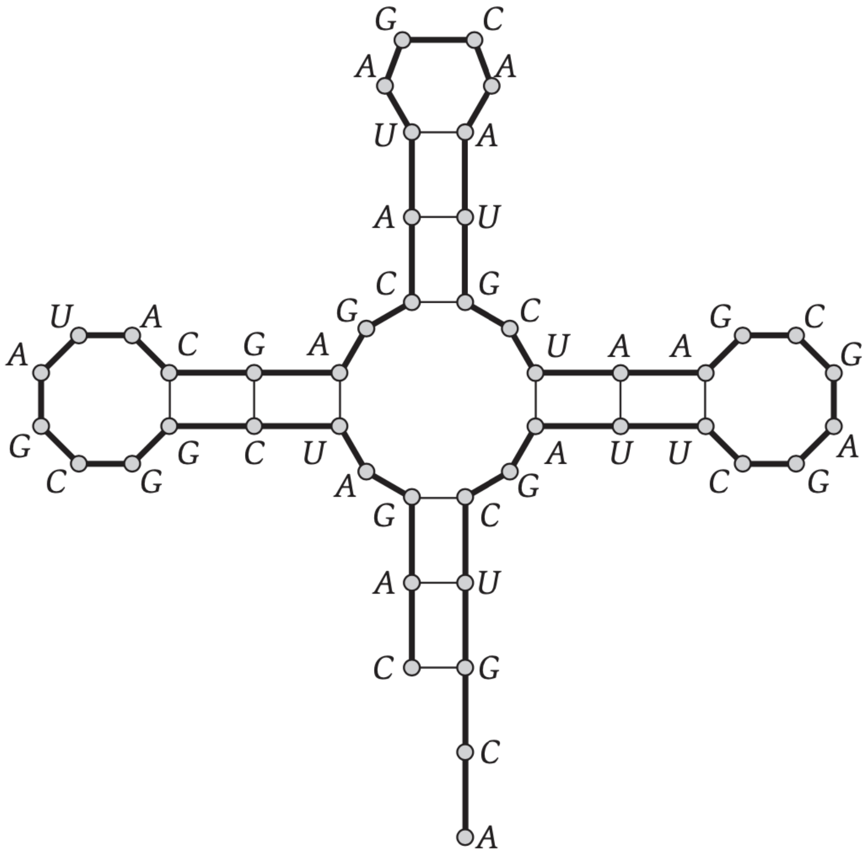
Fill M and get $\mathcal{O}_n$:

Recap

- Dynamic Programming works well when we have a problem structure such that:
  - Combining sub-problems solves the whole problem
  - Sub-problems overlap
- We can often reduce runtime complexity from exponential to polynomial or linear.

General interest: RNA folding

- RNA molecules are essential to all living organisms.
- Knowing the sequence is easy but not so informative, knowing the structure is hard but tells us a lot about the molecule’s function.

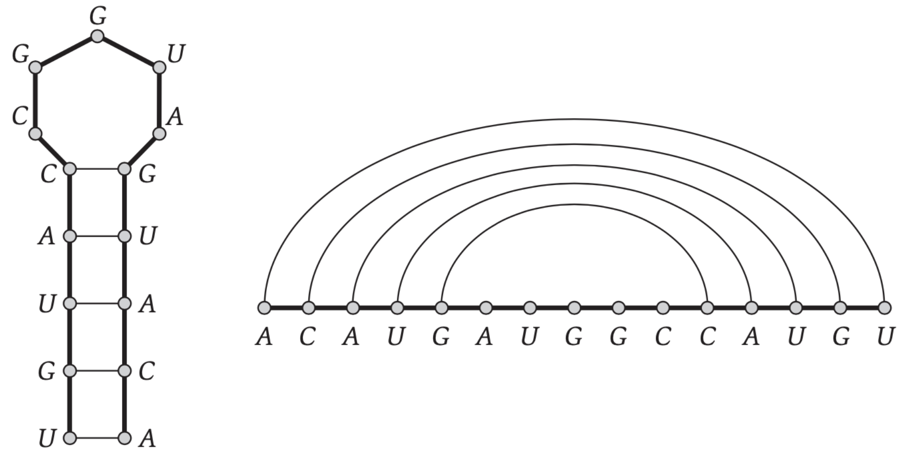


A Sketch of Nussinov’s Algorithm

- First attempt at solving this problem

	Ruth Nussinov, designed the first RNA folding algo in 1977.

- In CS terms, an RNA is a string on a 4-letter alphabet.
- An RNA **structure** is a pair of indices over the string with certain constraints.



RNA folding rules

We use some fairly realistic constraints on admissible structures:

1. Only `A–U` , `U–A` and `C–G` , `G–U` form pairs
2. Pairs shall not cross (nestedness).
3. Start and end of a pair should be separated by at least \(\theta\) spaces (remove steric clashes).
4. The best structure is the one that forms the most pairs (stability). e.g each pair adds 1 to the score.

This lets us identify the problem structure needed for a DP solution:
Consider an optimal set of pairs \(\mathcal{O}_{ij}\) between two indices \((i, j)\), and the score of the olution \(\mathrm{OPT}(i, j)\). We have two cases for index \(j\):
Case 1. \(j\) is not in \(\mathcal{O}_{ij}\):

Note we introduce a new variable \(\rightarrow\) 2 dimensional DP.
Case 2. \(j\) is in \(\mathcal{O}_{ij}\):

Ingredients?

- [ ] Optimal substructures

- [ ] Overlapping subproblems

From this we can build our recurrence that fills the table up to \(\mathrm{OPT}(1, L)\).

Bonus Questions: What is the runtime for filling the table?

> If you want an exercise sheet to learn how to implement this send me an email `carlos@ozeki.io`.