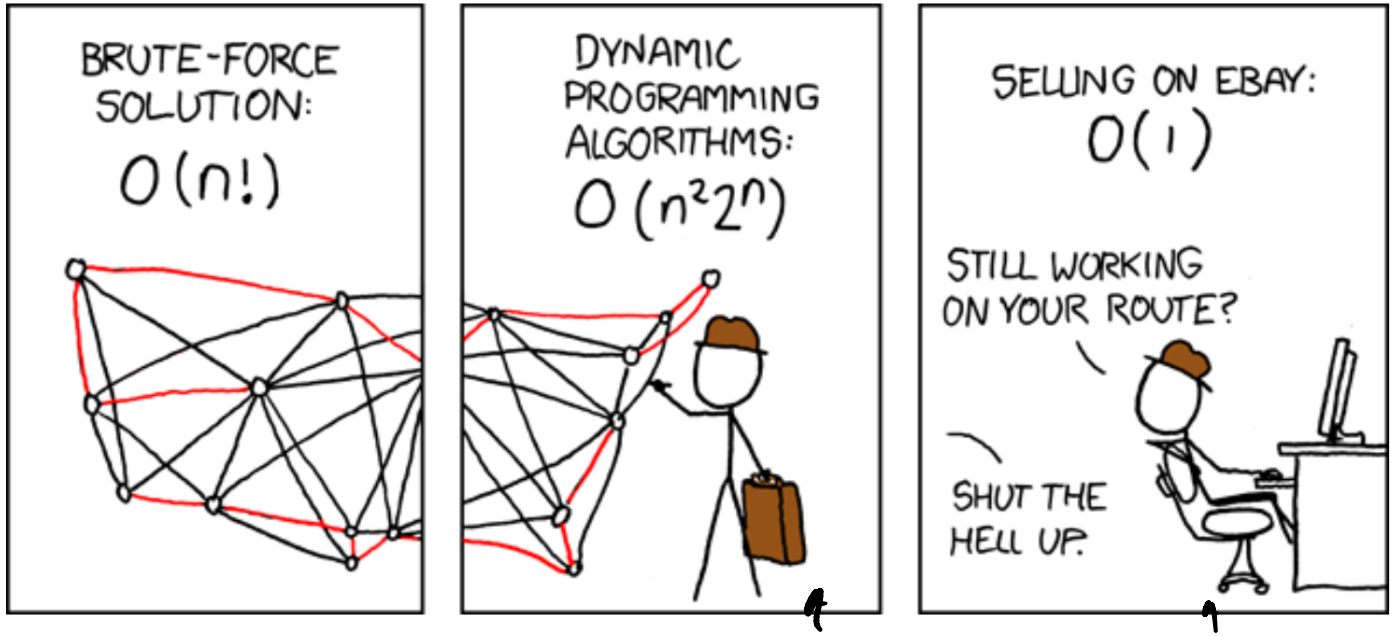
CSCI 3104: Algorithms

Lecture 18: Intro to Dynamic Programming

Rachel Cox

Department of Computer Science



Dynamic Programming can really speed up your work. But common sense can speed things up even further. (Traveling Salesman problem webcomic by XKCD)

What will we learn today?

- ❑ Intro to Dynamic Programming
- ❑ Memoization & Subproblems
- ❑ Fibonacci numbers
- ❑ Knapsack Problem

Intro to Algorithms, CLRS:
Sections 15.1

Intro to Dynamic Programming

Question: What is Dynamic Programming in a nutshell?

- Algorithmic Paradigm that breaks a problem into subproblems and store the results of subproblems to avoid re-computation.

Properties of Problems that can be Solved with Dynamic Programming

- Overlapping subproblems
- Optimal substructure

Intro to Dynamic Programming – Origin

From Wikipedia

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" 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". I'm not using the term lightly; I'm using it precisely. 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. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. 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. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

— Richard Bellman, *Eye of the Hurricane: An Autobiography* (1984, page 159)

Richard Ernest Bellman[1]

- DP is essentially a mathematical optimization method

Intro to Dynamic Programming

Two Approaches for Storing Data

1. Tabulation — Bottom-up Approach.
   - Idea: Solve subproblems first, build solutions to find solutions to larger subproblems
2. Memoization — Top-Down Approach: Recursion + Memoization

Knapsack Problem

} Fibonacci numbers

Intro to Dynamic Programming

What is Memoization? "memo – izatin"

- similar to recursive algorithm, but we check a lookup table to see if result has been computed.

> *Memoization is a way to lower a function's* **time** *cost in exchange for* **space** *cost*

> *In computing,* ***memoization*** *or* ***memoisation*** *is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.*

Comes from the Latin word: memorandum
US: "memo"

Source

❖ Meaning of word: turning the results of a function into something to be remembered.

Intro to Dynamic Programming

Exponential runtime

Example: The Fibonacci sequence

1 1 2 3 5 8 13 ...

$F_n = F_{n-1} + F_{n-2}$

Naïve Recursive Algorithm:

```
def fib(n):
    if n=1 or n=2:
        f=1
    else:
        f = fib(n-1) + fib(n-2)
                ↖       ↗ recursive function call
    return f
```

Runtime analysis:

Hand wavy ~

$T(n) = T(n-1) + T(n-2) + \Theta(1)$

$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$

exponential term

$T(n) = T(n-1) + T(n-2)$

$\geq 2T(n-2)$ → $n-2k=0$, $n=2k$, $n/2=k$

$= 2^k T(1)$

$= 2^{n/2}$ } exponential! BAD!

Intro to Dynamic Programming

Example: The Fibonacci sequence

$F_1 = 1$, $F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

linear runtime

Memoized DP Algorithm:

Idea: Compute Fibonacci numbers, store in a dictionary.
- when we need a fibonacci number, we first check that dictionary

```
memo = { }          # Dictionary for storing fibonacci's
def fib(n):
    if n is memo:                 # check
        return memo[n]            # Assume Θ(1)
    if n=1 or n=2
        f=1
    else:                         # Θ(n)
        f= fib(n-1) + f(n-2)      # Recursion
    store new fn in memo
```

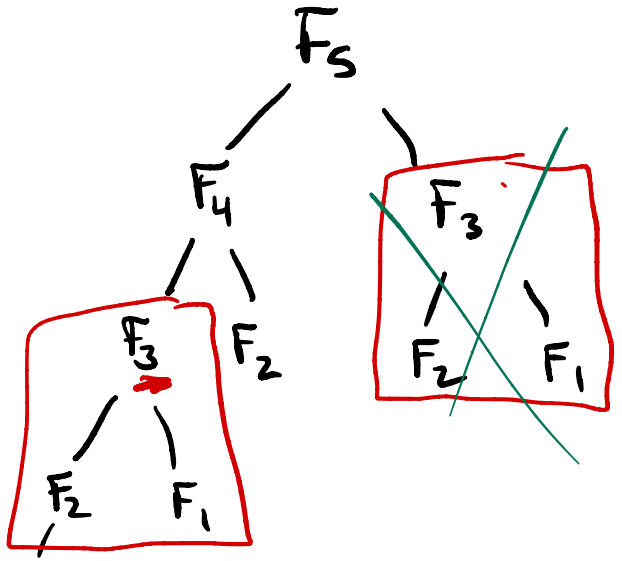
Runtime $\Theta(n)$

What is number of non-memoized calls?

fib(1), fib(2)
fib(3)....
fib(3)

$\Rightarrow n$ times

e.g. what if we wanted to compute F_5 with regular recursion



this is wasteful to compute F_3 over and over.

Intro to Dynamic Programming

Example: The Fibonacci sequence

Tabulation: Bottom up Approach - start with subproblems - build to bigger subproblems

1 1 2 3 5 8 13 → subproblems

```
fib = [ ]
for k in range(1, n+1):
    if k=1 or k=2:
        f=1
    else:
        f = fib[k-1] + fib[k-2]
        fib[k] = f
return fib(n)
```

hash table or array

lookup $\Theta(1)$

Note: This is more or less the same thing as the memorized algo, but with a for loop.

Runtime: $\Theta(n)$

Intro to Dynamic Programming

When should you NOT use Dynamic Programming?

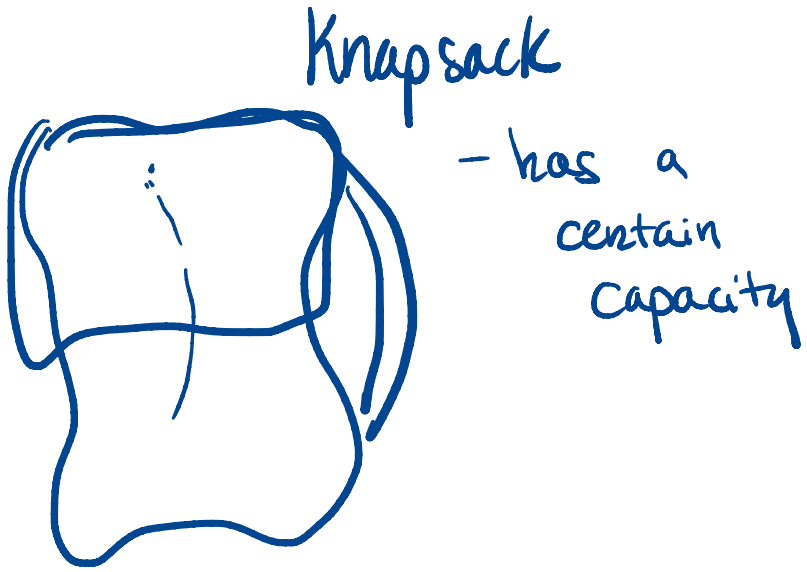
- Not useful for problems that don't have overlapping subproblems with repeat computations.

Binary Search

Intro to Dynamic Programming – Knapsack

Knapsack Problem

Given n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity ω, find the most valuable subset of items that fit into the knapsack.



- We want to find an "optimal" solution.

we want to bring as many items as possible so that the sum of item weights ≤ capacity of knapsack

AND we want to maximize the sum of item values

Intro to Dynamic Programming – Knapsack

Knapsack Problem

Given n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity ω, find the most valuable subset of items that fit into the knapsack.

Example: Given the following items, weights, and values. Make a greedy choice of selecting the items with most value first. Suppose the knapsack capacity is 4. What is the result?

Item	Weight	Value
1	3	1.5
2	2	1
3	2	1

If we choose the most valuable item, $v_1 = 1.5$

$\Rightarrow$ choose Item 1

But Item 1 has $w_1 = 3$ so we can't add Item 2 or Item 3.

Greedy Algo produces solution with value = 1.5

Optimal is to choose Item 2 + Item 3 Value = 2

Intro to Dynamic Programming – Knapsack

Example: Suppose our knapsack has a capacity = 5. The table below shows the items that are available to us to select to put in our knapsack. Use a Dynamic Programming approach to find the optimal solution.

Use tabulation.

Item	Weight	Value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Note: We only have one of each item.

Intro to Dynamic Programming – Knapsack

- for first i items with $1 \leq i \leq n$ with weights $w_1, \ldots, w_i$
  and values $v_1, \ldots, v_i$
  and a knapsack of capacity j, $0 \leq j \leq \omega$

- Let $F(i, j)$ be the value of the optimal solution for this instance.

$F(i, j)$ is a subproblem with i items and capacity j of the larger problem of n items and capacity ω

Assume we have a solution for $F(i-1, j)$. Does adding item i improve the solution?

Idea: Build a table for all $F(i, j)$ solutions

increment j from 0 to ω

i: particular item we might add

Intro to Dynamic Programming – Knapsack

Base Case:

$F(0, j) = 0, \quad j \geq 0$

$F(i, 0) = 0, \quad i \geq 0$

- Build a table for all $F(i, j)$ solutions
- Goal: Find $F(n, \omega)$

tabulation

$F(n, \omega)$

number of items to choose

capacity of Knapsack

Idea: If we include item i in the solution, the capacity for the other items decreases.

capacity for remaining items $= j - w_i$

w_i ← weight of item i

- Entries in table are the Values of items in Knapsack

Intro to Dynamic Programming – Knapsack

If we include item i in the solution, the capacity for $i-1$ items decreases. There could be a different $i-1$ subset selected if we add item i

Recurrence:

$$F(i,j) = \begin{cases} \max(F(i-1,j), v_i + F(i-1, j-w_i)) \\ F(i-1,j) \qquad\qquad if\ j-w_i < 0 \end{cases}$$

Intro to Dynamic Programming – Knapsack

Fill out the row named 1 (Item 1)

Incrementing +1 at a time

Full capacity of knapsack

w_i	v_i		$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
		0	0	0	0	0	0	0
2	12	1	0	F(1,1) 0	F(1,2) $12	F(1,3) $12	F(1,4) $12	F(1,5) $12
1	10	2	0					
3	20	3	0					
2	15	4	0					

can't add items if capacity is 0

intuitively:

Item 1 weighs 2 units

⇒ we can't add it to a knapsack with capacity < 2

⇒ F(1,1) = 0

* we assume we have one of each item

* The numbers in the table represent the value of the items in knapsack

Intro to Dynamic Programming – Knapsack

Fill out Row 2

Recall: Item 2 → $w_2 = 1$, $v_2 = 10$

Item 1: $w_1 = 2$, $v_1 = 12$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	F(2,1) 10	F(2,2) 12	F(2,3) 22	F(2,4) 22	F(2,5) 22
3	0					
4	0					

$$F(i,j) = \max(F(i-1,j),\ v_i + F(i-1, j-w_i))$$

$F(2,1) = \max(F(1,1),\ 10 + F(1,1-1)]$

$= \max(F(1,1),\ 10 + F(1,0)]$

$= \max(0,\ 10 + 0)$

$= 10$

$F(2,2) = \max(F(1,2),\ 10 + F(1,1))$

$= \max(12,\ 10 + 0)$

$= 12$

$F(2,3) = \max(F(1,3),\ 10 + F(1, 3-w_2))$ (current cap.)

$= \max(12,\ 10 + 12)$

Intro to Dynamic Programming – Knapsack

Fill out row 3

$w_3 = 3$, $v_3 = 20$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10				

F(3,3) F(3,4) F(3,5)

$F(3,2) = \max[F(2,2), 20 + F(2, 2-w_3)]$

$= \max(F(2,2), 20 + F(2,-1))$ (undefined)

$= F(2,2)$

$F(3,3) = \max(F(2,3), 20 + F(2, 3-3))$

$= \max(22, 20+0)$

$= 22$

$F(3,4) = \max[F(2,4), 20 + F(2,1)]$

$= \max(22, 30)$

$F(3,5) = \max(F(2,5), 20+12)$

$= \max(22, 32)$

Intro to Dynamic Programming – Knapsack

Item 4: $v_4 = 15$, $w_4 = 2$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

F(4,2) = max(12, 15+0) = 15

max(22, 15+10) = 25

max(30, 15+12) = 30

max(32, 15+22) = 37

So the max possible value is $37

Great. But what items are in knapsack?

Intro to Dynamic Programming – Knapsack

⇒ Item 1, Item 2, Item 4

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
→ 4	0	10	15	25	30	37

$F(4,5) = 37$

$F(4,5) > F(3,5)$

⇒ Item 4 in knapsack

$w_4 = 2$

⇒ Look $F(3,3)$

- Compare $F(3,3) = F(2,3)$
  ⇒ Item 3 not in knapsack.
- compare $F(2,3) > F(1,3)$
  ⇒ Item 2 is included
  $w_2 = 1$
  Look at $F(1,2) > F(0,2)$
  ⇒ Item 1 is included

0/1 Solution: [Item1, Item2, Item3, Item3] = [1, 1, 0, 1]