EECS 376: Foundations of Computer Science

Lecture 04 - Dynamic Programming

EECS ECE CSE

1

Today's Agenda

- Introduction to Dynamic Programming
  - Warm-up: Fibonacci
  - Weighted Task Selection
  - Longest Increasing Subsequence

6/13/24

2

Dynamic Programming

- **Dynamic programming** is both a mathematical optimization method and an algorithmic paradigm.
- The method was developed by Richard Bellman in 1953 and has found applications in numerous fields, from aerospace engineering to economics.

5/14/24

3

Dynamic Programming in Computer Science

- In computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have an *optimal substructure*.
- If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems called the Bellman equation.

5/14/24

4

Recap

- **Previously:** divide and conquer
  - A recurrence – break into smaller sub-problems and combine
  - Design goal is to minimize the number of recursive calls k and time to combine $O(n^d)$
  - Examples: Closest pair, Karatsuba

want to decrease

$$T(n) = k\,T(n/b) + O(n^d \log^w n)$$

want to decrease

6/13/24

6

Dynamic Programming

- **Today:** dynamic programming
  - A recurrence – break into smaller sub-problems and combine
  - ~~Design goal is to minimize the number of recursive calls k and time to combine $O(n^d)$~~
    - Don't worry about minimizing number of recursive calls!
  - **Idea: Maximize number of *repeated* recursive calls**

6/13/24

7

Divide-And-Conquer vs. Dynamic Programming

- Dynamic programming can be seen as an extension of the divide-and-conquer approach.
- While divide-and-conquer works well for problems with independent subproblems, dynamic programming is the strategy of choice for problems with overlapping subproblems, where storing and reusing solutions can lead to more efficient algorithms.

6/13/24

8

Divide-And-Conquer vs. Dynamic Programming

- Both strategies aim to simplify complex problems by breaking them down, but dynamic programming adds the element of memoization to optimize the process when subproblems overlap. *space trade time*
- This makes dynamic programming more efficient for a certain class of problems where the divide-and-conquer approach alone might lead to excessive recomputation of the same subproblems.

6/13/24

9

Warm-Up: Fibonacci

Recurrence for Fibonacci: F(n) = F(n-1) + F(n-2)
F(0) = F(1) = 1

Given a recurrence, three ways to compute its values:

1. **Top-down Recursive:** Starting at desired input, recurse down to base case(s)
2. **Top-down with Memoization:** Same as naïve, but save results as they're computed, reusing already-computed results
3. **Bottom-up Table (aka Dynamic Programming):** Start from base case(s), build up to desired result

13

Method 2: Top-down with Memoization

memo := array indexed from 0 to n

Algorithm Fib(n):
    If n = 0 OR n = 1:
        Return 1
    Else if n ∉ **memo**:
        memo(n) = Fib(n-1) + Fib(n-2)
    Return memo(n)
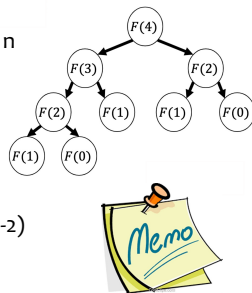
- **Pro:** way faster
- **Con:** not clear how to analyze running time

15

Method 3: Bottom-up Table
(aka Dynamic Programming)

Algorithm Fib(n):
    table := array indexed from 0 to n
    table(0) = 1
    table(1) = 1
        for i = 2 to n:
            table(i) = **table**(i-1) + **table**(i-2)
    Return table(n)

- **Pro:** fast, provides a roadmap for analyzing running time
- **Con:** need to translate from recurrence to table

16

Fib: Naïve Implementation

- The xth Fibonacci number, for x a non-negative integer:
$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$
- **Top-down recursive (naïve):**

$\mathbf{F}(n)$: // $n \geq 0$ an integer
if $n = 0$ or $n = 1$ then return 1
return $\mathbf{F}(n-1) + \mathbf{F}(n-2)$

- **Pro:** direct translation of recurrence
- **Con:** *exponential* runtime:
$$T(n) = T(n-1) + T(n-2) + O(1)$$
$$= O(F(n)) = O(\varphi^n) = O(1.62^n) \text{ using Binet's formula.}$$

Fib: Memoization

- The xth Fibonacci number, for n a non-negative integer:
$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$
- **Top-down memoization:**

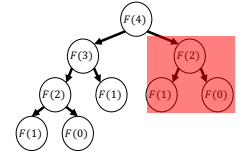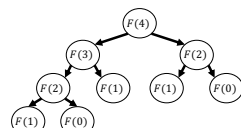*allocate* $F[1..n]$ // entries initially NULL
$F[1] \leftarrow 1, F[2] \leftarrow 1$
M-F(n): // memoized implementation of F_n
if $F[n] =$ NULL then
    $F[n] \leftarrow \mathbf{MF}(n-1) + \mathbf{MF}(n-2)$
return $F[n]$

- **Pros:** much faster (but how much?)
- **Con:** requires accessing global memory, hard to analyze runtime

---

Fib: Memoization – Time and Space Complexity

- Memoization trades space for time. The computation of Fib(n) requires $O(n)$ auxiliary space to store the results for each sub-input.
- On the other hand, since the answer to each sub-input is computed only once, the overall number of operations required (i.e., time complexity) is $O(n)$, a significant improvement over the exponential naïve algorithm.

6/13/24

19

Fib: Bottom up
(dynamic programming)

- Recurrence for Fibonacci:
$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$
- **Bottom-up Table:**

DP-F(x): // table implementation of F_n
allocate $F[1..n]$
$F[0] \leftarrow 1, F[1] \leftarrow 1$
for $i = 2..n$
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

x	F[x]
0	1
1	1
2	2
3	3
4	5
5	8
6	13
7	21

- **Pro:** much faster, no globals, easier to analyze runtime
- **Cons:** must compute *entire* table of smaller results (but usually end up doing this anyway, in every strategy)

Fib: Bottom up – Time and Space Complexity

- Like memoization, the bottom-up approach trades space for time.
- In the case of Fib(n), it too uses $O(n)$ auxiliary space to store the result of each subproblem, and the overall number of additions (i.e., time complexity) required is $O(n)$.
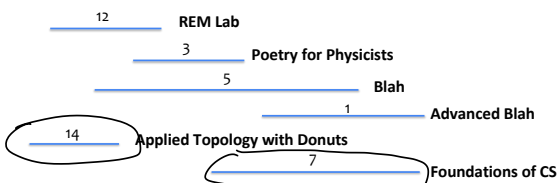
(# recurr)
$O(1.62^n) \rightarrow O(n)$ (dp)

6/13/24

21

Weighted Task Selection

Weighted Course Registration Problem
(aka Weighted Task Selection)

12 — **REM Lab**

3 — **Poetry for Physicists**

5 — **Blah**

1 — **Advanced Blah**

14 — **Applied Topology with Donuts**

7 — **Foundations of CS**

Goal: Choose a set of non-intersecting courses with largest total value.
(there may be many optimal solutions, we just seek one)

*Let the input size be n = #intervals.
6/13/24

Weighted Task Selection Recurrence

Assume the intervals $J_1, J_2, \ldots, J_n$ are given in order of **finish time**.

Let's start from J_n and work backwards.

There are two options:

- OPT has J_n **(use it!)** $\Leftarrow$ $OTV(J_1, \ldots, J_n) = val(J_n) + OTV(J_1, \ldots, J_i)$

  J_i is the last interval that doesn't overlap with J_n

- OPT doesn't have J_n **(lose it!)** $\Leftarrow$ $OTV(J_1, \ldots, J_n) = OTV(J_1, \ldots, J_{n-1})$

  "Optimal Task Value" i.e. the value of the optimal solution

---
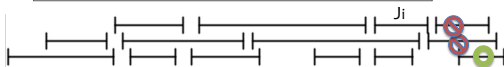
You could write code to implement this recurrence as an algorithm…

Algorithm $OTV(J_1, \ldots, J_n)$:
    if n = 0:
      Return 0
    else:
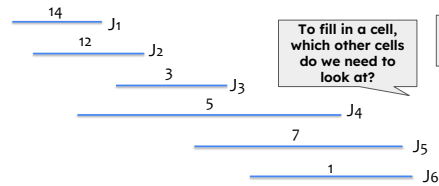   i = index of last interval (before J_n) that doesn't overlap with J_n
        Return max{val(J_n) + $OTV(J_1, \ldots, J_i)$, $OTV(J_1, \ldots, J_{n-1})$}

dp 第一步：建立 state transfer方程 （自上向下）

Dynamic Programming in Action!

For reference: $OTV(J_1, \ldots, J_n) = \max\{val(J_n) + OTV(J_1, \ldots, J_i), OTV(J_1, \ldots, J_{n-1})\}$

OTV(∅)	OTV(J₁)	OTV(J₁,J₂)	OTV(J₁,…,J₃)	OTV(J₁,…,J₄)	OTV(J₁,…,J₅)	OTV(J₁,…,J₆)
0	14 / 0					

14 — J_1

12 — J_2

3 — J_3

5 — J_4

7 — J_5

1 — J_6

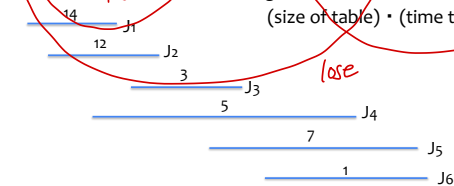> To fill in a cell, which other cells do we need to look at?

> In which order do we fill the table?

Dynamic Programming in Action!

For reference: $OTV(J_1, \ldots, J_n) = \max\{val(J_n) + OTV(J_1, \ldots, J_i), OTV(J_1, \ldots, J_{n-1})\}$

OTV(∅)	OTV(J₁)	OTV(J₁,J₂)	OTV(J₁,…,J₃)	OTV(J₁,…,J₄)	OTV(J₁,…,J₅)	OTV(J₁,…,J₆)
0	0+14 / 0	0+12 / 14	14+3 / 14	0+5 / 17	14+7 / 17	17+1 / 21

use / lose

14 — J_1

12 — J_2

3 — J_3

5 — J_4

7 — J_5

1 — J_6

Running time = (size of table) · (time to fill each entry)

The Final Pseudocode

Algorithm $OTV(J_1, \ldots, J_n)$:
  table := array indexed from 0 to n
  table(0) = 0
    for k = 1 to n:
      i = index of last interval (before J_k) that doesn't overlap with J_k
      // **E.g.** iterate through all intervals to find i
      table(k) = max{val(J_k) + **table**(i), **table**(k-1)}
  Return table(n)

第二步：转为 table 进行 recursion （自下向上）

The DP Recipe

1. Write recurrence ← usually the trickiest part
2. Size of table: How many dimensions? Range of each dimension?
3. What are the base cases?
4. To fill in a cell, which other cells do I look at? In which order do I fill the table?
5. Which cell(s) contain the final answer?
6. Running time = (size of table) · (time to fill each entry)
7. To reconstruct the solution (instead of just its size) follow arrows from final answer to base case

The Final Pseudocode

Algorithm OTV(**J₁, ..., Jₙ**):
   table := array indexed from o to n ⬅ step 2 of DP recipe
   table(o) = o ⬅ step 3 of DP recipe
     for k = 1 to n: ⬅ step 4 of DP recipe
        i = index of last interval (before **Jₖ**) that doesn't overlap with **Jₖ**
        table(k) = max{val(**Jₖ**) + **table**(**i**), **table**(**k-1**)}
   Return table(n) ⬆ step 5 of DP recipe ⬆ steps 1,4 of DP recipe

35

Time Complexity

- A naïve implementation of this algorithm, which does a linear scan inside the loop to determine j, takes $O(n^2)$ time because there are two nested loops each take at most n iterations.
- Since the tasks are sorted by finish time, a more sophisticated implementation would use a binary search, which would take just $O(\log n)$ time for the inner (search) loop, and hence $O(n \log n)$ time overall. (The initial time to sort the tasks by finish time is also $O(n \log n)$.)

6/13/24   36

Longest Increasing Subsequence (LIS)

37

Longest Increasing Subsequence (LIS)

- **Input:** Array S of N numbers

- **Output:** Longest increasing subsequence of S

- Example: What is the LIS of [1,4,3,7,2]?

38

Example

- The sequence S = (0, 8, 7, 12, 5, 10, 4, 14, 3, 6) has several longest increasing subsequences:
  - (0,8,12,14)
  - (0,8,10,14)
  - (0,7,12,14)
  - (0,7,10,14)
  - (0,5,10,14)

6/13/24   39

Dynamic Programming in Action!

- As in the previous problem, we first focus on finding the length of an LIS before concerning ourselves with finding an LIS itself. Let N be the length of S. For our first attempt, we look at the subproblems of computing the length of the LIS for each sequence $S[1..i]$ for $i \in [1,N]$. In the case of S = (0,8,7,12,5,10,4,14,3,6), we can determine these lengths by hand.

1	2	2	3	3	3	3	4	4	4
1	2	3	4	5	6	7	8	9	10

6/13/24   40

Recurrence for the length of the LIS

> It's easy! Check it out!
>
> $$L(A[1..n]) = \begin{cases} L(A[1..n\text{-}1]) + 1 & \text{if } A[n] > A[n\text{-}1] \text{ (use it!)} \\ L(A[1..n\text{-}1]) & \text{otherwise} \quad \text{(lose it!)} \end{cases}$$

Let L denote the length of the LIS
What is wrong?
Give a counter-example:

counter ex: 1 3 2 6 3 4 ⟹ 136
然而 longest 为 1234

41

Let's try to find a suitable recurrence!

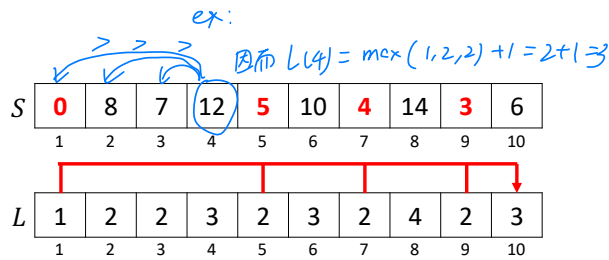- Define $CLIS(S[1..N])$ to be the longest of the increasing subsequences of $S[1..N]$ that contain $S[N]$.
- $CLIS(S[1..N])$ is referred to as a Constrained Longest Increasing Subsequence of a sequence S[1..N].
- Define L(i) to be the length of $CLIS(S[1..i])$
- Then, we have:

$$L(i) = 1 + \max\{L(j) : 0 < j < i \text{ and } S[j] < S[i]\}$$

6/13/24   42

Example

(handwritten: ex: 即 $L(4) = \max(1,2,2)+1 = 2+1 = 3$)

S	0	8	7	12	5	10	4	14	3	6
	1	2	3	4	5	6	7	8	9	10

L	1	2	2	3	2	3	2	4	2	3
	1	2	3	4	5	6	7	8	9	10

The full recurrence

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \text{ or } S[j] \geq S[i] \text{ for all } 0 < j < i \\ 1 + \max\{L(j) : 0 < j < i \text{ and } S[j] < S[i]\} & \text{otherwise.} \end{cases}$$

Once we have $L(i)$ for all $i \in [1, N]$, we just take the maximum value of $L(i)$ as the result for the length of the unconstrained LIS.

Pseudocode

Algorithm LIS(S[1..N]) :
    table := array indexed from 1 to N
      for i = 1 to N:
        if A[i] is the minimum so far: // determined by scanning A[1..i-1]
            table(i) = 1
        Else:
            table(i) = 1+ max{**table**(j)} among all j<i with **A[j]**<**A[i]**
                // find j by scanning **table**
    Return $\max_i$ **table**(i)

What is the runtime? $O(n^2)$

Time and space complexity

- Thus, it takes $O(N)$ time to compute $L(i)$ for a single i, and $O(N^2)$ to compute them all.
- Finding the maximum $L(i)$ takes linear time, as does backtracking.
- So, the algorithm overall takes $O(N^2)$ time, and it uses $O(N)$ space to store the values of $L(i)$.

Wrapping Up

Divide-And-Conquer vs. Dynamic Programming

- **Divide-And-Conquer:**
  - Split the problem into subproblems
  - **Effective when** (1) few recursive calls (2) fast combine step
    - Master Theorem bounds the total runtime

- **Dynamic Programming:**
  - Split the problem into subproblems. Same!
  - **Effective when** there are small number of possible subproblems, which get repeated! So, save your answers.

How to design dynamic programming algorithms

- **Step 1:** (Creative)
  Write a *recursive formulation* of the solution.
  Bound the number of *distinct subproblems* that ever appear in your formulation

- **Step 2:** (Mechanical)
  Create a table representing distinct subproblems. Fill in the table from the **bottom up.**