CMSC 27200 - Guerilla Section 5.

Topics.

1. Overview to dynamic programming
   - Fibonacci numbers ... again.
2. Example: ~~largest~~ contiguous subseq. w/ largest sum.
   - Deriving the recurrence relation.
   - Ordering the subproblems.
   - Writing the pseudocode.
   - Proving correctness - it's free real estate

Dynamic Programming.

We're nearly complete in our tour through algorithm design paradigms.

- Divide and conquer
- Graph algorithms.
- Greedy algorithms.
- Dynamic programming. Today.
- Linear programming | network flow.

What is dynamic programming...

→ Dynamic programming solves ~~an algorithm~~ computational problems by defining smaller subproblems and solving them from smallest first.

What is dynamic programming not?

Dynamic programming is <u>not</u> divide and conquer.

- They might sound the same because both rely on <u>identifying subproblems</u>.
- But they differ in <u>what order</u> the subproblems are solved.
  - ↳ Divide and conquer solves them <u>recursively</u> i.e. <u>top-down</u>
  - ↳ Dynamic programming solves them in a <u>bottom-up fashion</u> starting from the <u>base case</u>.

<u>Example:</u> Fibonacci.

So you might recall that when we started talking about algorithms we ~~went over~~ gave some algorithms to compute the fibonacci #s.

One used ~~recursion~~ divide and conquer. recall.

$$\begin{cases} F_n = F_{n-1} + F_{n-2}. & \forall n \geq 2. \\ F_0 = F_1 = 1 \end{cases}$$

So that this suggested the algorithm.

<u>PROC:</u> Fib

<u>Input:</u> $n \geq 0$ integer.

<u>Do:</u>

1) If $n = 0$ or $n = 1$: output 1.

2) Else output $Fib(n-1) + Fib(n-2)$.

What was the running time of this?

- If $T(n)$ = running time of $Fib(n)$ then

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

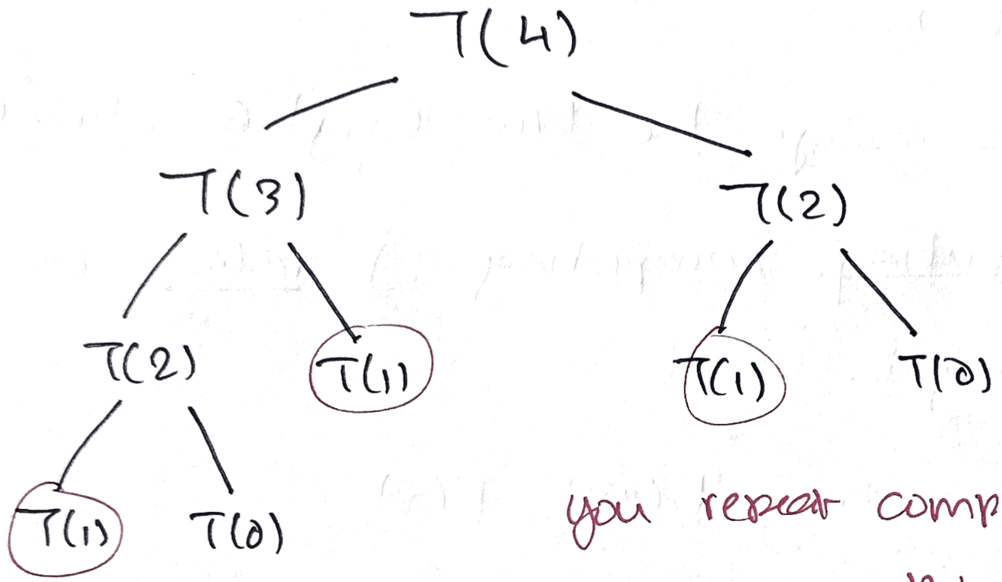
i.e. the n^{th} fibonacci #? Recall

$$T(n) \leq O(2^{n/2}). \longrightarrow \text{exponential time!}$$

Why does it take so long to run?

→ Because we repeat a lot of work!

Think of the recursive tree for $n = 4$.



you repeat computing $T(1)$ multiple times!

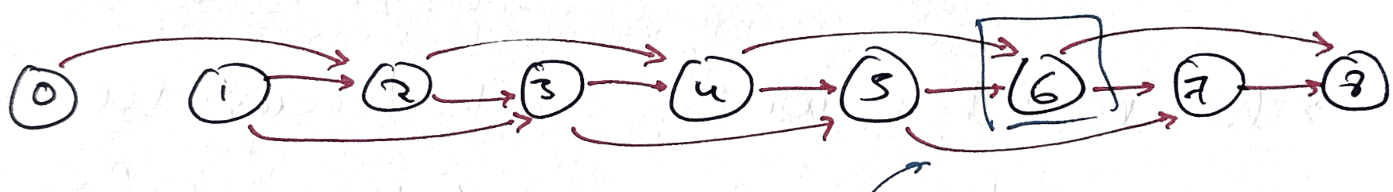
→ We repeat a lot of work because we computed the recurrence relation defining what we wanted to output. In a top-down fashion.

→ That means we should compute $F(n)$ bottom-up. But how?

Let's write out a ~~chain~~ ~~graph corresponding~~ ~~to what we~~ dependency graph

- Let each node represent $F(n)$ for some value of n.
- Let ~~each~~ an edge be b/w (i, j) ~~&~~ directed if ~~computing~~ computing j relies on computing i.

Example: computing ~~F(...)~~ $F(8)$



because $F(6) = F(5) + F(4)$.

Now.. what kind of graph is this?

↳ A DAG!

~~How do I~~ What's the order ~~the~~ nodes ~~so~~ ~~that a node~~ are placed so that they always goes after ~~its~~ their dependencies.
in which

↳ Topological sort!

What does a topological sort correspond to here?

↳ An order to compute F(n) so that I never have to <u>repeat</u> a computation

Give me a <u>simple</u> topological ordering of this DAG...

↳ 0, 1, 2, 3, 4 ...

→ This is to suggest: compute F(n)'s in increasing order of n's so that you <u>never</u> have to <u>repeat</u> a ~~ste~~ case

<u>PROC</u>: FIB DP.

<u>INPUT</u>: $n \geq 0$.

<u>DO</u>:

1) Create an array DP[0...n] of len n+1
2) Initialize DP[0] = DP[1] = 1.
3) For i = 2 ... n:
   - DP[i] = DP[i-1] + DP[i-2]

4) Output DP[n].

→ Questions?

What are the highlights?

- A <u>DP algorithm</u> usually ~~initializes~~ creates a "DP array".
- It initializes the DP array w/ the base case of recurrence relation defining the <u>intended output</u>.
- It iterates through entries of the DP array to fill in the table based on the <u>dependency graph</u>. → So that <u>no</u> ~~the~~ <u>work</u> is repeated

  ↳ The entry is filled in based on the <u>recursive case</u> of the recurrence relation.

— Output is "usually" an entry of the DP table.

→ Questions?

⟹ The DP blueprint.

So how do you know when to use DP?

↳ You can clearly define your output using a recurrence relation.

↳ Computing the recurrence relation benefits in iterating in a bottom-up fashion.

How do you apply DP?

1. Identify your intended output and write a recurrence relation which computes it.

2. Draw a dependency graph to ~~show~~ see how to compute your recurrence relation

in a bottom-up fashion.

3. Write the algorithm which computes the recurrence relation in the correct order

Now let's see this plan in action.

Using DP.

So what problems have we encountered use DP?

- Fibonacci.
- All-pairs shortest path. (Bellman-Ford).
- Edit distance
- Longest path in a DAG.
- Knapsack.

These are problems that you should associate w/ DP. The textbook has a few others

- Traveling Salesman.
- Independent set in trees.

Today we cover another. Given an array of #s. we want a contiguous subsequence w/ the largest sum.

↳ A subsequence is contiguous if it is made of consecutive elems. in the sequence

$5 \quad 15 \quad \underbrace{-30 \quad 10 \quad -5 \quad 40}_{\text{Contiguous}} \quad 10.$

INPUT: A list of #s $A = [a_1 \ldots a_n]$

OUTPUT: The contiguous subsequence w/ largest sum

↳ A length zero subseq. also has sum 0.

→ Question?

⟹ Step #1: Defining a recurrence relation.

So here's a way to break it down.

For problems you typically apply DP to, you're usually doing some optimization task. i.e. extremizing an "objective value" over a set of "feasible solns"

Goal: find the contiguous subsequence w/ maximum sum.

feasible soln

objective.

↳ Feasible Solns: contiguous subseqs.

↳ Objective: its sum

The recurrence relation typically computes the best objective over a subset of solns describable by a simple ~~vector~~ parameter.

HAZARD: when defining the relation you have to make sure ranging over all params hits all feasible solns.

How do we guess this function?

↳ A bit of an art... but this is one heuristic that works.

① Pick a way to parameterize all feasible solns.

↳ All contiguous subsequences can be described by where they start and where they end. Call these indices $(i, j) \in [n] \times [n]$.

② Write the objective fn under this parameterization

Objective = max sum over any contiguous subsequence

max over all endings j

$$= \max_{j=1..n} \max_{i=1..j} \underbrace{\sum_{\ell=i}^{j} A[\ell]}$$

max over all starts before j.

Sum of subseq from $i \ldots j$.

3. Pick the "<u>inner</u>" <u>part</u> to be what you want ~~to define recursively~~. your <u>recurrence relation</u> to define.

$$\max_{j=1..n} \underbrace{\max_{i=1..j} \sum_{\ell=i}^{j} A[\ell]}_{}$$

let this be $f(j)$.

that is $f(j)$ = "max contiguous subseq sum ending at j".

And now we need to double check.

↳ if we compute all values of $f(j)$ i.e. $\forall$ $j=1...n$. can we output what we want?

↳ Yeah kinda immediately because how we wrote the objective

↳ Given $f(j)$ for all $j=1...n$. loop over all j and output the max.

Also note: we get the invariant that we'll want to prove in order to establish our algorithm is correct:

Claim: For all $j = 1 \ldots n$, $f(j)$ computes the maximum contiguous subseq. sum for a subseq ending at j.

→ And now we want to define $f(j)$ ~~recursively~~ via a recurrence relation.

1. Base case: what are immediate values we can fill in.

$$f(1) = \max \{0, A[1]\}.$$

o.w. it's $\emptyset$.

if $A[1] > 0$ the max subseq is $\{A[1]\}$.

Inductive case: Suppose we have $f(j)$ computed for all $j = 1 \dots n$. How do we compute $f(n+1)$?

notice that's strong ind.

Compare side by side

$$f(n+1) = \max_{i=1\dots n, n+1} \text{Sum of } A[i \dots n+1]$$

$$f(n) = \max_{i=1\dots n} \text{Sum of } A[i \dots n].$$

notice what you need to compute in $n+1$ overlaps w/ n.

Let's make that more explicit.

$$f(n+1) = \max_{i=1\dots n, n+1} \sum_{l=i}^{n+1} A[l]$$

$$= \max_{i=1\dots n, n+1} \left(\sum_{l=i}^{n} A[l] \right) + A[n+1]$$

$$= \max \left\{ \underbrace{\max_{i=1\dots n} \sum_{l=i}^{n} A[l]}_{f(n)\,!} + A[n+1] \,,\, \underbrace{A[n+1]}_{\text{for when } i=n+1} \right\}$$

That is our recursive case

$$f(j+1) = \max \{ f(j) + A[j+1],\ A[j+1] \}.$$

So to put it all together

$$f(j) = \begin{cases} \max \{0, A[1]\} & \text{if } j=1 \\ \max \{ f(j) + A[j+1], A[j+1] \} & \text{o.w.} \end{cases}$$

⟶ Notice in deriving the recurrence relation we're basically proven the correctness of the algorithm ... That's by design!

To recap: the first step to applying DP is to write a recurrence relation. This had two parts.

1) Make a guess for what the recurrence relation computes ⟶ gives the IH.
2) Actually define it according to the IH

→ **Step #2: The dependency graph.**

Now to draw the dependency graph. Look at the recursive case

$$\underbrace{f(j+1)} = \max\{\underbrace{f(j)} + A[j+1],\ A[j+1]\}$$

node reprs this.

Only one edge! $(j, j+1)$.

$$f(1) \rightarrow f(2) \rightarrow \cdots \rightarrow f(n)$$

You can iterate through $j = 1 \ldots n$ and never have to recompute a case.

→ **Step #3: Write down the algorithm.**

Proc: Max SubSeq.

Input: List $A = \{a_1 \ldots a_n\}$.

Do:

1. Initialize $DP[1 \ldots n]$.

2. ~~for~~ $DP[1] = \max\{0, A[1]\}$.

3. for $j = 2 \ldots n$: do

   – Set $DP[j] = \max\{DP[j-1] + A[j], A[j]\}$

4. Loop through $j = 1 \ldots n$. output max $DP[j]$

Are we done? ... no! We didn't actually output the subsequence

↳ Easier to first compute ~~solve for~~ the value of the problem

↳ Then find the soln. because alg for the soln usually looks the same!

1. Init $DP[1 \ldots n]$ and $DP\ Index[1 \ldots n]$

2. Set $DP[1] = \max\{0, A[1]\}$.

3. If $A[1] > 0$: set $DP\ Index[1] = 1$ o.w. $DP\ Index[1] = 2$.

4. For $j = 2 \ldots n$, do.

Let DP Index denote the start index of the max contiguous subseq ending at j.

134

- Set $DP[j] = \max\{\underbrace{DP[j-1] + A[j]}, A[j]\}$

In this case
$DPIndex[j] = DPIndex[j-1]$

o.w. $DPIndex[j] = j$

- If $A[j] > DP[j-1] + A[j]$
  - Set $DPIndex[j] = j$
- Else $DPIndex[j] = DPIndex[j-1]$..

→ Questions?

⇒ Proof of correctness

To run a proof of correctness ... literally run our derivation of the recurrence relation in reverse.

Lemma: Given any list $A[1 \ldots n]$ of n elem

$\forall j = 1 \ldots n$, $f(j)$ computes.

should have set $f(0) = 0 \ldots$

$$f(j) = \max\left\{ \max_{i=1 \ldots j} \sum_{\ell=i}^{j} A[\ell], 0 \right\}.$$

Pf: ~~As a~~ proceed via induction on j.

Base case $j=1$: ~~In case~~ either $A[1] > 0$ or not. If $A[1] > 0$ then $f(1) = A[1]$ o.w. $f(1) = 0$. $\longrightarrow$ $f(1) = \max\{0, A[1]\}$.

Inductive case: Suppose $f(j)$ satisfies

$$f(j) = \max_{i=1 \ldots j} \sum_{\ell=i}^{j} A[\ell]$$

then by defn.

$$f(j+1) = \max\{f(j) + A[j+1], A[j+1]\}$$

136

$$= \max_{\cancel{j=1..j}} \left\{ \max_{i=1..j} \sum_{l=i}^{j} A[l] + A[j+1],\ A[j+1] \right\}$$

$$= \max \left\{ \max_{i=1..j} \sum_{l=i}^{j+1} A[l],\ A[j+1] \right\}$$

$$= \max_{i=1..j+1} \left\{ \sum_{l=i}^{j+1} A[l] \right. \qquad \square$$

∴ alg outputs correct value...

$\Rightarrow$ Running Time: $O(n)$.