Dynamic Programming

Kevin Zatloukal

July 18, 2011

Motivation

Dynamic programming deserves special attention:

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice
  - the (few) novel algorithms I've invented used it

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice
  - the (few) novel algorithms I've invented used it
  - dynamic programming algorithms are ubiquitous in CS

Motivation

applications of dynamic programming in CS

compilers	parsing general context-free grammar, optimal code generation
machine learning	speech recognition
databases	query optimization
graphics	optimal polygon triangulation
networks	routing
applications	spell checking, file diffing, document layout, regular expression matching

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice
  - the (few) novel algorithms I've invented used it
  - dynamic programming algorithms are ubiquitous in CS
- more robust than greedy to changes in problem definition

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice
  - the (few) novel algorithms I've invented used it
  - dynamic programming algorithms are ubiquitous in CS
- more robust than greedy to changes in problem definition
- actually simpler than greedy

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice
  - the (few) novel algorithms I've invented used it
  - dynamic programming algorithms are ubiquitous in CS
- more robust than greedy to changes in problem definition
- actually simpler than greedy
  - (usually) easy correctness proofs and implementation

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice
  - the (few) novel algorithms I've invented used it
  - dynamic programming algorithms are ubiquitous in CS
- more robust than greedy to changes in problem definition
- actually simpler than greedy
  - (usually) easy correctness proofs and implementation
  - easy to optimize

Motivation

Dynamic programming deserves special attention:

- technique you are most likely to use in practice
  - the (few) novel algorithms I've invented used it
  - dynamic programming algorithms are ubiquitous in CS
- more robust than greedy to changes in problem definition
- actually simpler than greedy
  - (usually) easy correctness proofs and implementation
  - easy to optimize

In short, it's simpler, more general, and more often useful.

What is Dynamic Programming?

Key is to relate the solution of the whole problem and the solutions of subproblems.

- a subproblem is a problem of the same type but smaller size
- e.g., solution for whole tree to solutions on each subtree

Same is true of divide & conquer, but here the subproblems need not be disjoint.

- they need not divide the input (i.e., they can "overlap")
- divide & conquer is a special case of dynamic programming

A dynamic programming algorithm computes the solution of *every subproblem* needed to build up the solution for the whole problem.

- compute each solution using the above relation
- store all the solutions in an array (or matrix)
- algorithm simply fills in the array entries in some order

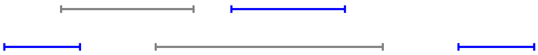
Example 1: Weighted Interval Scheduling

Recall Interval Scheduling

In the Interval Scheduling problem, we were given a set of intervals $I = \{(s_i, f_i) \mid i = 1, \ldots n\}$, with start and finish times s_i and f_i.

Our goal was to find a subset $J \subset I$ such that

- no two intervals in J overlap and
- $|J|$ is as large as possible



Greedy worked by picking the remaining interval that finishes first.

- This gives the blue intervals in the example.

Example 1: Weighted Interval Scheduling

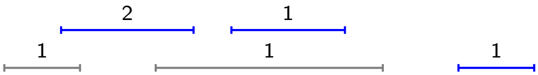
Problem Definition

In the Weighted Interval Scheduling problem, we are given the set I along with a set of weights $\{w_i\}$.

Now, we wish to find the subset $J \subset I$ such that

- no two intervals in J overlap and
- $\sum_{i \in J} w_i$ is as large as possible

For example, if we add weights to our picture, we get a new solution shown in blue.



Example 1: Weighted Interval Scheduling

Don't Be Greedy

As this example shows, the greedy algorithm no longer works.

- greedy throws away intervals regardless of their weights

Furthermore, no simple variation seems to fix this.

- we know of no greedy algorithm for solving this problem

As we will now see, this can be solved by dynamic programming.

- dynamic programming is more general
- we will see another example of this later on

Example 1: Weighted Interval Scheduling

Relation

Let $\text{OPT}(I')$ denote the value of the optimal solution of the problem with intervals chosen from $I' \subset I$.

Consider removing the last interval $\ell_n = (s_n, f_n) \in I$.

- How does $\text{OPT}(I)$ relate to $\text{OPT}(I - \{\ell_n\})$?
- $\text{OPT}(I - \{\ell_n\})$ is the value of the optimal solution that does not use ℓ_n.

Example 1: Weighted Interval Scheduling

Relation

Let $\mathrm{OPT}(I')$ denote the value of the optimal solution of the problem with intervals chosen from $I' \subset I$.

Consider removing the last interval $\ell_n = (s_n, f_n) \in I$.

- How does $\mathrm{OPT}(I)$ relate to $\mathrm{OPT}(I - \{\ell_n\})$?
- $\mathrm{OPT}(I - \{\ell_n\})$ is the value of the optimal solution that does not use ℓ_n.
- What is the value of the optimal solution that does use ℓ_n?
- It must be $w_n + \mathrm{OPT}(I - \mathrm{conflicts}(\ell_n))$, where $\mathrm{conflicts}(\ell_n)$ is the set of intervals overlapping ℓ_n. (Why?)

Example 1: Weighted Interval Scheduling

Relation

Let $\mathrm{OPT}(I')$ denote the value of the optimal solution of the problem with intervals chosen from $I' \subset I$.

Consider removing the last interval $\ell_n = (s_n, f_n) \in I$.

- How does $\mathrm{OPT}(I)$ relate to $\mathrm{OPT}(I - \{\ell_n\})$?
- $\mathrm{OPT}(I - \{\ell_n\})$ is the value of the optimal solution that does not use ℓ_n.
- What is the value of the optimal solution that does use ℓ_n?
- It must be $w_n + \mathrm{OPT}(I - \mathrm{conflicts}(\ell_n))$, where $\mathrm{conflicts}(\ell_n)$ is the set of intervals overlapping ℓ_n.
- Hence, we must have:

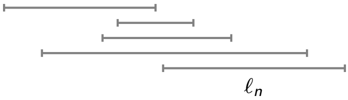
$$\mathrm{OPT}(I) = \max\{\mathrm{OPT}(I - \{\ell_n\}), w_n + \mathrm{OPT}(I - \mathrm{conflicts}(\ell_n))\}.$$

Example 1: Weighted Interval Scheduling

Relation (cont.)

We can simplify this by looking at conflicts(ℓ_n) in more detail:

- conflicts(ℓ_n) is the set of finishing after ℓ_n starts.
- If we sort I by finish time, then these are a suffix.



Let $p(s_n)$ denote the index of the first interval finishing after s_n.

- conflicts$(\ell_n) = \{\ell_{p(s_n)}, \ldots, \ell_n\}$
- $I - \{\ell_n\} = \{\ell_1, \ldots, \ell_{n-1}\}$
- $I -$ conflicts$(\ell_n) = \{\ell_1, \ldots, \ell_{p(s_n)-1}\}$

Let $\mathrm{OPT}(k) = \mathrm{OPT}(\{\ell_1, \ldots, \ell_k\})$. Then we have

$$\mathrm{OPT}(n) = \max\{\mathrm{OPT}(n-1), w_n + \mathrm{OPT}(p(s_n) - 1)\}.$$

Example 1: Weighted Interval Scheduling

Pseudocode

Store the values of OPT in an array *opt-val*.

- start out with $\text{OPT}(0) = 0$
- fill in rest of the array using the relation

SCHEDULE-WEIGHTED-INTERVALS(*start*, *finish*, *weight*, *n*)

sort *start*, *finish*, *weight* **by** *finish*
opt-val $\leftarrow$ NEW-ARRAY()
opt-val[0] $\leftarrow 0$
for $i \leftarrow 1$ **to** n
do $j \leftarrow$ BINARY-SEARCH(*start*[*i*], *finish*, *n*)
  opt-val[*i*] $\leftarrow \max\{\textit{opt-val}[i-1], \textit{weight}[i] + \textit{opt-val}[j-1]\}$
return *opt-val*[*n*]

Running time is clearly $O(n \log n)$.

Example 1: Weighted Interval Scheduling

Observations

- This is efficient primarily because of the special structure of conflicts(ℓ_n). (Depends on ordering the intervals.)

  If we had to compute OPT(J) for every $J \subset I$, the algorithm would run in $\Omega(2^n)$ time.
- This is still mostly a brute-force search. We excluded only solutions that are suboptimal on subproblems.
- Dynamic programming always works, but it is not always efficient. (Textbooks calls it "dynamic programming" only when it is efficient.)
- It is hopefully intuitive that dynamic programming often gives efficient algorithms when greedy does not work.

Example 1: Weighted Interval Scheduling

Finding the Solution (Not Just Its Value)

Often we want the actual solution, not just its value.

The simplest idea would be to create another array opt-set, such that opt-set[k] stores the set of intervals with weight opt-val[k].

- each set might be $\Theta(n)$ size
- so the algorithm might now be $\Theta(n^2)$

Instead, we can just record enough information to figure out whether each ℓ_n was in the optimal solution or not.

- but this is in the opt-val array already
- ℓ_i is included iff $\mathrm{OPT}(i) = w_i + \mathrm{OPT}(p(i) - 1)$ or equivalently iff $\mathrm{OPT}(i) > \mathrm{OPT}(i - 1)$

Example 1: Weighted Interval Scheduling

Finding the Solution (Not Just Its Value) (cont)

```
OPTIMAL-WEIGHTED-INTERVALS(opt-val, n)
    opt-set ← ∅
    i ← n
    while i > 0
     do if opt-val[i] > opt-val[i − 1]
           then opt-set ← opt-set ∪ {i}
                i ← BINARY-SEARCH(start[i], finish, n) − 1
           else i ← i − 1
    return opt-set
```

This approach can be used for any dynamic programming algorithm.

Example 2: Maximum Subarray Sum

Problem Definition

In this problem, we are given an array A of n numbers.

Our goal is to find the subarray $A[i \ldots j]$ whose sum is as large as possible.

For example, in the array below, the subarray with largest sum is shaded blue.

1	-2	7	5	6	-5	5	8	1	-6

Example 2: Maximum Subarray Sum

Problem History

- Problem was first published in Bentley's *Programming Pearls*.
- It was originally described to him by a statistician trying to fit models of a certain type. (See example in the textbook.)
- It has several different solutions with a range of efficiencies:
  - $O(n^3)$ brute-force
  - $O(n^2)$ optimized brute-force
  - $O(n \log n)$ divide & conquer
  - $O(n)$ clever insight
- Became a popular interview question (e.g., at Microsoft).
- However, the clever solution can also be produced by applying dynamic programming.

Example 2: Maximum Subarray Sum

Relation

As before, consider whether $A[n]$ is in the optimal solution:

- if not, then $\text{OPT}(1 \dots n) = \text{OPT}(1 \dots n-1)$
- if so, then the optimal solution is $A[i \dots n]$ for some i
- but $A[i \dots n-1]$ need not be $\text{OPT}(1 \dots n-1)$ (Why?)

Example 2: Maximum Subarray Sum

Relation

As before, consider whether $A[n]$ is in the optimal solution:

- if not, then $\mathrm{OPT}(1 \dots n) = \mathrm{OPT}(1 \dots n-1)$
- if so, then the optimal solution is $A[i \dots n]$ for some i
- but $A[i \dots n-1]$ need not be $\mathrm{OPT}(1 \dots n-1)$

1	-1	1	-4	5

In this example, $\mathrm{OPT}(1 \dots 5)$, shown in red, is achieved at $A[3 \dots 5]$, which includes $A[5]$. However, $\mathrm{OPT}(1 \dots 4)$, shown in blue (checkered), is achieved at $A[1 \dots 3]$.

The value $\mathrm{OPT}(1 \dots n-1) + A[n]$ would be sum of $A[1 \dots 3] \cup A[5]$, which is not actually a subarray.

What we do know is that $A[i \dots n-1]$ is the optimal solution *ending at* $A[n-1]$.

Example 2: Maximum Subarray Sum

Relation (cont)

Let's instead focus on computing $\mathrm{OPT}'(n)$: the optimal sum of a subarray ending at $A[n]$.

Consider whether $A[n]$ is in the optimal solution:

- if so, then $\mathrm{OPT}'(n) = \mathrm{OPT}'(n-1) + A[n]$
- if not, then $\mathrm{OPT}'(n) = 0$ (sum of the empty array)

Thus, we have the relation

$$\mathrm{OPT}'(n) = \max\{\mathrm{OPT}'(n-1) + A[n], 0\}.$$

Example 2: Maximum Subarray Sum

Relation (cont more)

Repeating our argument from before, we can see that

$$\mathrm{OPT}(n) = \max\{\mathrm{OPT}(n-1), \mathrm{OPT}'(n)\}.$$

If the optimal solution does not include $A[n]$, then $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$. And if it does include $A[n]$, then it must be the optimal subarray ending at n, i.e., $\mathrm{OPT}'(n)$.

Note that we can simplify this to just

$$\mathrm{OPT}(n) = \max\left\{\mathrm{OPT}'(j) \mid j = 1, \ldots n\right\}.$$

In retrospect, this should have been obvious: $\mathrm{OPT}(n)$ is simply the maximum value of $\mathrm{OPT}'(j)$ since the optimal subarray ends at *some* index j.

Example 2: Maximum Subarray Sum

Pseudocode

MAX-SUBARRAY-SUM(A, n)

$opt \leftarrow 0, opt' \leftarrow 0$
for $i \leftarrow 1$ **to** n
do $opt' \leftarrow \max\{0, opt' + A[i]\}$
    $opt \leftarrow \max\{opt, opt'\}$
return opt

Here, we have performed a further optimization:

- since we only need $\mathrm{OPT}(n-1)$ (not all earlier values), we can just keep a single variable
- this is typical of the sort of optimization that can be performed on dynamic programming algorithms: removing wasted space / work

This final solution looks clever. However, it came from the standard dynamic programming approach and simple optimizations.

Etimology of Dynamic Programming

Where does the term "programming" mean?

- a program is something you might get at a concert
- a "program" is like a "schedule" but more general
  - includes both what to do and when to do it
- "programming" is like "scheduling"
  - coming up with a program

What does the term "dynamic" mean?

- means "relating to time"
- Bellman was studying multi-stage decision processes
- decide what to do in step 1, then in step 2, etc.
- steps need not really be "time"

History of Dynamic Programming

Invented by Richard Bellman in the 1950s.

In his book *Dynamic Programming*, Bellman described the origin of the name as above.

But in his autobiography, Bellman admitted other reasons:

- Secretary of Defense (Wilson) did not like math research
- Bellman wanted a name that didn't sound like math
- "it's impossible to use the word 'dynamic' in a pejorative sense"
- "it was [a name] not even a Congressman could object to"

Examples in 2 Dimensions

We defined dynamic programming to be solving a problem by using solutions of subproblems of smaller "size".

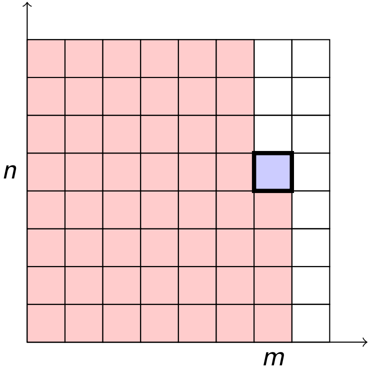
In the first two examples, the size was n. So smaller means $i < n$.

However, we can generalize further:

- Other examples will have two measures of size, n and m.
- Now there are multiple ways to define "smaller".
- E.g., we can say (n', m') is smaller than (n, m) if $n' < n$ or $n' = n$ and $m' < m$.

In principle, we can generalize dynamic programming to any number of size dimensions. In practice, more than 2 is very rare.

Examples in 2 Dimensions (cont)



In this picture, all the red squares are smaller than the blue one.

We can fill in this matrix from bottom-to-up, then left-to-right.

Example 3: Knapsack

Problem Definition

We are given a set of n items, each with weight v_i and value w_i, along with a weight limit W.

Our goal is to find a subset of items $S \subset [n]$ that:

- fits in the sack: $\sum_{i \in S} w_i \leq W$
- has $\sum_{i \in S} v_i$ as large as possible

This problem comes up often.

- I've implemented the next algorithm at least 3 times in various settings

Example 3: Knapsack

Relation

Consider the last item, n:

- if n is not in the optimal solution, then we have $\mathrm{OPT}([n]) = \mathrm{OPT}([n-1])$
- but if n is in the optimal solution, then the rest of the optimal solution need not be $\mathrm{OPT}([n-1])$ (why not?)

Example 3: Knapsack

Relation

Consider the last item, n:

- if n is not in the optimal solution, then we have $\mathrm{OPT}([n]) = \mathrm{OPT}([n-1])$
- but if n is in the optimal solution, then the rest of the optimal solution need not be $\mathrm{OPT}([n-1])$
- what we need is the optimal solution over $[n-1]$ with total weight at most $W - w_n$

Let $\mathrm{OPT}(k, V)$ be the value of the optimal solution over items $[k]$ with total weight at most V. Then we have

$$\mathrm{OPT}(n, W) = \max\{\underbrace{\mathrm{OPT}(n-1, W-w_n) + v_n}_{n \text{ included}}, \underbrace{\mathrm{OPT}(n-1, W)}_{\text{not included}}\}$$

Example 3: Knapsack

Pseudocode

```
KNAPSACK(w, v, n, W)
opt ← NEW-MATRIX()
for V ← 1 to W
do opt[0, V] ← 0
for k ← 1 to n
do for V ← 1 to W
      do opt[k, V] ← max{opt[k − 1, V − w[k]] + w[k],
                         opt[k − 1, V]}
return opt[n, W]
```

This algorithm can be optimized:

- we don't need the whole matrix (how much do we need?)

Example 3: Knapsack

Pseudocode

```
KNAPSACK(w, v, n, W)
opt ← NEW-MATRIX()
for V ← 1 to W
do opt[0, V] ← 0
for k ← 1 to n
do for V ← 1 to W
     do opt[k, V] ← max{opt[k − 1, V − w[k]] + w[k],
                        opt[k − 1, V]}
return opt[n, W]
```

This algorithm can be optimized:

- we don't need the whole matrix
- can get away with two just two columns

As before, easy to compute solution as well. (how?)

Example 3: Knapsack

Pseudocode (cont)

Easy to see that this runs in $O(nW)$ time.

- is that actually efficient?

Example 3: Knapsack

Pseudocode (cont)

Easy to see that this runs in $O(nW)$ time.

- is that actually efficient?
- only if W is small
- this is often the case in practice

Knapsack problem is actually NP-hard for general W.

Algorithms like the one we just saw (where the running time depends on an input) are called *pseudo-polynomial time*.

Intermission

Welcome Back

To start off, some quick review:

- weighted interval scheduling

Afterward, we will look at some more sophisticated examples.

Finally, we will summarize the key points from the examples.

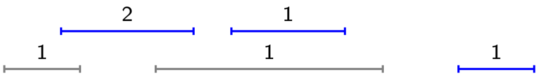
Review: Weighted Interval Scheduling

Problem Definition

In the Weighted Interval Scheduling problem, we are given the set I along with a set of weights $\{w_i\}$.

Now, we wish to find the subset $J \subset I$ such that

- no two intervals in J overlap and
- $\sum_{i \in J} w_i$ is as large as possible



Review: Weighted Interval Scheduling

Relation

We sorted the intervals by finish time: $\ell_1, \ell_2, \ldots, \ell_n$. Then the intervals conflicting with ℓ_n are $\ell_{p(n)}, \ldots, \ell_n$ for some $p(n)$.

(Specifically, $p(n)$ is the index of the first interval finishing after ℓ_n starts.)

Define $\mathrm{OPT}(i)$ to be the optimum solution value over $\ell_1, \ldots, \ell_i$. We argued before that:

$$\mathrm{OPT}(i) = \max\{\underbrace{\mathrm{OPT}(p(i)-1) + w_i}_{i \text{ included}}, \underbrace{\mathrm{OPT}(i-1)}_{\text{not included}}\}$$

Two options to consider: ℓ_i is included or not.

- if not, then solution value is $\mathrm{OPT}(i-1)$ by definition
- if it is included, ...

Review: Weighted Interval Scheduling

Relation (cont)

Let the optimal solution be $J \subset \{\ell_1, \ldots, \ell_i\}$ with $\ell_i \in J$.

Then $J = J_0 \cup \{\ell_i\}$ with $J_0 \subset \{\ell_1, \ldots, \ell_{p(i)-1}\}$. (Why?)

Review: Weighted Interval Scheduling

Relation (cont)

Let the optimal solution be $J \subset \{\ell_1, \ldots, \ell_i\}$ with $\ell_i \in J$.

Then $J = J_0 \cup \{\ell_i\}$ with $J_0 \subset \{\ell_1, \ldots, \ell_{p(i)-1}\}$.

Claim: $\sum_{j \in J_0} w_j = \mathrm{OPT}(p(i) - 1)$ (i.e, J_0 is optimal over $\ell_1, \ldots, \ell_{p(i)-1}$)

Suppose $\exists$ valid $J_0' \subset \{\ell_1, \ldots, \ell_{p(i)-1}\}$ with $\sum_{j \in J_0'} w_j > \sum_{j \in J_0} w_j$.
Then let $J' = J_0' \cup \{\ell_i\}$. J' is a valid solution over $\ell_1, \ldots, \ell_i$
(Why?)

Review: Weighted Interval Scheduling

Relation (cont)

Let the optimal solution be $J \subset \{\ell_1, \ldots, \ell_i\}$ with $\ell_i \in J$.

Then $J = J_0 \cup \{\ell_i\}$ with $J_0 \subset \{\ell_1, \ldots, \ell_{p(i)-1}\}$.

Claim: $\sum_{j \in J_0} w_j = \mathrm{OPT}(p(i) - 1)$ (i.e, J_0 is optimal over $\ell_1, \ldots, \ell_{p(i)-1}$)

Suppose $\exists$ valid $J_0' \subset \{\ell_1, \ldots, \ell_{p(i)-1}\}$ with $\sum_{j \in J_0'} w_j > \sum_{j \in J_0} w_j$.
Then let $J' = J_0' \cup \{\ell_i\}$. J' is a valid solution over $\ell_1, \ldots, \ell_i$ and
$\sum_{j \in J'} w_j = w_i + \sum_{j \in J_0'} w_j > w_i + \sum_{j \in J_0} w_j = \sum_{j \in J} w_j$.

This is the basic fact that allows us to use the solution to a subproblem to solve the whole problem. We will use it repeatedly.

Review: Weighted Interval Scheduling

Pseudcode

Algorithm: compute all the values of OPT in an array.

- start out with $\text{OPT}(0) = 0$
- fill in rest of the array using the relation

$\textsc{Schedule-Weighted-Intervals}(\mathit{start}, \mathit{finish}, \mathit{weight}, n)$

sort *start*, *finish*, *weight* **by** *finish*
$\mathit{opt\text{-}val} \leftarrow \textsc{New-Array}()$
$\mathit{opt\text{-}val}[0] \leftarrow 0$
for $i \leftarrow 1$ **to** n
do $j \leftarrow \textsc{Binary-Search}(\mathit{start}[i], \mathit{finish}, n)$
    $\mathit{opt\text{-}val}[i] \leftarrow \max\{\mathit{opt\text{-}val}[i-1], \mathit{weight}[i] + \mathit{opt\text{-}val}[j-1]\}$
return $\mathit{opt\text{-}val}[n]$

Review: Weighted Interval Scheduling

Finding the Solution (Not Just Its Value)

The relation for OPT contains enough information to find the optimal solution, not just its value.

Let $\mathcal{S}(i)$ be the solution with value $\mathrm{OPT}(i)$.

Then we have two cases for which option maximized $\mathrm{OPT}(i)$:

- if $\mathrm{OPT}(i) = \mathrm{OPT}(i-1)$, then $\mathcal{S}(i) = \mathcal{S}(i-1)$
- otherwise, $\mathcal{S}(i) = \{i\} \cup \mathcal{S}(p(i))$

Review: Weighted Interval Scheduling

Finding the Solution (Not Just Its Value) (cont)

Optimal-Weighted-Intervals(*opt-val*, *n*)

```
opt-set ← ∅
i ← n
while i > 0
 do if opt-val[i] > opt-val[i − 1]
         then opt-set ← opt-set ∪ {i}
                 i ← Binary-Search(start[i], finish, n) − 1
         else  i ← i − 1
return opt-set
```

This approach can be used for any dynamic programming algorithm.

Review: Examples in 2 Dimensions

We defined dynamic programming to be solving a problem by using solutions of subproblems of smaller "size".

In the first two examples, the size was n. So smaller means $i < n$.

However, we can generalize further:

- Other examples will have two measures of size, n and m.
- Now there are multiple ways to define "smaller".
- E.g., we can say (n', m') is smaller than (n, m) if $n' < n$ or $n' = n$ and $m' < m$.

Last time, we looked at the knapsack problem, whose size measures were n and W.

Example 4: String Search With Wildcards

Problem Definition

We are given a string s and a pattern p. In addition to letters, p may contain wildcards:

- '?' matches any single character
- '*' matches any sequence of one or more characters

Our goal is to find the first, longest match $s[i \dots j]$ with p:

- i as small as possible (first) breaking ties by
- j as large as possible (longest)

This problem arises in just about any application that displays text.

This has two obvious measures of problem size:

- size of the string, n
- size of the pattern, m

Example 4: String Search With Wildcards

Relation

Let's think about the last character in the pattern.

If $s[i \ldots j]$ matches the pattern, then $s[j]$ must match $p[m]$:

- if $p[m]$ is a letter, $s[j]$ is that same letter
- if $p[m]$ is '?' or '*', then $s[j]$ can be any letter

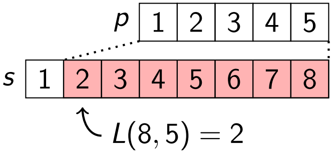
What must $s[i \ldots j-1]$ match?

- if $p[m]$ is a letter or '?', then it matches $p[1 \ldots m-1]$
- if $p[m]$ is '*', then it could match $p[1 \ldots m-1]$ or $p[1 \ldots m]$ (Why?)

It seems that we need to think about prefixes of both the string s and the pattern p...

Example 4: String Search With Wildcards

Relation (cont)



Let $L(j,k)$ be the start of the longest match that:

- ends at $s[j]$
- matches $p[1 \dots k]$

Then we have just argued above that:

- if $p[k] \in$ '?' or $p[k] = s[j]$, then $L(j,k) = L(j-1,k-1)$
- if $p[k] =$ '$*$', then $L(j,k) = \min\{L(j-1,k-1), L(j-1,k)\}$ (Why?)

Example 4: String Search With Wildcards

Pseudocode

```
WILDCARD-MATCHES(s, n, p, m)
L ← NEW-MATRIX()
for k ← 1 to n
do L[0, k] ← ∞
for j ← 1 to n
do L[j, 0] ← j − 1
    for k ← 1 to W
    do switch
            case p[k] = s[j]  or  p[k] = '?' :
                L[j, k] ← L[j − 1, k − 1]
            case p[k] = '*' :
                L[j, k] ← min{L[j − 1, k − 1], L[j − 1, k]}
            case default :
                L[j, k] ← −∞
    if L[j, m] = j − n + 1
      then return (L[j, m], j)
return "no match"
```

Runs in $O(nm)$ time. Can optimize to use $O(m)$ space. (How?)

- In practice m is small (say, $m \leq 100$), so this is very fast.

Example 4: String Search With Wildcards

Observations

Like most dynamic programming algorithms, here it is easy to:

- analyze the efficiency
- optimize space

Most of the work is in working out how to relate the solution of the whole problem to the solutions of subproblems.

Reflecting on the relation we worked out in this case:

- if we have a match of $s[i \dots j]$ with $p[1 \dots k]$, the only part that affects whether $s[i \dots j+1]$ matches is $p[k]$

  (hence, we consider all choices of k — brute force)
- there may be many ways of matching suffixes of $s[1 \dots j]$ to $p[1 \dots k]$, but we only need one (with i as small as possible)
- both are typical of efficient dynamic programming algorithms

Example 4: String Search With Wildcards

Generalizations

A more general problem is to find the first match of $s[i \dots j]$ to p, where p is an arbitrary regular expression.

The algorithms that do this are very similar to this one.

- main difference is that, rather than keeping track of characters of p, we keep track of states of an NFA
- translate regular expression to an NFA, then apply this algorithm

Example 5: Edit Distance

Problem Definition

We are given strings $s[1 \dots n]$ and $t[1 \dots m]$.

Our goal is to find the least costly way to convert s into t by:

- inserting or deleting a character, with cost α or β
- substituting b for a, with cost $\gamma_{a,b}$

For example, suppose that $s =$ “tab” and $t =$ “out”.

- delete ‘t’, ‘a’, ‘b’, then insert ‘o’, ’u’, ’t’: cost $3\alpha + 3\beta$
- delete ‘a‘, ‘b’, then insert ‘u’, ‘o’ before ‘t’: cost $2\alpha + 2\beta$
- substitute ‘o’ for ‘t’, ‘u’ for ‘a’, ‘t’ for ‘b’:
  cost $\gamma_{t,o} + \gamma_{a,u} + \gamma_{b,t}$

This problem is solved in many spell checkers.

This problem is equivalent to *sequence alignment*, which is critically important in computational biology.

Example 5: Edit Distance

Relation

We can see that there are two size dimensions, n and m.

After example 4, it may already be clear we should consider the distance between $s[1 \dots i]$ and $t[1 \dots j]$. Call this $\mathrm{OPT}(i, j)$.

As in our previous examples, consider what happens with the last characters:

- if $t[j]$ is inserted, then cost is $\alpha + \mathrm{OPT}(i, j-1)$
- if $s[i]$ is deleted, then cost is $\beta + \mathrm{OPT}(i-1, j)$
- if $t[j]$ is substituted for $s[i]$, then cost is $\gamma_{s[i],t[j]} + \mathrm{OPT}(i-1, j-1)$

$$\mathrm{OPT}(i,j) = \min \begin{cases} \mathrm{OPT}(i, j-1) + \alpha, \\ \mathrm{OPT}(i-1, j) + \beta, \\ \mathrm{OPT}(i-1, j-1) + \gamma_{s[i],t[j]} \end{cases}$$

Example 5: Edit Distance

Pseudocode

```
EDIT-DISTANCE(s, n, t, m)
    opt ← NEW-MATRIX()
    for j ← 0 to m
    do opt[0, j] ← ∞
    for i ← 1 to n
    do opt[i, 0] ← ∞
        for j ← 1 to m
        do opt[i, j] ← min{opt(i, j − 1) + α,
                            opt(i − 1, j) + β,
                            opt(i − 1, j − 1) + γ_{s[i],t[j]}}
    return opt[n, m]
```

Runs in $O(nm)$ time and (optimized) $O(\min\{n, m\})$ space. (How?)

We can also compute the solution, not just its value. (How?)
Can we compute the solution in $O(\min\{n, m\})$ space? (See book.)

Counting Solutions

It is also possible to count the number of optimal solutions.

We can produce a relation for this number $\mathrm{NUM}(i,j)$ based on our relation for $\mathrm{OPT}(i,j)$:

$$\mathrm{NUM}(i,j) = \begin{array}{l} [\mathrm{OPT}(i,j) = \mathrm{OPT}(i,j-1) + \alpha] \cdot \mathrm{NUM}(i,j-1) + \\ [\mathrm{OPT}(i,j) = \mathrm{OPT}(i-1,j) + \beta] \cdot \mathrm{NUM}(i,j-1) + \\ [\mathrm{OPT}(i,j) = \mathrm{OPT}(i-1,j-1) + \alpha] \cdot \mathrm{NUM}(i-1,j-1). \end{array}$$

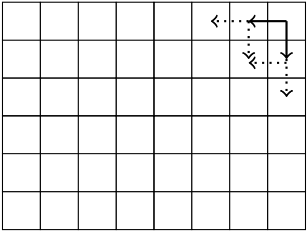
Here, [P] means 1 if P is true and 0 if not.

Hence, we can compute NUM by dynamic programming as well.

Counting Solutions

Interview Question

Count the number ways a robot can move from (n, m) to $(1, 1)$ on a grid, moving only down or left.



Intented solution was the one we just looked at.

Counting Solutions

Interview Question (cont)

Unfortunately, this is a poor interview question because it's too easy. The answer doesn't need to be computed. It is exactly:

$$\binom{n+m-2}{n-1}$$

Counting Solutions

Interview Question (cont)

Unfortunately, this is a poor interview question because it's too easy. The answer doesn't need to be computed. It is exactly:

$$\binom{n+m-2}{n-1}$$

A friend gave the this answer, which was not the intended solution.

- "How would you write a program to produce the answer?"

Counting Solutions

Interview Question (cont)

Unfortunately, this is a poor interview question because it's too easy. The answer doesn't need to be computed. It is exactly:

$$\binom{n+m-2}{n-1}$$

A friend gave the this answer, which was not the intended solution.

- "How would you write a program to produce the answer?"
- "I'd write: `print Choose(n+m-2,n-1)`."

Counting Solutions

Interview Question (cont)

Unfortunately, this is a poor interview question because it's too easy. The answer doesn't need to be computed. It is exactly:

$$\binom{n+m-2}{n-1}$$

A friend gave the this answer, which was not the intended solution.

- "How would you write a program to produce the answer?"
- "I'd write: `print Choose(n+m-2,n-1)`."

Could make a workable problem by allowing some substitutions.

Example 6: Shortest Path

Problem Definition

Give a graph G, edge lengths $\ell_{i,j}$, and nodes s and t. Find the shortest path from s to t.

- familiar problem with many applications
- actually a generalization of edit distance problem

There is a greedy algorithm for computing shortest paths.

- that algorithm does not work with negative weights
- another example where dynamic programming is more general

The algorithm we will see is also very important.

- variants of this are used in real Internet routers
- optimized implementations are faster than greedy

Example 6: Shortest Path

Relation

Our usual technique of considering the last node or edge does not work well in this case. (It works, but it's tricky.)

Suppose we knew that the optimal solution had k edges.

Let $\mathrm{OPT}(k, v)$ be the shortest path from s to v using $\leq k$ edges.

- problem is to find $\mathrm{OPT}(n-1, t)$ (Why?)

Example 6: Shortest Path

Relation

Our usual technique of considering the last node or edge does not work well in this case. (It works, but it's tricky.)

Suppose we knew that the optimal solution had k edges.

Let $\mathrm{OPT}(k, v)$ be the shortest path from s to v using $\leq k$ edges.

- problem is to find $\mathrm{OPT}(n-1, t)$ (Why?)
- if the last edge is $(w, v) \in E$, then optimal cost must be $\mathrm{OPT}(k-1, w) + \ell_{w,v}$ (Why?)

Example 6: Shortest Path

Relation

Our usual technique of considering the last node or edge does not work well in this case. (It works, but it's tricky.)

Suppose we knew that the optimal solution had k edges.

Let $\mathrm{OPT}(k, v)$ be the shortest path from s to v using $\leq k$ edges.

- problem is to find $\mathrm{OPT}(n-1, t)$ (Why?)
- if the last edge is $(w, v) \in E$, then optimal cost must be $\mathrm{OPT}(k-1, w) + \ell_{w,v}$ (Why?)

Thus, we have the relation:

$$\mathrm{OPT}(k, v) = \min_{w \in V,\ (v,w) \in E} \mathrm{OPT}(k-1, w) + \ell_{w,v}$$

Example 6: Shortest Path

Relation

```
SHORTEST-PATH(n, s, t, E, ℓ)
opt ← NEW-MATRIX()
for v ← 1 to n
do opt[0, v] ← ∞
opt[0, s] ← 0
for k ← 1 to n − 1
do for v ← 1 to n
    do opt[k, v] ← opt[k − 1, v]
        for w  such that (v, w) ∈ E
        do opt[k, v] ← min{opt[k, v], opt[k − 1, w] + ℓ_{w,v}}
return opt[n − 1, t]
```

Running time is $O(n^3)$. (Is that right?)

Example 6: Shortest Path

Relation

$\textsc{Shortest-Path}(n, s, t, E, \ell)$

$opt \leftarrow \textsc{New-Matrix}()$
for $v \leftarrow 1$ **to** n
do $opt[0, v] \leftarrow \infty$
$opt[0, s] \leftarrow 0$
for $k \leftarrow 1$ **to** $n - 1$
do for $v \leftarrow 1$ **to** n
    do $opt[k, v] \leftarrow opt[k - 1, v]$
        for w such that $(v, w) \in E$
        do $opt[k, v] \leftarrow \min\{opt[k, v], opt[k - 1, w] + \ell_{w,v}\}$
return $opt[n - 1, t]$

Running time is $O(nm)$. Can be optimized to use $O(n)$ space.

We can find the solution from just the last row. (Why?)

Design Heuristics

We have seen that the hard part of dynamic programming is figuring out how to relate the solution of the whole problem to the solution of subproblems.

Now that we've seen many examples, we can look for patterns.

In each case, the relation was found by asking ourselves two questions. . . .

Design Heuristics

Heuristic #1: Last Item

How does the last item of the input contribute to the solution?

- interval scheduling: is the last interval included?
- knapsack: is the last item included?

  identical reasoning to interval scheduling

  note that the second dimension (weight) suggested itself by thinking about last item
- edit distance: how are the last characters of s and t used?

Design Heuristics

Heuristic #2: Guess a Variable

What information, if we knew it, would make this problem easy?

Try all possibilities for that value (brute force).

- shortest path: length of the shortest path
- maximum subarray sum: where does the subarray end

The ability to guess the value of any variable we want is quite powerful.

Wildcard matching was solved by asking both questions.

- guess a variable: where does the match end in s?

  suppose it ends at $s[j]$...
- last input: how is $p[m]$ used to match $s[1 \dots j]$?

This was perhaps the most complex example, but it too required only asking ourselves these two questions.

When is Dynamic Programming Efficient?

Ordering

The only hard and fast rule is: try it and see how many subproblems you get.

However, in the examples, we only had to consider:

- every prefix $1 \dots i$ of the input, $O(n)$
- every range $i \dots j$ of the input, $O(n^2)$

This happened because of the way the inputs were ordered:

- order was given: max subarray sum, wildcard matching, edit distance
- order was unimportant (so we could pick any order): knapsack, shortest paths
- we found a clever ordering: interval scheduling

Let's look at one final example where this also occurs...

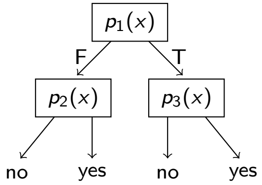
Example 7: Optimal Decision Trees

Ordering

We are given:

- a set of keys $x_1, \ldots x_n$
- probability p_i that each x_i will be requested
- probability q that some $x \notin \{x_1, \ldots x_n\}$ will be requested

Goal is to design a decision tree to answer $x \in \{x_1, \ldots, x_n\}$ with expected access path length is as low as possible.



Example 7: Optimal Decision Trees

Complexity

In general, this problem is NP-hard.

For example, suppose that we use $f_i(x) = \lfloor x/2^i \rfloor \bmod 2$. (I.e., $f_i(x)$ is the i-th bit of x.)

At the each node, we can consider using f_1, f_2, ..., f_m.

In that case, the subproblems that arise are the subsets of $x_1, \ldots, x_n$ whose i_1-th bit is equal to b_1, i_2-th bit is equal to b_2, and so on.

Unfortunately, we can't say anything about what those subsets look like. It may be that we have to solve the problem for all 2^n subsets, which would not be efficient.

Example 7: Optimal Decision Trees

Ordering

Suppose that we want to use $f_i(x) = [x \le x_i]$.

Now, the subproblems that arise are over the subsets with each x satisfying $x \ge x_{i_1}$, $x \ge x_{i_2}$, $x \le x_{j_1}$, and so on.

But this is equivalent to $x \in [\max x_{i_k}, \min x_{j_k}] = [x_i, x_j]$, for some i and j.

Hence, we can sort the x_i's and then solve the subproblems corresponding to all $O(n^2)$ intervals.

In fact, the problem in this case is finding an optimal binary search tree, which is efficiently solvable using dynamic programming.

In summary, if the inputs are ordered or can be ordered in some useful way, then this is a clue that dynamic programming may be efficient. (Still, you should always try it and count the subproblems.)