Dynamic Programming

Slides by Kevin Zatloukal (with modifications)

October 20, 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
- 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
- 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
- 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
- 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
- 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
- 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

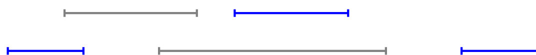
Example 1: Weighted Interval Scheduling

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

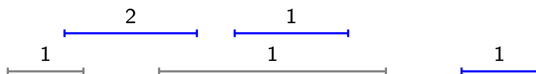
Problem Definition

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

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 $\text{OPT}(I')$ 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.
- What is the value of the optimal solution that does use ℓ_n?
- It must be $w_n + \text{OPT}(I - \text{conflicts}(\ell_n))$, where $\text{conflicts}(\ell_n)$ is the set of intervals overlapping ℓ_n.
- Hence, we must have:

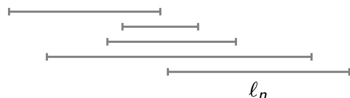
$$\text{OPT}(I) = \max\{\text{OPT}(I - \{\ell_n\}), w_n + \text{OPT}(I - \text{conflicts}(\ell_n))\}.$$

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

Pseudcode

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)
1  sort start, finish, weight by finish
2  opt-val ← NEW-ARRAY()
3  opt-val[0] ← 0
4  for i ← 1 to n
5  do j ← BINARY-SEARCH(start[i], finish, n)
6       opt-val[i] ← max{opt-val[i − 1], weight[i] + opt-val[j − 1]}
7  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 $\mathrm{OPT}(J)$ 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 $\text{OPT}(i) = w_i + \text{OPT}(p(i) - 1)$ or equivalently iff $\text{OPT}(i) > \text{OPT}(i - 1)$

Example 1: Weighted Interval Scheduling

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

```
OPTIMAL-WEIGHTED-INTERVALS(opt-val, n)
 8  opt-set ← ∅
 9  i ← n
10  while i > 0
11   do if opt-val[i] > opt-val[i − 1]
12         then opt-set ← opt-set ∪ {i}
13              i ← BINARY-SEARCH(start[i], finish, n) − 1
14         else  i ← i − 1
15  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 $A[i \dots j]$ 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 $\mathrm{OPT}(1 \dots n-1)$

0	-1	2	-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)
16  opt ← 0, opt' ← 0
17  for i ← 1 to n
18  do opt' ← max{0, opt' + A[i]}
19      opt ← max{opt, opt'}
20  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.

Etymology 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"

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

Example 3: Knapsack

Relation

Consider the last item, n:

- if n is not in the optimal solution, then we have $\text{OPT}([n]) = \text{OPT}([n-1])$
- but if n is in the optimal solution, then the rest of the optimal solution need not be $\text{OPT}([n-1])$ (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)
21  opt ← NEW-MATRIX()
22  for V ← 1 to W
23  do opt[0, V] ← 0
24  for k ← 1 to n
25  do for V ← 1 to W
26        do opt[k, V] ← max{opt[k − 1, V − w[k]] + w[k],
27                             opt[k − 1, V]}
28  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)
29  opt ← NEW-MATRIX()
30  for V ← 1 to W
31  do opt[0, V] ← 0
32  for k ← 1 to n
33  do for V ← 1 to W
34        do opt[k, V] ← max{opt[k − 1, V − w[k]] + w[k],
35                           opt[k − 1, V]}
36  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*.

Example 4: Edit Distance

Problem Definition

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

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 $3\alpha + 3\beta$
- insert 'o', 'u', keep 't' delete 'a', 'b': 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 4: Edit Distance

Relation

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

Our subproblems will consider alignments between $s[1 \ldots i]$ and $t[1 \ldots j]$. Call this $\mathrm{OPT}(i, j)$.

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

- if $t[j]$ is inserted last, then cost is $\alpha + \mathrm{OPT}(i, j-1)$
- if $s[i]$ is deleted last, 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 4: Edit Distance

Pseudocode

```
EDIT-DISTANCE(s, n, t, m)
37  opt ← NEW-MATRIX()
38  for j ← 0 to m
39  do opt[0, j] ← ∞
40  for i ← 1 to n
41  do opt[i, 0] ← ∞
42      for j ← 1 to m
43      do opt[i, j] ← min{opt(i, j − 1) + α,
44                         opt(i − 1, j) + β,
45                         opt(i − 1, j − 1) + γ_{s[i],t[j]}}
46  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 $\mathrm{OPT}(i,j)$:

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

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

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

Example 5: 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 5: 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 5: 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 $\text{OPT}(k, v)$ be the shortest path from s to v using $\leq k$ edges.

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

Example 5: 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 5: Shortest Path

Relation

```
SHORTEST-PATH(n, s, t, E, ℓ)
47  opt ← NEW-MATRIX()
48  for v ← 1 to n
49  do opt[0, v] ← ∞
50  opt[0, s] ← 0
51  for k ← 1 to n − 1
52  do for v ← 1 to n
53         do opt[k, v] ← opt[k − 1, v]
54             for w  such that (v, w) ∈ E
55             do opt[k, v] ← min{opt[k, v], opt[k − 1, w] + ℓ_{w,v}}
56  return opt[n − 1, t]
```

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

Example 5: Shortest Path

Relation

```
SHORTEST-PATH(n, s, t, E, ℓ)
57  opt ← NEW-MATRIX()
58  for v ← 1 to n
59  do opt[0, v] ← ∞
60  opt[0, s] ← 0
61  for k ← 1 to n − 1
62  do for v ← 1 to n
63        do opt[k, v] ← opt[k − 1, v]
64           for w  such that (v, w) ∈ E
65           do opt[k, v] ← min{opt[k, v], opt[k − 1, w] + ℓ_{w,v}}
66  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 several 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?

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.

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 considered things like:

- every prefix $1 \ldots i$ of the input, $O(n)$
- every pair of prefixes (i, j) of the input, $O(n^2)$

This happened because of the way the inputs were ordered:

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