Dynamic Programming

Tao Hou

Outline

1. Some theory or guidelines
2. Dynamic programming problems:
    1. Longest increasing subsequence (entry-level)
    2. Text segmentation (entry-level)
    3. Knapsack (in-the-middle)
    4. Edit distance (advanced)
    5. Matrix-chain multiplication (advanced)

Dynamic Programming

- Dynamic programming does not refer to writing code
- The term was coined in the 1950's. It referred to scheduling/planning, and typically involves filling out a table
- Dynamic programming is often used to solve *optimization* problems such as computing a longest sequence, shortest path, etc. (there are exceptions though)

Dynamic Programming vs. Divide-and-Conquer

Both *decompose a problem into subproblems*, but there are differences:

1. Subproblem size:
   - **Divide-and-conquer**: breaks the problem into *substantially smaller* subproblems (e.g. $n \Rightarrow n/2$ or $7n/10$)
   - **Dynamic programming**: typically reduces a problem of size n to one of size $n - c$ (e.g. $n \Rightarrow n - 1$ or $n - 2$)
2. Disjoint vs. Overlap
   - **Divide-and-conquer**: splits the problem into *disjoint* subproblems
   - **Dynamic programming**: subproblems typically *overlap*
     - So, recursion does not work so well for dynamic programming

Bookkeeping using a table

Due to the previous reasons:

- A dynamic-programming algorithm solves each subproblem *just once* and then *saves its answer in a table*, avoiding repetitive computation of subproblems (*trade space for time*)
- Therefore, a dynamic-programming algorithm typically solves *smaller subproblems first* (and keep the results in memory), then bigger problems, because bigger ones rely on the results of the smaller subproblems.

Example of overlap computation

The Fibonacci Sequence

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Recursive Definition

$$F(n) = \begin{cases} 1 & \text{if } n = 1, 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

A Recursive Fibonacci Algorithm

Rec-Fib(n)

1. **if** $n = 1$ or $n = 2$ **then return** 1;
2. **else return** Rec-Fib($n - 1$) + Rec-Fib($n - 2$);

Figure 1: The recursive Fibonacci algorithm

Input Size: n

$$T(n) = \begin{cases} O(1) & \text{if n=1, 2} \\ T(n-1) + T(n-2) + O(1) & \text{otherwise} \end{cases}$$

$T(n) = \Theta(\phi^n)$, where $\phi \approx 1.618$ is the Golden Ratio

What Went Wrong?

Overlapping Computations

Many Fibonacci numbers were recomputed over and over again.

Solution

Instead of using a top-down approach, use a bottom-up approach and *store* the Fibonacci numbers for *reuse* once they have been computed

Bottom-UP Fibonacci (n)

1. create a table $F[1 \ldots n]$; $F[1] = F[2] = 1$;
2. **for** $i = 3$ **to** n **do** $F[i] = F[i-1] + F[i-2]$;

Figure 2: Bottom-Up Fibonacci algorithm

Running Time: $T(n) = \Theta(n)$

Four steps of dynamic programming

1. Characterize the structure of the problem by identifying an *optimal solution function* (typically denoted as OPT)
2. Write down a recursively formula for the optimal solution function (you need to consider both the *base* case and the *general* case)
3. Compute the value of optimal solution, typically in a bottom-up fashion
4. (Optional) Construct an optimal solution from computed information

Four steps of dynamic programming

1. Characterize the structure of the problem by identifying an *optimal solution function* (typically denoted as OPT)
2. Write down a recursively formula for the optimal solution function (you need to consider both the *base* case and the *general* case)
3. Compute the value of optimal solution, typically in a bottom-up fashion
4. (Optional) Construct an optimal solution from computed information

We can use computing Fibonacci numbers as an example

More on the steps of dynamic programming

Step 1 and 2 boils down to:

- Identify the **subproblems**
- Describe the solution to the whole problem in terms of the solutions to **smaller** subproblems.

(This is the hard part!)

More on the steps of dynamic programming

How to write an algorithm based on the recurrence in Step 3:

- *Choose a table.* Find a data structure that can store solutions to all subproblems you identified (usually a *multidimensional array*).
- *Identify dependencies.* Except for the base cases, every subproblem depends on other subproblems. Identify the dependencies.
- *Find a good* ***evaluation order****.* Order the subproblems based on their dependencies so that each problem comes after the subproblems it depends on.
  - A useful rule for the coining dependency is that your subproblems should have a natural notion of 'size' so that larger subproblems should depend on smaller subproblems
  - The evaluation order is then to evaluate smaller subproblems first, and then the larger subproblems

Optimal Substructure Property: An Essential Condition

Optimal Substructure Property. In order for DP to work, you have to be able to easily combine optimal solutions for subproblems to build an optimal solution to the original (bigger) problem

Optimal Substructure Property: Example

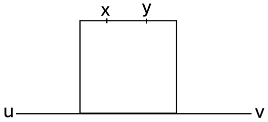
Shortest path: given $G = (V, E)$, find the a path with minimum number of edges (i.e., the shortest path) from u to v

- decompose $u \rightsquigarrow v$ into $u \rightsquigarrow x \rightsquigarrow y \rightsquigarrow v$
- easy to prove that, if $u \rightsquigarrow x \rightsquigarrow y \rightsquigarrow v$ is shortest then $x \rightsquigarrow y$ is also shortest
- i.e., a shortest path is made up of several (smaller) shortest paths
- this is the *optimal substructure property*

Optimal Substructure Property: Counterexample

Longest simple path: given $G = (V, E)$, find the length of the longest simple (i.e., no cycles) path from u to v

- we can also decompose $u \leadsto v$ into $u \leadsto x \leadsto y \leadsto v$
- however, we can not prove that, if $u \leadsto x \leadsto y \leadsto v$ is maximal, then $x \leadsto y$ is also maximal (e.g., the below graph)



Why we study DP?

Dynamic Programming Applications Areas:

- Bioinformatics
- Control theory
- Information theory
- Operations research
- Computer science: theory, graphics, AI, compilers, systems, ...

Some famous dynamic programming algorithms:

- Unix diff for comparing two files
- Viterbi for hidden Markov models
- Smith-Waterman for genetic sequence alignment
- Bellman-Ford for shortest path routing in networks
- Cocke-Kasami-Younger for parsing context free grammars

Longest Increasing Subsequence

Problem Definition

Given a sequence of numbers $A = \langle a_1, \ldots, a_n \rangle$, compute a *subsequence* with maximum length whose elements are increasing

Example

$$A = 5, 2, 8, 6, 3, 6, 9, 7$$

Solution: 2, 3, 6, 7

More on Subsequence

- Formally, a *subsequence* of $a_1, a_2, \ldots, a_n$ can be represented as

$$S = a_{s_1}, a_{s_2}, \ldots, a_{s_\ell},$$

where the subscripts satisfy $1 \leq s_1 < s_2 < \cdots < s_\ell \leq n$.

More on Subsequence

- Formally, a *subsequence* of $a_1, a_2, \ldots, a_n$ can be represented as

$$S = a_{s_1}, a_{s_2}, \ldots, a_{s_\ell},$$

  where the subscripts satisfy $1 \leq s_1 < s_2 < \cdots < s_\ell \leq n$.

- Then, an *increasing* subsequence should satisfy:

$$a_{s_1} < a_{s_2} < \cdots < a_{s_\ell},$$

Approaching the problem

- In order to solve DP problems, we typically focus on some special traits of problems
- In this specific problem, since we want to find increasing subsequences, we focus on the *ending element* of a sequence
  - The reason should be evident after the problem is solved
- Our first major progress on solving the problem is then: for each a_i, we try to find the longest increasing subsequence **ending with** it
  - These are the *subproblems* we identify
  - The longest increasing subsequence of A can then be derived from solutions to the subproblems

DP Solution, Step 1: Optimal solution function

- Let $OPT(i)$ denote the length of the longest increasing subsequence (LIS) of A ending with a_i
- The length of the LIS of A is then $\max\{OPT(i) \mid 1 \leq i \leq n\}$

Step 2: Writing the Recursive Formula for $OPT(i)$

- Goal: Express $OPT(i)$ in terms of other $OPT(j)$'s

Step 2: Writing the Recursive Formula for $OPT(i)$

- Goal: Express $OPT(i)$ in terms of other $OPT(j)$'s
- Suppose we have S as the LIS ending with a_i

Step 2: Writing the Recursive Formula for $OPT(i)$

- Goal: Express $OPT(i)$ in terms of other $OPT(j)$'s
- Suppose we have S as the LIS ending with a_i
- What would the form of S be?
- Well, in S, there must be another element a_j **immediately before** a_i

Step 2: Writing the Recursive Formula for $OPT(i)$

- Goal: Express $OPT(i)$ in terms of other $OPT(j)$'s
- Suppose we have S as the LIS ending with a_i
- What would the form of S be?
- Well, in S, there must be another element a_j **immediately before** a_i
- Observe: Since S is the LIS ending with a_i, the part of S excluding a_i must be the **LIS ending with** a_j
  - The *optimal substructure property*
  - Provable by *cut-and-paste*: If the part of S excluding a_i is not LIS ending with a_j, we can replace it with the LIS ending with a_j, getting a longer sequence ending with a_i (a contradiction)

Step 2: Writing the Recursive Formula for $OPT(i)$

- This means the that LIS ending with a_i consists of:

$$[\text{LIS ending with } a_j] + [a_i]$$

where a_j is an element before a_i

Step 2: Writing the Recursive Formula for $OPT(i)$

- This means the that LIS ending with a_i consists of:

$$[\text{LIS ending with } a_j] + [a_i]$$

  where a_j is an element before a_i
- So, to get LIS ending with a_i, we must have an a_j before a_i and the LIS ending with it

Step 2: Writing the Recursive Formula for $OPT(i)$

- This means the that LIS ending with a_i consists of:

$$[\text{LIS ending with } a_j] + [a_i]$$

  where a_j is an element before a_i
- So, to get LIS ending with a_i, we must have an a_j before a_i and the LIS ending with it
- Q: How do we get the LIS ending with an a_j?

Step 2: Writing the Recursive Formula for $OPT(i)$

- This means the that LIS ending with a_i consists of:

$$[\text{LIS ending with } a_j] + [a_i]$$

  where a_j is an element before a_i
- So, to get LIS ending with a_i, we must have an a_j before a_i and the LIS ending with it
- Q: How do we get the LIS ending with an a_j?
- A: This is the subproblem and we can assume we know it, which is $OPT(j)$

Step 2: Writing the Recursive Formula for $OPT(i)$

- This means the that LIS ending with a_i consists of:

$$[\text{LIS ending with } a_j] + [a_i]$$

  where a_j is an element before a_i
- So, to get LIS ending with a_i, we must have an a_j before a_i and the LIS ending with it
- Q: How do we get the LIS ending with an a_j?
- A: This is the subproblem and we can assume we know it, which is $OPT(j)$
- Q: How do we know which a_j to choose?

Step 2: Writing the Recursive Formula for $OPT(i)$

- This means the that LIS ending with a_i consists of:

$$[\text{LIS ending with } a_j] + [a_i]$$

  where a_j is an element before a_i
- So, to get LIS ending with a_i, we must have an a_j before a_i and the LIS ending with it
- Q: How do we get the LIS ending with an a_j?
- A: This is the subproblem and we can assume we know it, which is $OPT(j)$
- Q: How do we know which a_j to choose?
- A: We don't know. Rather, we enumerate all possibilities and choose the longest one

Recursive formula for $OPT(i)$

$$OPT(i) = \begin{cases} 1 & \text{if } i = 1 \\ \max\left\{\{1\} \cup \{1 + OPT(j) \mid 1 \leq j < i \wedge a_j < a_i\}\right\} & \text{otherwise} \end{cases}$$

The answer to the input is:

$$\max\{OPT(i) \mid 1 \leq i \leq n\}$$

Some remarks

- In the previous solution, the problem of finding LIS ending with a_i relies on a bunch of **subproblems** of finding LIS ending with a_j ($j < i$)
- We can view the "size" of a subproblem as the subscript of the ending element, e.g., for the problem of finding the LIS ending with a_i, the "size" is i
- So, we have a problem of size i relies on a bunch of problems of size j ($j < i$)
  $\Rightarrow$ Solving a **bigger** problem relies solving a bunch of **smaller** problems

The Algorithm

$PREV[i]$: index of the element immediately before a_i in the LIS ending with a_i (used to the recover the LIS ending with a_i)

LIS(A)

1. Use a table $OPT[1 \dots n]$ and $PREV[1 \dots n]$
2. initialize $OPT[1] = 1$; $PRVE[i] = null$ for $i = 1, \dots, n$;
3. **for** $i = 2$ **to** n **do**
   3.1. $OPT[i] = 1$
   3.2. **for** $j = 1$ **to** $i - 1$ **do**
   if $A[j] < A[i]$ and $OPT[j] + 1 > OPT[i]$ **then**
   $OPT[i] = OPT[j] + 1$; $PREV[i] = j$
4. **return** $\max\{OPT[1], OPT[2], \dots, OPT[n]\}$

Figure 3:

Running Time

The running time is clearly $O(n^2)$.

The Algorithm

PRT-LIS()

1. $i = \arg\max_i \{OPT[i]\}$
2. **while** $i \neq \mathit{null}$ **do**
    2.1. print $A[i]$
    2.2. $i = PREV[i]$
3. reverse what is printed

Figure 4: Recover the LIS ending with a_i

Exercise on Board

- Let's try to compute on board the OPT and $PREV$ arrays for LIS on the example input

Remarks

- A more "natural" definition of OPT would be: Let $OPT(i)$ denote the length of the longest increasing subsequence of $\langle a_1, \ldots, a_i \rangle$
  - However, this does not help solve the problem
  - You can think of why by, say, trying to solve the problem with this alternative definition
- By enforcing the ending element in our OPT definition, we can ensure that the subsequences being considered are increasing

Some guidelines on designing OPT function:

1. Before defining OPT, first determine what your **subproblems** should be
   - The subproblems should also contain the **original problem** and **a clear base case**
   - OPT is nothing but the optimal solutions for subproblems
2. The parameter of OPT should relate to the "size" of a subproblem so that optimal solution of a larger subproblem can be solved from solutions of smaller subproblems
   - Specifically, you should be able to easily derive a *base case* of your OPT which is of the "smallest size"
3. Ultimately, you choice of OPT should enable you to write down a recursive formula for it, which should be computable (can find an order for evaluating all the OPT entries)

Text Segmentation

Problem

We are given a string $s[1 \dots n]$ and a subroutine $dict(w)$ that determines whether a given string w is a valid word (assume this can be done in constant time). We want to know whether s can be partitioned into a sequence of valid words.

Example

s=algorithmsisacomputersciencecourse

Solution: s is valid; a valid decomposition of s:

algorithms is a computer science course

Why DP?

- The first algorithm someone could come with up would be a simple "greedy matching" algorithm:
  - Scan the string from the beginning.
  - Whenever you find a match to a word, stop, and mark this as a separation point
  - You then do the matching again starting from the previous separation point until you hit the end

Why DP?

- The first algorithm someone could come with up would be a simple "greedy matching" algorithm:
  - Scan the string from the beginning.
  - Whenever you find a match to a word, stop, and mark this as a separation point
  - You then do the matching again starting from the previous separation point until you hit the end
- But this algorithm is wrong
- Suppose our dictionary contains only three words: *abc*, *abcd*, *ef*
- Given an input "abcdef", the above algorithm would first find a match when "abc" is scanned, leaving "def" without a match
- But "abcdef" can be separated into two valid words: "abcd" and "ef"

The dynamic programming ingredients

Key observation

If the string s can be split into two substrings $s[1 \ldots i]$ and $s[i+1 \ldots n]$ s.t.

- $s[1 \ldots i]$ can be partitioned into valid words
- $s[i+1 \ldots n]$ is a valid word

then the whole string s can also be partitioned.

E.g., 'algorithmsisacomputersciencecourse' can be split into:

- 'algorithmsisacomputerscience': algorithms is a computer science
- 'course': is a valid word

So 'algorithmsisacomputersciencecourse' can be partitioned

The dynamic programming ingredients

Key observation

If the string s can be split into two substrings $s[1 \dots i]$ and $s[i+1 \dots n]$ s.t.

- $s[1 \dots i]$ can be partitioned into valid words
- $s[i+1 \dots n]$ is a valid word

then the whole string s can also be partitioned.

Notice that we are considering a general prefix $s[1 \dots i]$ of s.
Indeed, all such prefixes constitute the **subproblems** of our solution:

- There is a natural 'size' for each such subproblem, i.e., the size of the prefix
- When $i = n$, it gives the answer to the original problem.
- It has a clear base case, i.e., $i = 0$ (empty string)

The dynamic programming ingredients

Key observation

If the string s can be split into two substrings $s[1 \ldots i]$ and $s[i+1 \ldots n]$ s.t.

- $s[1 \ldots i]$ can be partitioned into valid words
- $s[i+1 \ldots n]$ is a valid word

then the whole string s can also be partitioned.

Now let $OPT(i)$ denote whether $s[1 \ldots i]$ can be partitioned into valid words, i.e.,

$$OPT(i) = \begin{cases} \text{True} & \text{if } s[1 \ldots i] \text{ can be partitioned into valid words} \\ \text{False} & \text{otherwise} \end{cases}$$

Writing down recursive formula for $OPT(i)$

- Observe: If $s[1 \ldots i]$ can be split into two substrings $s[1 \ldots j]$ and $s[j+1 \ldots i]$ s.t.
  1. $s[1 \ldots j]$ can be partitioned into valid words
  2. $s[j+1 \ldots i]$ is a valid word

  then $s[1 \ldots i]$ can also be partitioned.
- Let choose a certain j and fix it
- Easy to know whether $s[j+1 \ldots i]$ is a valid word: $dict(s[j+1 \ldots i])$
- How do we know whether $s[1 \ldots j]$ can be partitioned?

Writing down recursive formula for $OPT(i)$

- Observe: If $s[1 \dots i]$ can be split into two substrings $s[1 \dots j]$ and $s[j+1 \dots i]$ s.t.
  1. $s[1 \dots j]$ can be partitioned into valid words
  2. $s[j+1 \dots i]$ is a valid word

  then $s[1 \dots i]$ can also be partitioned.
- Let choose a certain j and fix it
- Easy to know whether $s[j+1 \dots i]$ is a valid word: $dict(s[j+1 \dots i])$
- How do we know whether $s[1 \dots j]$ can be partitioned?
  - *It's a subproblem (substructure property)!*
  - We can assume we know the answer which is $OPT(j)$
  - Moreover, in order to solve the problem that whether $s[1 \dots i]$ is valid, we rely on the **subproblem** that whether $s[1 \dots j]$ is valid ($j < i$) (large problem relies on smaller problems)

Writing down recursive formula for $OPT(i)$

- Observe: If $s[1 \dots i]$ can be split into two substrings $s[1 \dots j]$ and $s[j+1 \dots i]$ s.t.
  1. $s[1 \dots j]$ can be partitioned into valid words
  2. $s[j+1 \dots i]$ is a valid word

  then $s[1 \dots i]$ can also be partitioned.
- Let choose a certain j and fix it
- Easy to know whether $s[j+1 \dots i]$ is a valid word: $dict(s[j+1 \dots i])$
- How do we know whether $s[1 \dots j]$ can be partitioned?
  - *It's a subproblem (substructure property)!*
  - We can assume we know the answer which is $OPT(j)$
  - Moreover, in order to solve the problem that whether $s[1 \dots i]$ is valid, we rely on the **subproblem** that whether $s[1 \dots j]$ is valid ($j < i$) (large problem relies on smaller problems)
- But how do we determine j?

Writing down recursive formula for $OPT(i)$

- Observe: If $s[1 \ldots i]$ can be split into two substrings $s[1 \ldots j]$ and $s[j+1 \ldots i]$ s.t.
  1. $s[1 \ldots j]$ can be partitioned into valid words
  2. $s[j+1 \ldots i]$ is a valid word

  then $s[1 \ldots i]$ can also be partitioned.
- Let choose a certain j and fix it
- Easy to know whether $s[j+1 \ldots i]$ is a valid word: $dict(s[j+1 \ldots i])$
- How do we know whether $s[1 \ldots j]$ can be partitioned?
  - *It's a subproblem (substructure property)!*
  - We can assume we know the answer which is $OPT(j)$
  - Moreover, in order to solve the problem that whether $s[1 \ldots i]$ is valid, we rely on the **subproblem** that whether $s[1 \ldots j]$ is valid ($j < i$) (large problem relies on smaller problems)
- But how do we determine j?
  - *We enumerate all the possibilities!*

The DP design: finalized

Recursive formula of $OPT(i)$:

1. General case ($i > 0$):
   a. For all j ($0 \leq j < i$),
      if there is a j s.t $OPT(j) \wedge dict(s[j+1 \ldots i]) =$ True:
      $$OPT(i) = \text{True}$$
   b. Otherwise: $OPT(i) =$ False
2. Base case ($i = 0$): $OPT(0) =$ True

More formal recursive formula

$$OPT(i) = \begin{cases} \big(OPT(0) \wedge dict(s[1 \dots i])\big) \vee \\ \big(OPT(1) \wedge dict(s[2 \dots i])\big) \vee \cdots \vee & i > 0 \\ \big(OPT(i-1) \wedge dict(s[i \dots i])\big) & \\ \\ \textsc{True} & i = 0 \end{cases}$$

Since $OPT(i)$ relies on those $OPT(j)$ with $j < i$, we will start computing OPT function with $i = 1$ and increase i

- We've figured out a valid **evaluation order** for the subproblems which is computable

The Algorithm

$S[0 \ldots n]$ records the separation point so that we know how to separate the string into words

1. Initialize a table $T[0 \ldots n]$ and $S[0 \ldots n]$; $T[0] = True$
2. **for** $i = 1$ **to** n **do**
   $T[i] = False$;
   for $j = 0$ **to** $i - 1$ **do**
       if $T[j] \wedge dict(s[j+1 \ldots i])$ **then**
           $T[i] = True$; $S[i] = j$;
           break;

Figure 5: Text Segmentation Algorithm

Running Time

The running time is clearly $O(n^2)$

0-1 Knapsack problem

0-1 Knapsack problem

Given n objects and a "knapsack" with an integer capacity W s.t.

- Each object i has an integer *weight* $w_i > 0$ and a *profit* $v_i > 0$.

Goal: Find a subset of the objects s.t. the sum of weights $\leq W$ and the sum of profits is maximum.

Remark: The difference of the 0-1 knapsack with the previous fractional knapsack is that we are only allowed to put the entire object into the knapsack, or do not put this object into the knapsack at all

0-1 Knapsack problem

Example (from slides for [Kleinberg&Tardos, Algorithm design])

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Optimal solution: Choose $\{3, 4\}$ with profit 40.

0-1 Knapsack problem

Example (from slides for [Kleinberg&Tardos, Algorithm design])

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Optimal solution: Choose $\{3, 4\}$ with profit 40.

Greedy choice: repeatedly add item with maximum *unit profit* v_i/w_i until you cannot fit in any remaining items
Ex: $\{5, 2, 1\}$ achieves only profit 35 $\Rightarrow$ greedy is *not optimal*

Dynamic Programming: First Attempt

Dynamic Programming: First Attempt

Optimal solution function

Let $OPT(i)$ be the maximum sum of profits when putting objects $1, 2, \ldots, i$ into the knapsack

Dynamic Programming: First Attempt

Optimal solution function

Let $OPT(i)$ be the maximum sum of profits when putting objects $1, 2, \ldots, i$ into the knapsack

Try to recursively define $OPT(i)$:

- Case 1: Do not include object i in the knapsack
  - $OPT(i)$ will be best of $\{1, 2, \ldots, i-1\}$

Dynamic Programming: First Attempt

Optimal solution function

Let $OPT(i)$ be the maximum sum of profits when putting objects $1, 2, \ldots, i$ into the knapsack

Try to recursively define $OPT(i)$:

- Case 1: Do not include object i in the knapsack
  - $OPT(i)$ will be best of $\{1, 2, \ldots, i-1\}$
- Case 2: Include object i in the knapsack
  - Will have to consider how to put $\{1, 2, \ldots, i-1\}$ into the knapsack now with a *remaining capacity* $W - w_i$

Dynamic Programming: First Attempt

Optimal solution function

Let $OPT(i)$ be the maximum sum of profits when putting objects $1, 2, \ldots, i$ into the knapsack

Try to recursively define $OPT(i)$:

- Case 1: Do not include object i in the knapsack
  - $OPT(i)$ will be best of $\{1, 2, \ldots, i-1\}$
- Case 2: Include object i in the knapsack
  - Will have to consider how to put $\{1, 2, \ldots, i-1\}$ into the knapsack now with a *remaining capacity* $W - w_i$
  - The current definition of OPT does not cover this: *We need to add the capacity of the knapsack as a parameter for* OPT

Dynamic Programming: Solution

Optimal solution function

Let $OPT(i, w)$ be the maximum sum of profits when putting objects $1, 2, \ldots, i$ into a knapsack with capacity w

Dynamic Programming: Solution

Optimal solution function

Let $OPT(i, w)$ be the maximum sum of profits when putting objects $1, 2, \ldots, i$ into a knapsack with capacity w

The algorithm will return $OPT(n, W)$

Dynamic Programming: Solution

Optimal solution function

Let $OPT(i, w)$ be the maximum sum of profits when putting objects $1, 2, \ldots, i$ into a knapsack with capacity w

The algorithm will return $OPT(n, W)$

To recursively define $OPT(i, w)$:

- Case 1: Do not include object i in the knapsack with capacity w
  - We need best of $\{1, 2, \ldots, i-1\}$ with capacity w

Dynamic Programming: Solution

Optimal solution function

Let $OPT(i, w)$ be the maximum sum of profits when putting objects $1, 2, \dots, i$ into a knapsack with capacity w

The algorithm will return $OPT(n, W)$

To recursively define $OPT(i, w)$:

- Case 1: Do not include object i in the knapsack with capacity w
  - We need best of $\{1, 2, \dots, i-1\}$ with capacity w
- Case 2: Include object i in the knapsack with capacity w
  - We need best of $\{1, 2, \dots, i-1\}$ with capacity $w - w_i$

Dynamic Programming: Solution

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max \left\{ \begin{array}{l} OPT(i-1,w), \\ v_i + OPT(i-1, w-w_i) \end{array} \right\} & \text{otherwise} \end{cases}$$

The Knapsack Algorithm

Knapsack $(\{w_1, w_2, \ldots, w_n\}, \{v_1, v_2, \ldots, v_n\}, W)$

1. **for** $w = 0$ **to** W **do** $OPT[0, w] = 0$;
2. **for** $i = 0$ **to** n **do** $OPT[i, 0] = 0$;
3. **for** $i = 1$ **to** n **do**
   - **for** $w = 1$ **to** W **do**
     - **if** $w_i > w$ or $OPT[i-1, w] > v_i + OPT[i-1, w-w_i]$ **then**
       - $OPT[i, w] = OPT[i-1, w]$
     - **else**
       - $OPT[i, w] = v_i + OPT[i-1, w-w_i]$
4. **return** $OPT[n, W]$;

Figure 6: Knapsack Algorithm.

Running Time

The running time is clearly $O(n \cdot W)$

Knapsack algorithm

Example of table filling:

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

(from slides for [Kleinberg&Tardos, Algorithm design])

Derivation of the optimal solution for the example

$$OPT(5, 11) = \max \left\{ \begin{array}{l} OPT(4, 11), \\ v_5 + OPT(4, 11 - w_5) \end{array} \right\}$$

Derivation of the optimal solution for the example

So,

$$OPT(5, 11) = OPT(4, 11)$$

- This means that we do not include 5 into $OBJ(5, 11)$ (the optimal set of objects for $OPT(5, 11)$)
- Equivalently, we have:

$$OBJ(5, 11) = OBJ(4, 11)$$

Derivation of the optimal solution for the example

$$OPT(4, 11) = \max \left\{ \begin{array}{l} OPT(3, 11), \\ v_4 + OPT(3, 11 - w_4) \end{array} \right\}$$

Derivation of the optimal solution for the example

So,

$$OPT(4, 11) = v_4 + OPT(3, 5)$$

- This means that we include 4 into $OBJ(4, 11)$.
- Equivalently, we have:

$$OBJ(4, 11) = \{4\} \cup OBJ(3, 5)$$

Derivation of the optimal solution for the example

$$OPT(3,5) = \max\left\{\begin{array}{l} OPT(2,5), \\ v_3 + OPT(2, 5 - w_3) \end{array}\right\}$$

Derivation of the optimal solution for the example

So,

$$OPT(3,5) = v_3 + OPT(2,0)$$

- This means that we include 3 into $OBJ(3,5)$.
- Equivalently, we have:

$$OBJ(3,5) = \{3\} \cup OBJ(2,0)$$

Derivation of the optimal solution for the example

$OPT(2,0) = 0$ is the base case, and we have $OBJ(2,0) = \varnothing$

Edit Distance

Definition

- For two strings, we want to transform one string into the other using three operations: (1) letter insertion, (2) letter deletion, and (3) letter substitution.
- The *edit distance* between the two strings is the minimum number of operations required to complete the transform.

Example: $FOOD \longrightarrow MONEY$

$$\underline{F}OOD \rightarrow MO\underline{O}D \rightarrow MON\;\_D \rightarrow MONE\underline{D} \rightarrow MONEY$$

Edit Distance (FOOD, MONEY)=4

Edit Distance

Definition

- For two strings, we want to transform one string into the other using three operations: (1) letter insertion, (2) letter deletion, and (3) letter substitution.
- The *edit distance* between the two strings is the minimum number of operations required to complete the transform.

Question:

For any two strings, is there always such a sequence of operations?

Edit Distance

Problem

Given two strings $A[1 \dots m]$ and $B[1 \dots n]$, find the *shortest sequence of edit operations* that transforms A into B

Applications

The problem has important applications in DNA sequencing and search engine

Edit Distance as an Alignment Problem

	c	d		i	i	c
A:	F	O	O			D
B:	M		O	N	E	Y

- We add any number of *blanks* (in the middle or to the ends) to the two strings to make them have equal length.
- Then we align the letters and derive the following operation for each pair of matched letters:
  1. A letter in A matched to a blank in B: **delete** the letter in A
  2. A blank in A matched to a letter in B: **insert** the letter in B
  3. A letter in A matched to a letter in B:
     1. Two letter are *different*: **substitute** the letter in A with the letter in B
     2. Two letter are *same*: do nothing
  4. Two blanks in A, B matched: we typically avoid this as this means nothing

Edit Distance as an Alignment Problem

Remark:

- We have transformed the problem of finding an edit sequence for two strings (which is pretty obscure) into finding an alignment for two strings (which is easy to visualize)
- So, instead of trying to find an optimal edit sequence from A to B, we instead try to find an **optimal alignment** between A and B

Alignment: Optimal Substructure Property

	?			i	c
A:	F	O	O		D
B:	M	O	N	E	Y

- Suppose we have partially aligned the right (black) parts. Under the current partial alignment, what do we do on the unaligned (gray) parts to *make the total alignment optimal*?
  - It's simple. Just take the *optimal* alignment for the gray parts (optimal substructure property).
  - Proof is easy (use cut-and-paste)

Alignment: Optimal Substructure Property

	?			i	c
A:	F	O	O		D
B:	M	O	N	E	Y

- The optimal substructure property implies that suppose we have known how to align the suffixes ('right parts') of the two strings A, B, then the problems boils down to finding the shortest edit sequence for two **prefixes** ('left parts') of A, B
  - Prefixes are shorter than the original strings
  - So, we have that solving a *larger* problem relies on solving *smaller* problems

Alignment: Optimal Substructure Property

	?			i	c
A:	F	O	O		D
B:	M	O	N	E	Y

- The optimal substructure property implies that suppose we have known how to align the suffixes ('right parts') of the two strings A, B, then the problems boils down to finding the shortest edit sequence for two **prefixes** ('left parts') of A, B
  - Prefixes are shorter than the original strings
  - So, we have that solving a *larger* problem relies on solving *smaller* problems

Optimal solution function

- Consider the *prefix* $A[1 \ldots i]$ of A and the *prefix* $B[1 \ldots j]$ of B
- Let $Edit(i, j)$ denote the edit distance between $A[1 \ldots i]$ and $B[1 \ldots j]$

Recursive formula for $Edit(i, j)$

$Edit(i, j)$: edit distance between $A[1 \dots i]$ and $B[1 \dots j]$

- Inspired by the previous example, we shall:
    1. Partially align some 'right' parts (suffixes) of $A[1 \dots i]$ and $B[1 \dots j]$
    2. Take the optimal align for the remaining 'left' parts (prefixes).
- How long should the suffix be?

Recursive formula for $Edit(i, j)$

$Edit(i, j)$: edit distance between $A[1 \dots i]$ and $B[1 \dots j]$

- Inspired by the previous example, we shall:
  1. Partially align some 'right' parts (suffixes) of $A[1 \dots i]$ and $B[1 \dots j]$
  2. Take the optimal align for the remaining 'left' parts (prefixes).
- How long should the suffix be? Answer: *We only take one letter*.

Recursive formula for $Edit(i, j)$: Case 1

$Edit(i, j)$: edit distance between $A[1 \dots i]$ and $B[1 \dots j]$

Case 1: Consider matching the last letter $A[i]$ of $A[1 \dots i]$ with a blank inserted at the end of $B[1 \dots j]$ (producing an operation of deleting $A[i]$)

Example: For the previous two strings where $i = m$, $j = n$:

	?	d
A:	F O O	D
B:	M O N E Y	

Shortest edit sequence in this case:

$$Edit(i - 1, j) + 1$$

Recursive formula for $Edit(i, j)$: Case 2

$Edit(i, j)$: edit distance between $A[1 \ldots i]$ and $B[1 \ldots j]$

Case 2: Consider matching the last letter $B[j]$ of $B[1 \ldots j]$ with a blank inserted at the end of $A[1 \ldots i]$ (producing an operation of inserting $B[j]$)

Example: For the previous two strings where $i = m$, $j = n$:

	?				i
A:	F	O	O	D	
B:	M	O	N	E	Y

Shortest edit sequence in this case:

$$Edit(i, j-1) + 1$$

Recursive formula for $Edit(i, j)$: Case 3

$Edit(i, j)$: edit distance between $A[1 \dots i]$ and $B[1 \dots j]$

Case 3: Consider matching the last letter $A[i]$ of $A[1 \dots i]$ with the last letter $B[j]$ of $B[1 \dots j]$ (producing an operation of substituting $A[i]$ with $B[j]$ if $A[i] \neq B[j]$)

Example: For the previous two strings where $i = m$, $j = n$:

	?				c
A:	F	O	O		D
B:	M	O	N	E	Y

Shortest edit sequence in this case:

$$Edit(i-1, j-1) + 0/1$$

Recursive formula for $Edit(i, j)$

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i-1,j)+1 \\ Edit(i,j-1)+1 \\ Edit(i-1,j-1)+\mathrm{diff}(i,j) \end{array} \right\} & \text{otherwise} \end{cases}$$

where $\mathrm{diff}(i,j) = 1$ if $A[i] \neq B[j]$ and 0 otherwise.

The algorithm returns $Edit(m, n)$ as answer

Reason for only needing the 3 cases (advanced material)

- We need to show that the previous 3 cases cover all possibilities for the alignment of $A[1 \ldots i]$ and $B[1 \ldots j]$
- For this, we show that, if an alignment is not Case 1 and Case 3, then it must be Case 2.
- The alignment does not fall in Case 1 means that $A[i]$ is not matched with a blank inserted at the end of $B[1 \ldots j]$
- The alignment does not fall in Case 3 means that $A[i]$ is also not matched with the last letter of $B[1 \ldots j]$
- This means that $A[i]$ has to be matched with some letter before $B[j]$ or some blank inserted before $B[j]$
- Either way, since $A[i]$ is the last letter of $A[1 \ldots i]$, $B[j]$ has to be matched with a blank inserted at the end of $A[1 \ldots i]$ (Case 2)

The Edit Distance Algorithm

Edit-Distance (A, B)

1. **for** $j = 0$ **to** n **do** $Edit[0, j] = j$;
2. **for** $i = 1$ **to** m **do** $Edit[i, 0] = i$;
3. **for** $i = 1$ **to** m **do**
   3.1. **for** $j = 1$ **to** n **do**
      3.1.1. $Edit[i, j] =$
      $\min\{1 + Edit(i-1, j), 1 + Edit(i, j-1), \mathrm{diff}(i, j) + Edit(i-1, j-1)\}$;
4. **return** $Edit[m, n]$;

Figure 7: Edit Distance Algorithm.

Running Time

The running time is clearly $O(mn)$

Matrix-chain multiplication

Given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of matrices, and we wish to compute the product $A_1 A_2 \cdots A_n$.

Background:

- Two matrices A and B can be multiplied iff A has dimension $p \times q$ and B has dimension $q \times r$, i.e., the number of columns of A equals the number of rows of B
- Multiplying A and B has a *cost* $p \cdot q \cdot r$, which is the number of scalar multiplications/summations

Matrix-chain multiplication

Background (continued):

- Matrix multiplication is *associative*: different parenthesizations (orders for which two matrices to multiply first) yield the same product.
  - Different ways to multiply four matrices:

$$1 : A_1(A_2(A_3A_4))$$

$$2 : A_1((A_2A_3)A_4)$$

$$3 : (A_1A_2)(A_3A_4)$$

$$4 : (A_1(A_2A_3))A_4$$

$$5 : ((A_1A_2)A_3)A_4$$

Matrix-chain multiplication

Background (continued):

- Assume we multiply two matrices each time
- Different ways of multiplying the matrices can have a dramatic impact on the cost:
  - E.g., for three matrices $\langle A_1, A_2, A_3 \rangle$, with dimensions

$$10 \times 100,\ 100 \times 5, \text{ and } 5 \times 50$$

  - Cost of $(A_1A_2)A_3$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$
  - Cost of $A_1(A_2A_3)$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$

Problem definition

Matrix-chain multiplication

Given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of matrices, where each A_i has dimension $p_{i-1} \times p_i$, we want to find an way of multiplying $A_1 A_2 \cdots A_n$ with the minimum cost.

Notice:

- The input to the algorithm is a sequence of numbers: $p_0, p_1, \ldots, p_n$ encoding the dimensions of the n matrices
- We are not actually multiplying the matrices. Our goal is only to *determine an order* for the multiplication that has the lowest cost (remember that each time we multiply only two matrices).
- Typically, the time invested in determining this optimal order can be greatly less than the time can we can save compared to an arbitrary multiplication

Brute-force?

Let $P(n)$ denote the different ways of multiplying n matrices, then

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{otherwise} \end{cases}$$

$P(n) \in \Omega(2^n)$: Brute-force doesn't work

The dynamic programming ingredients

- To get the minimum cost for multiplying the sequence, we first make a choice by splitting the sequence into two parts (without caring about how to choose k)

$$\langle A_1, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_n \rangle$$

- We want to figure out a way to multiply the two parts first, and then multiply the products of the two
- Clearly, under this choice, the optimal cost for the multiplication is:

optimal cost for multiplying the left part +
optimal cost for multiplying the right part +
cost for multiplying the products of the two parts

The dynamic programming ingredients

- To get the minimum cost for multiplying the sequence, we first make a choice by splitting the sequence into two parts (without caring about how to choose k)

$$\langle A_1, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_n \rangle$$

- We want to figure out a way to multiply the two parts first, and then multiply the products of the two
- Clearly, under this choice, the optimal cost for the multiplication is:

optimal cost for multiplying the left part +
optimal cost for multiplying the right part +
cost for multiplying the products of the two parts

- This is the *Optimal Substructure Property*! (Proof by cut-and-paste)

Optimal solution function

- Previously, the two subsequences

$$\langle A_1, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_n \rangle$$

  are indeed **subproblems** (combining solutions to the two subproblems produces solution to the original problem)
- Notice that neither the start nor the end of the subsequences are fixed

Optimal solution function

- Previously, the two subsequences

$$\langle A_1, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_n \rangle$$

  are indeed **subproblems** (combining solutions to the two subproblems produces solution to the original problem)
- Notice that neither the start nor the end of the subsequences are fixed
- General form of our subproblem: find the minimum cost for multiplying matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$

Optimal solution function

- Previously, the two subsequences

$$\langle A_1, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_n \rangle$$

  are indeed **subproblems** (combining solutions to the two subproblems produces solution to the original problem)
- Notice that neither the start nor the end of the subsequences are fixed
- General form of our subproblem: find the minimum cost for multiplying matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$

Let $OPT(i,j)$ be the minimum cost for multiplying matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

Recursive formula for $OPT(i, j)$

- To get the optimal way of multiplying $A_i A_{i+1} \cdots A_j$, as previous, we first split $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ into two parts (without caring about how to choose k)

$$\langle A_i, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_j \rangle$$

Recursive formula for $OPT(i, j)$

- To get the optimal way of multiplying $A_i A_{i+1} \cdots A_j$, as previous, we first split $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ into two parts (without caring about how to choose k)

$$\langle A_i, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_j \rangle$$

- Clearly, under this choice, the optimal cost for the multiplication is:

optimal cost for multiplying the left part +
optimal cost for multiplying the right part +
cost for multiplying the products of the two parts

Recursive formula for $OPT(i, j)$

- To get the optimal way of multiplying $A_i A_{i+1} \cdots A_j$, as previous, we first split $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ into two parts (without caring about how to choose k)

$$\langle A_i, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_j \rangle$$

- Clearly, under this choice, the optimal cost for the multiplication is:

optimal cost for multiplying the left part $+$
optimal cost for multiplying the right part $+$
cost for multiplying the products of the two parts

which is $OPT(i, k) + OPT(k, j) + p_{i-1} p_k p_j$

Recursive formula for $OPT(i, j)$

- To get the optimal way of multiplying $A_i A_{i+1} \cdots A_j$, as previous, we first split $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ into two parts (without caring about how to choose k)

$$\langle A_i, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_j \rangle$$

- Clearly, under this choice, the optimal cost for the multiplication is:

  optimal cost for multiplying the left part +
  optimal cost for multiplying the right part +
  cost for multiplying the products of the two parts

  which is $OPT(i, k) + OPT(k, j) + p_{i-1} p_k p_j$
- How do we determine the split position k?

Recursive formula for $OPT(i, j)$

- To get the optimal way of multiplying $A_i A_{i+1} \cdots A_j$, as previous, we first split $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ into two parts (without caring about how to choose k)

$$\langle A_i, A_2, \ldots, A_k \rangle \quad \text{and} \quad \langle A_{k+1}, A_{k+2}, \ldots, A_j \rangle$$

- Clearly, under this choice, the optimal cost for the multiplication is:

  optimal cost for multiplying the left part +
  optimal cost for multiplying the right part +
  cost for multiplying the products of the two parts

  which is $OPT(i, k) + OPT(k, j) + p_{i-1} p_k p_j$
- How do we determine the split position k?
  We enumerate all the possibilities!

Recursive formula for $OPT(i, j)$

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{OPT(i,k) + OPT(k+1,j) + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Algorithm returns $OPT(1, n)$

Recursive formula for $OPT(i, j)$

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{OPT(i,k) + OPT(k+1,j) + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Algorithm returns $OPT(1, n)$

How do we fill out the table?

- $OPT(i, j)$ relies on those $OPT(i', j')$ with $i' > i$ and $j' < j$
- We cannot start from the minimum i and j and increase
- But we also cannot start with the max i $(= n)$ and min j $(= 1)$ because $OPT(n, 1)$ doesn't make sense

Recursive formula for $OPT(i, j)$

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{OPT(i,k) + OPT(k+1,j) + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Algorithm returns $OPT(1, n)$

How do we fill out the table?

- $OPT(i, j)$ relies on those $OPT(i', j')$ with $i' > i$ and $j' < j$
- We cannot start from the minimum i and j and increase
- But we also cannot start with the max i $(= n)$ and min j $(= 1)$ because $OPT(n, 1)$ doesn't make sense
- Observation: $OPT(i, j)$ relies on those $OPT(i', j')$ whose *length* is less (Again, bigger subproblems rely on smaller subproblems)
- So we start from the ones with the minimum length and increase

Matrix-chain multiplication algorithm

Matrix-chain-multiplication $(p_0, p_1, \ldots, p_n)$

1. **for** $i = 1$ **to** n **do** $OPT[i,i] = 0$;
2. **for** $\ell = 2$ **to** n **do**
    - 2.1. **for** $i = 1$ **to** $n - \ell + 1$ **do**
        - 2.1.1. $j = i + \ell - 1$;
        - 2.1.2. $OPT[i,j] = \infty$;
        - 2.1.3. **for** $k = i$ **to** $j - 1$ **do**
          $c = OPT[i,k] + OPT[k+1,j] + p_{i-1}p_k p_j$;
          if $c < OPT[i,j]$ **then** $OPT[i,j] = c$;
3. **return** $OPT[1,n]$;

Figure 8: Matrix-chain multiplication algorithm

Running Time

The running time is $O(n^3)$