CS624 - Analysis of Algorithms

Dynamic Programming

October 21, 2024

Problem: Making Change

Task: Pay for a cup of coffee that costs 63 cents.

- You must give exact change.
- You have an unlimited number of coins of the following denominations: 1 cent, 5 cents, 10 cents, and 25 cents.


1 cent


5 cents


10 cents


25 cents

Questions:

- Is there any solution?
- Can you find a solution, any solution?
- Can you find a solution that minimizes the number of coins?

Greedy Thinking

The “greedy” approach:

- While the debt is at least 25 cents, give a quarter.
- Then while the debt is at least 10 cents, give a dime.
- Then while the debt is at least 5 cents, give a nickel.
- Then while the debt is at least 1 cent, give a penny.

Example

For 63 cents, we give $2(25) + 1(10) + 0(5) + 3(1) = 63$,
using $2 + 1 + 0 + 3 = 6$ coins.

Greedy Algorithms

A greedy person grabs everything they can as soon as possible.

Similarly, a **greedy algorithm** makes locally optimized decisions that appear to be the best thing to do at each step.

Sometimes, a greedy approach cannot solve the problem.
We must prove that a greedy algorithm does not miss solutions.

Change Making

Does the greedy method always work for change-making?

- If we use US coinage: coin denominations $\{1, 5, 10, 25\}$ — yes.
- If we only have quarters and dimes ($\{10, 25\}$) — no.
  - Some problems are just *unsolvable*.
    There's no way to make change for 63 cents.
  - The greedy approach sometimes *misses solutions*.
    For example, make change for 30 cents: $1(25) +$ ...stuck...,
    even though $3(10)$ is a solution.
- Even with $\{1, 10, 25\}$, the greedy solution can be *suboptimal*:
  For example, for 30 cents, it says $1(25) + 5(1)$, but $3(10)$ is better.

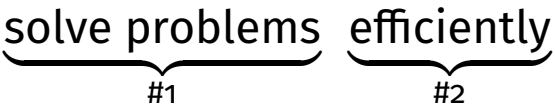
Greedy Algorithms

Lessons:

- Greedy algorithms are popular, because they are (generally) simple and fast.
- But the greedy approach does not always solve the problem. It might produce a sub-optimal solution, or it might miss a solution completely!
- We will revisit greedy algorithms later in the course, but for now, **don't be greedy!**

That is, don't get trapped into short-sighted, greedy thinking.

Reorder Your Priorities



Recursive Strategy

Idea: divide and conquer

- Break each non-trivial problem into two subproblems.
- There are lots of ways (“places”) to divide the problem.

  $63 = 1 + 62 = 2 + 61 = \cdots = 33 + 30 = \cdots = 62 + 1$

  Try all of them, pick the best result.

Recursive Solution

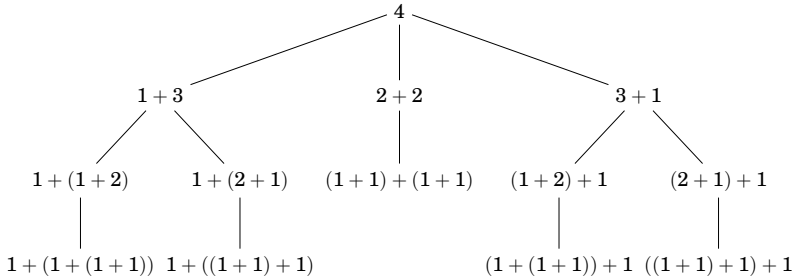
```
// minCoins : int -> int
// Returns the minimum number of coins needed for given amount.
function minCoins(amount) {
  if (amount < 0) return ∞; // ‘‘no solution’’
  else if (amount == 0) return 0;
  else if (amount == 1) return 1;
  else if (amount == 5) return 1;
  else if (amount == 10) return 1;
  else if (amount == 25) return 1;
  else return min{ for m from 1 to amount-1 } (
    minCoins(m) + minCoins(amount - m)
  );
}
```

Analysis

Correctness: “obviously” correct

Running time: incredibly bad, right?

Consider minCoins(4).



All of these correspond to the *task solution* $1 + 1 + 1 + 1$.

Recursive Strategy: A Better Idea

What does an optimal solution for 63 cents look like?

Any solution to the *task* is a list of coins.

An optimal solution for 63 must be one of the following:

- one penny (1 cent) + an optimal solution for 62
- one nickel (5 cents) + an optimal solution for 58
- one dime (10 cents) + an optimal solution for 53
- one quarter (25 cents) + an optimal solution for 38

This is called the optimal substructure property. (Proof?)

So we can calculate those candidates and then pick the minimum-length one.

Recursive Strategy

Refinement to divide and conquer:

- Break each non-trivial problem into
  - one piece of the task solution — the first (next) coin given
  - one subproblem — how to handle leftover amount
- Only 4 choices of first piece of solution!
  (In general, n choices for n different coin denominations.)

Recursive Solution

```
// minCoins : int -> int
// Returns the minimum number of coins needed for given amount.
function minCoins(amount) {
  if (amount < 0) return ∞; // ‘‘no solution’’
  else if (amount == 0) return 0;
  else return min(
    1 + minCoins(amount - 25), // try a quarter
    1 + minCoins(amount - 10), // try a dime
    1 + minCoins(amount - 5),  // try a nickel
    1 + minCoins(amount - 1)   // try a penny
  );
}
```

Recursive Solution

```
// makeChange : int -> [int]
// Returns a minimum-length list of coins summing to amount.
function makeChange(amount) {
  if (amount < 0) return ∞;      // ‘‘no solution’’
  else if (amount == 0) return [];
  else return shortest(
    [25] ++ makeChange(amount - 25), // try a quarter
    [10] ++ makeChange(amount - 10), // try a dime
    [5]  ++ makeChange(amount - 5),  // try a nickel
    [1]  ++ makeChange(amount - 1)   // try a penny
  );
}
```

Recursive Solution, Generalized

```
// minCoins : int [int] -> int
// Returns the minimum number of coins needed for given amount.
function minCoins(amount, denominations) {
  if (amount < 0) return ∞; // ‘‘no solution’’
  else if (amount == 0) return 0;
  else return min {for d in denominations } (
    1 + minCoins(amount - d, denominations)
  );
}
```

Recursive Solution, Generalized

```
// makeChange : int [int] -> [int]
// Returns a minimum-length list of coins summing to amount.
function makeChange(amount, denominations) {
  if (cents < 0) return ∞;      // ‘‘no solution’’
  else if (cents == 0) return [];
  else return shortest { for d in denominations } (
    [d] ++ makeChange(amount - d, denominations)
  );
}
```

Analysis

Correctness: still “obvious”

Running time: still lots of recursive calls (branch factor of 4!)

The next refinement:

- Insight: We keep encountering the same subproblems.
- Save (**memoize**) the result so we only compute it once.

This approach is what we call **dynamic programming**.

Then to make change for n with k different denominations of coins, the running time is $O(nk)$.

Dynamic Programming, Top Down

```
// minCoins : int [int] -> void
// Returns the minimum number of coins needed for given amount.
function minCoins(amount, denominations) {
  return minCoinsTD(amount, denominations, new array[amount]);
}

// minCoinsInner : int [int] [int] -> int
function minCoinsTD(amount, denominations, table) {
  if table[amount] is undefined {
    if (amount == 0) table[amount] = 0;
    else table[amount] = min
        { for d in denominations where d ≤ amount } (
      1 + minCoinsTD(amount - d, denominations, table)
    );
  }
  return table[amount];
}
```

Dynamic Programming, Bottom Up

```
// minCoins : int -> int
// Returns the minimum number of coins needed for given amount.
function minCoins(amount) {
  let table = new array[amount];
  table[0] = 0;
  for m = 1 to amount do {
    table[m] = min { for d in denominations where d ≤ m } (
      1 + table[m - d]
    );
  }
  return table[amount];
}
```

Dynamic Programming

Dynamic programming (DP) is an algorithm design technique for **optimization problems**—generally, minimizing or maximizing some quantity with respect to some constraint.

- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not disjoint; they may share subsubproblems. (That is, they may "overlap".)

  (But subproblems are still self-contained. There's no hidden dependence between sibling subproblems.)
- DP *correctness* relies on optimal substructure property.
- DP *efficiency* relies on memoization of overlapping subproblems.

Self-Contained Subproblems

Subproblems must be independent, even though they may overlap.

For example, the change-making problem assumes I have an **unlimited supply** of each denomination of coin.

A subproblem is identified simply by the *amount of change to make*.

If I have a **limited supply** of each denomination of coin, the previous decomposition of the problem is no longer valid, because I might reach the same amount through paths that use up different portions of my coin supply.

Instead, now a subproblem is identified by the *amount of change to make* together with *the remaining supply of coins*.

Solving a Problem with Dynamic Programming

Let $denoms \subset \mathbb{N}$ be fixed. Then

$$mincoins(m) = \begin{cases} \infty & \text{if } m < 0 \\ 0 & \text{if } m = 0 \\ \min_{d \in denoms}(1 + mincoins(m - d)) & \text{if } m > 0 \end{cases}$$

Solving a Problem with Dynamic Programming

Let $denoms \subset \mathbb{N}$ be fixed. Then

$$mincoins(m) = \begin{cases} \infty & \text{if } m < 0 \\ 0 & \text{if } m = 0 \\ \min_{d \in denoms}(1 + mincoins(m - d)) & \text{if } m > 0 \end{cases}$$

That is the solution, effectively.

There is little need to write out the algorithm.

- The function arguments determine the subproblem labeling.
- The equations determine the recursion structure and base cases.
- Memoization (bottom-up or top-down) can be added mechanically.

But you still must show optimal substructure and analyze the running time given "de-duplicated" overlapping subproblems.

More Examples of Dynamic Programming

- longest common subsequence (LCS)
- optimal binary search tree
- chained matrix multiplications

Subsequences

Definition (Subsequence)

A **subsequence** of a sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence $B = \langle b_1, b_2, \ldots, b_m \rangle$ (with $m \leq n$) such that

- each b_i is an element of A, and
- if $i < j$, then b_i occurs before b_j in A

Note: The elements of B might not be consecutive elements of A.

Example

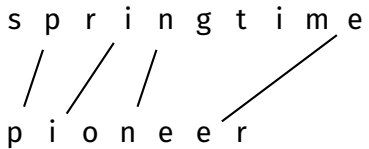
- "axdy" is a subsequence of "baxefdoym"
- "abba" is a subsequence of "abracadabra"

Longest Common Subsequence (LCS)

Longest Common Subsequence (LCS) Problem

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ (note that the sequences may have different lengths), find a subsequence common to both whose length is longest.

Example: LCS



- This is part of a class of what are called *alignment problems*, which are extremely important in biology.
- It can help us to compare genome sequences to deduce quite accurately how closely related different organisms are, and to infer the real “tree of life”.
- Trees showing the evolutionary development of classes of organisms are called “phylogenetic trees”.
- A lot of this kind of comparison amounts to finding common subsequences.

LCS: Naive approach

LCS: Inputs are $X = \langle x_1, \ldots, x_m \rangle$, $Y = \langle y_1, \ldots, y_n \rangle$.

Naive approach: List all the subsequences of X and check each to see if it is a subsequence of Y, and pick the longest one that is.

Analysis:

- There are 2^m subsequences of X.
- To check to see if a subsequence of X is also a subsequence of Y will take time $O(n)$. (Is this obvious?)
- The cost of this naive method is $O(n2^m)$.
- That's pretty awful, since the strings that we are concerned with in biology have hundreds or thousands of elements *at least*.

LCS: Dynamic Programming

Recipe for applying Dynamic Programming:

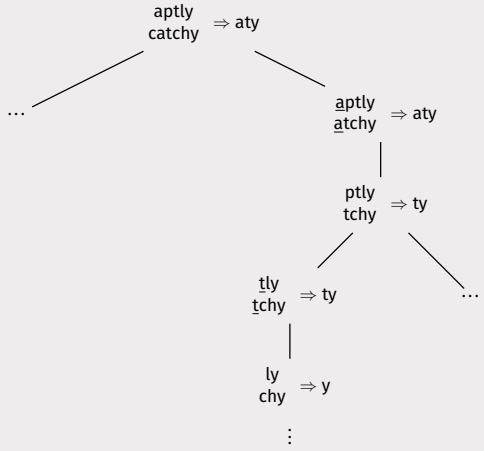
- Determine the optimal substructure property.
- Use that to formulate a recursive solution.
- Memoize the recursive solution.

What is the optimal substructure property for LCS? That is,

- What are a problem's *sub-problems*?
  What stays the same, what varies?
- Even assuming we found optimal solutions to the sub-problems, how does that help us optimally solve the original problem?

Example: LCS

Example (LCS of “aptly” and “catchy”)



LCS: DP Strategy

Strategy: subproblem = pair of suffixes of original strings

- If both strings have common first letter, take it, one subproblem (rest of both strings).
- Otherwise, discard first letter of one of the strings.
  - Cannot discard from both, might lose an optimal solution!
  - Don't know a priori which to discard from.
  - Try both subproblems, pick best result.

In fact, prefixes work as well as suffixes.
Just compare *last* letters instead, discard from end.

Prefixes

Definition

Let $X = \langle x_1, \ldots, x_m \rangle$ be a sequence.

The **prefix** of length $k \leq m$ of X is $X_k = \langle x_1, \ldots, x_k \rangle$.

LCS: Optimal Substructure

LCS: Input is two strings, with possibly different lengths:
$X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$

Theorem

Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. *If $x_m = y_n$, then $z_k = x_m = y_n$,*
   and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. *If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.*
3. *If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}.*

LCS: Proof of Optimal Substructure

Proof.

Let X, Y be sequences, and suppose Z is an LCS for X and Y.
Case analysis:

1. Case $x_m = y_n$: Then $z_k = x_m = y_n$. Suppose for sake of contradiction it isn't. Then Z must be a common subsequence of X_{m-1} and Y_{n-1}. But then $Z' = Z \,||\, \langle x_m \rangle$ would also be a common subsequence of X and Y, and longer than Z, which contradicts the premise that Z is an LCS.

   Moreover, Z_{k-1} is an LCS for X_{m-1} and Y_{n-1}. If it were not, we could again create a better common subsequence than Z, contradicting the premise.

2. Case $z_k \neq x_m$: Then Z must be a subsequence of X_{m-1}, and so it is a common subsequence of X_{m-1} and Y. If there were a longer one, then it would also be a common subsequence of X and Y, which would be a contradiction.

3. Case $z_k \neq y_n$: Similar to case 2.

LCS: Optimal Substructure

Corollary

If $x_m \neq y_n$, then either

- *Z is an LCS of X_{m-1} and Y, or*
- *Z is an LCS of X and Y_{n-1}.*

The optimal substructure theorem assures us that if our recursive function considers only the sub-problems named in the theorem, that is *sufficient* to find an optimal solution.

Similar properties hold for suffixes, but prefixes are slightly tidier.

Recursive Algorithm

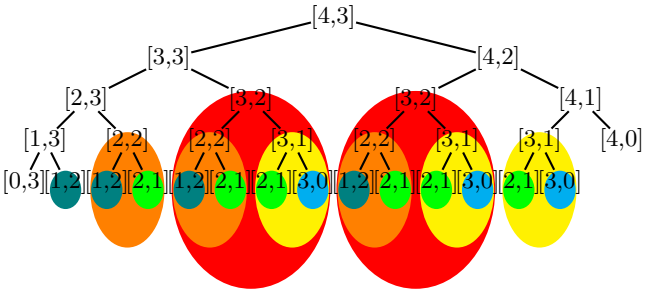
Let $c[i,j]$ be the length of the LCS of X_i and Y_j.

Based on the optimal substructure theorem, we can write the following recurrence:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- The optimal substructure property allows us to write down an elegant recursive algorithm.
- However, the cost is still far too great – we can see that there are $\Omega(2^{\min\{m,n\}})$ nodes in the tree, which is still a killer.

Analysis of Overlapping Subproblems



Analyis of Overlapping Subproblems

- There are only $O(mn)$ distinct nodes, but many nodes appear multiple times.
- We only have to compute each subproblem once, and save the result so we can use it again.
- This is called *memoization*, which refers to the process of saving (i.e., making a "memo") of a intermediate result so that it can be used again without recomputing it.
- Of course the words "memoize" and "memorize" are related etymologically, but they are different words, and you should not mix them up.

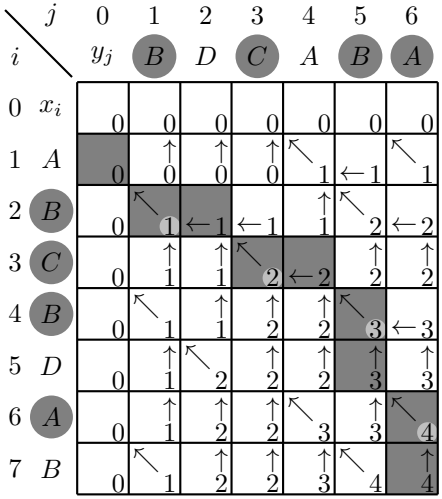
Constructing the Actual LCS

$c[i,j]$ stores the length of the LCS; $b[i,j]$ stores which case that subproblem used

Algorithm 1 LCSLength(X,Y,m,n)

```
1:  for i ← 1...m do
2:      c[i,0] ← 0
3:  end for
4:  for j ← 0...n do
5:      c[0,j] ← 0
6:  end for
7:  for i ← 1...m do
8:      for j ← 1...n do
9:          if x_i == y_j then
10:             c[i,j] ← c[i−1,j−1] + 1; b[i,j] ← "↖"
11:         else if c[i−1,j] ≥ c[i,j−1] then
12:             c[i,j] ← c[i−1,j]; b[i,j] ← "↑"
13:         else
14:             c[i,j] ← c[i,j−1]; b[i,j] ← "←"
15:         end if
16:     end for
17: end for
18: return c and b
```

Constructing the Actual LCS



Just backtrack from $b[m,n]$ following the arrows:

Algorithm 2 PrintLCS(b, X, i, j)

```
1: if i = 0 or j = 0 then
2:     return
3: end if
4: if b[i,j] = "↖" then
5:     PrintLCS(b, X, i − 1, j − 1)
6:     PRINT x_i
7: else if b[i,j] = "↑" then
8:     PrintLCS(b, X, i − 1, j)
9: else
10:    PrintLCS(b, X, i, j − 1)
11: end if
```

What Makes Dynamic Programming Work?

It is important to understand the two properties of this problem that made it possible for use of dynamic programming:

- Optimal substructure: Subproblems are just "smaller versions" of the main problem.

  Finding the LCS of two substrings could be reduced to the problem of finding the LCS of shorter substrings.

  This property enables us to write a recursive algorithm to solve the problem, but this recursion is much too expensive — typically, it has an exponential cost.

- Overlapping subproblems: This is what saves the running time. The same subproblem is encountered many times, so we can just solve each subproblem once and "memoize" the result.

  In the current problem, that memoization cut down the cost from exponential to quadratic, a dramatic improvement.

Optimal Binary Search Tree

Optimal Binary Search Tree Problem

Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i, build a binary search tree (BST) with minimum *expected search cost*.

This is a simplified version of what the book calls the Optimal BST problem:
It assumes that there will never be searches for absent keys.

Example: translation dictionary, where not all words have equal frequency

Example: Optimal BST

Suppose we have 5 keys:

$$k_1 < k_2 < k_3 < k_4 < k_5$$

and suppose the following table shows the probabilities of searching for these different nodes:

i	1	2	3	4	5
p_i	0.25	0.20	0.05	0.20	0.30

Expected Search Cost

Suppose that T is a BST containing all of the k_i keys. Then:

- The cost of looking up a specific key k_i is

$$depth_T(k_i) + 1$$

  For any node x in the tree T, let us say that $depth_T(x)$ is the distance of x from the root of T. (So the root has depth 0.)

- Let $E(T)$ be the expected search cost—the average over all keys $\{k_i\}$ with the distribution described by $\{p_i\}$. Then:

$$\begin{aligned} E(T) &= \sum_{i=1}^{n} (depth_T(k_i) + 1) \cdot p_i \\ &= \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \end{aligned}$$

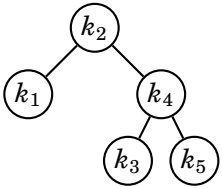
Expected Search Cost

- If we think of the $\{p_i\}$ probabilities as "weights",
  then the total weight of the tree is $w(1,n) = \sum_{i=1}^{n} p_i$.
- So the equation above could also be written like this:

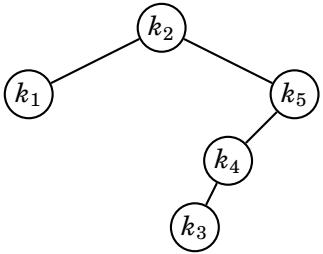
$$E(T) = w(1,n) + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i$$

- Since the $\{p_i\}$ weights are *probabilities*, then $w(1,n) = 1$, but this is not necessary, and it won't be true for subproblems.

Example: Expected Search Costs



key	prob.	depth+1	cost
k_1	0.25	2	0.50
k_2	0.20	1	0.20
k_3	0.05	3	0.15
k_4	0.20	2	0.40
k_5	0.30	3	0.90
Total			2.15



key	prob.	depth+1	cost
k_1	0.25	2	0.50
k_2	0.20	1	0.20
k_3	0.05	4	0.20
k_4	0.20	3	0.60
k_5	0.30	2	0.60
Total			2.10

So “more balanced” is not necessarily optimal for this problem.

Optimal Substructure

The number of binary trees on n nodes is

$$\frac{1}{n+1}\binom{2n}{n} = \frac{4^n}{\sqrt{\pi}n^{3/2}}(1+O(1/n))$$

So exhaustive search is not viable.

What is the optimal substructure property for this problem?

- What are the original problem's *sub-problems*?
- Even assuming we found optimal solutions to the sub-problems, how does that help us optimally solve the original problem?

Brainstorming

- Input is a set $K = \{k_1, \ldots, k_n\}$, $\{p_i\}$ (treat as function).
- Final output is a BST.
- Most likely, a subproblem solution is also a BST.
  How do we divide and combine?

Brainstorming

- Input is a set $K = \{k_1, \ldots, k_n\}$, $\{p_i\}$ (treat as function).
- Final output is a BST.
- Most likely, a subproblem solution is also a BST.
  How do we divide and combine?
- Possibility: Minimize over last leaf inserted

- Possibility: Minimize over choice of root node

Brainstorming

- Input is a set $K = \{k_1, \ldots, k_n\}$, $\{p_i\}$ (treat as function).
- Final output is a BST.
- Most likely, a subproblem solution is also a BST.
  How do we divide and combine?
- Possibility: Minimize over last leaf inserted
  1. Pick a key k_i, let T_i be solution for $K - \{k_i\}$
  2. For each i, calculate $\mathrm{Insert}(T', k_i)$
  3. Minimize $E(T_i)$ over choice of i
  4. Subproblems identified by *subset* of K, so 2^n total
- Possibility: Minimize over choice of root node

Brainstorming

- Input is a set $K = \{k_1, \ldots, k_n\}$, $\{p_i\}$ (treat as function).
- Final output is a BST.
- Most likely, a subproblem solution is also a BST.
  How do we divide and combine?
- Possibility: Minimize over last leaf inserted
  1. Pick a key k_i, let T_i be solution for $K - \{k_i\}$
  2. For each i, calculate $\mathrm{Insert}(T', k_i)$
  3. Minimize $E(T_i)$ over choice of i
  4. Subproblems identified by *subset* of K, so 2^n total
- Possibility: Minimize over choice of root node
  1. Pick a key k_i to be the root
  2. Let L, R be solutions for subsets of K left/right of k_i
  3. Let T_i be tree with k_i as root, L and R as children
  4. Minimize $E(T_i)$ over choice of i
  5. Subproblems identified by *interval* in $1 \,..\, n$, so $O(n^2)$ total

Brainstorming

- Input is a set $K = \{k_1, \ldots, k_n\}$, $\{p_i\}$ (treat as function).
- Final output is a BST.
- Most likely, a subproblem solution is also a BST.
  How do we divide and combine?
- Possibility: Minimize over last leaf inserted — not promising
  1. Pick a key k_i, let T_i be solution for $K - \{k_i\}$
  2. For each i, calculate $\mathrm{Insert}(T', k_i)$
  3. Minimize $E(T_i)$ over choice of i
  4. Subproblems identified by *subset* of K, so 2^n total
- Possibility: Minimize over choice of root node — let's try this
  1. Pick a key k_i to be the root
  2. Let L, R be solutions for subsets of K left/right of k_i
  3. Let T_i be tree with k_i as root, L and R as children
  4. Minimize $E(T_i)$ over choice of i
  5. Subproblems identified by *interval* in $1 \,..\, n$, so $O(n^2)$ total

Brainstorming

The second approach (minimize over choices of the key at the root) seems more promising.

But we don't know yet if that will actually yield a viable algorithm.

What optimal substructure property would justify this approach?

- We don't have a clever strategy for picking the root key.
- So many trees built from optimal sub-trees will be sub-optimal!
- But an optimal tree can only be built from optimal sub-trees.
- The algorithm still must search over trees built from optimal sub-trees, but at least it must search *only* over such trees.

Optimal Substructure

Theorem (Optimal Substructure for Optimal BST Problem)

If T is an optimal binary search tree and if T' is any subtree of T, then T' is an optimal binary search tree for its keys.

Proof.

By the standard optimal substructure argument:

Suppose for sake of contradiction that T' is not optimal. Then there is a better solution, but you could use that to improve T, which violates the premise that T is optimal. □

Computing the Optimal Solution

- Let $T_{i,j}$ be an optimal BST for keys $\{k_i, \ldots, k_j\}$ ($i \leq j$).
- Let $e[i,j]$ be the expected cost of searching an optimal binary search tree containing the keys $\{k_i, \ldots, k_j\}$.

  That is, $e[i,j]$ is the expected cost of searching the tree $T_{i,j}$.

  Ultimately, we want to compute $e[1,n]$.
- Let $w(i,j) = \sum_{k=i}^{j} p_k$. That is, $w(i,j)$ is the sum of the weights of keys k_i through k_j.

Computing the Optimal Solution

If $T_{i,j}$ has k_r at its root, then $e[i,j]$ has three components:

- The expected cost of searching the left child. This tree is $T_{i,r-1}$, but as the left child, each node it at a depth 1 greater than accounted for by $e[i,r-1]$; we must adjust to compensate.

  The cost is rather $e[i,r-1] + \sum_{k=i}^{r-1} p_k = e[i,r-1] + w(i,r-1)$.
- The cost of searching for the root k_r, which is $p_{k_r} = w(r,r)$.
- The expected cost of searching the right child. This tree is $T_{r+1,j}$, but we must likewise adjust for the increased depth.

  Its cost is $e[r+1,j] + \sum_{k=r+1}^{j} p_k = e[r+1,j] + w(r+1,j)$.

So the total cost is

$$\begin{aligned} e[i,j] &= e[i,r-1] + e[r+1,j] + w(i,r-1) + w(r,r) + w(r+1,j) \\ &= e[i,r-1] + e[r+1,j] + w(i,j) \end{aligned}$$

Simplifying Things a Little

If $T_{i,j}$ has k_r as the root, then

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j)$$

We must find r by minimizing the expect cost over all possible choices:

$$e[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{otherwise} \end{cases}$$

Overlapping Subproblems

Let n be the number of keys in the original problem.

How many distinct subproblems are there?

Each nontrivial subproblem is identified by an interval $i..j$, where $1 \leq i \leq j \leq n$.

There are $O(n^2)$ such intervals.
That's the size of the memo table.

Running Time Analysis

- Precompute $w(i,j)$, store in a $n \times n$ table $w[i,j]$. We have

$$w[i,j] = \begin{cases} 0 & \text{if } i > j \\ w[i,j-1] + p_j & \text{otherwise} \end{cases}$$

  There are $O(n^2)$ values of $w[i,j]$ and each one takes a constant time to compute, so the cost of computing the w array is $O(n^2)$.
- The cost of computing each value of $e[i,j]$ is $O(n)$ and there are $O(n^2)$ such values, so the cost of computing all the values of $e[i,j]$ is $O(n^3)$.
- So the total cost of computing the w array first and then the e array is $O(n^2) + O(n^3) = O(n^3)$

Chain Operations

Chain Operation Problem

Determine the optimal sequence for performing a series of operations. This general class of problems is important in compiler design for code optimization & in databases for query optimization.

Example (Chain Matrix Multiplication)

Given a series of matrices: $A_1 \dots A_n$, we can "parenthesize" this expression however we like, since matrix multiplication is associative but not commutative.

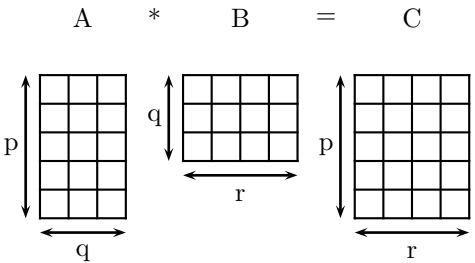
Multiplying a $p \times q$ matrix A by a $q \times r$ matrix B produces a $p \times r$ matrix C. (# of columns of A must be equal to # of rows of B.)

Matrix Multiplications

Suppose A is a $p \times q$ matrix and B is a $q \times r$ matrix.
Then $C = AB$ is a $p \times r$ matrix defined by

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j]$$

Observe that each element of C takes $O(q)$ time to compute, thus the total time to multiply A and B is pqr.



Chain Matrix Multiplication (CMM)

Chain Matrix Multiplication (CMM) Problem

Given a sequence of matrices $A_1, A_2, \ldots A_n$, and dimensions $p_0, p_1 \ldots p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine a multiplication sequence that minimizes the number of operations.

Note: This algorithm does not perform the multiplication, it just figures out the best order in which to perform the multiplication.

Example: Chain Matrix Multiplication

- Consider 3 matrices: A_1 is 5×4, A_2 is 4×6, and A_3 is 6×2.
- Count the number of operations:

$$cost[((A_1A_2)A_3)] = (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180$$
$$cost[(A_1(A_2A_3))] = (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88$$

- Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

CMM: Naive Algorithm

Case: 1 item

- Then there is only one way to parenthesize.

Case: n>1 items

- There are $n-1$ places where we could break the list into two non-empty subproblems.
- When we split just after the k^{th} item, we create two sub-lists to be parenthesized, one with k items and the other with $n-k$ items. Then we consider all ways of parenthesizing these.
- If there are L ways to parenthesize the left sub-list, R ways to parenthesize the right sub-list, then the total number of possibilities is $L \cdot R$.

Enumerate all parenthesizations, choose one with least cost.

Cost of Naive Algorithm

The number of different ways of parenthesizing n items is

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

This is related to Catalan numbers, which in turn are related to the number of different binary trees on n nodes.

Specifically, $P(n) = C(n-1)$.

$$C(n) = \frac{1}{n+1} \cdot \binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

CMM: Dynamic Programming Solution

If we choose to break the original list of n matrices between k and $k+1$, we don't want to enumerate and consider *all* parenthesizations of $[1 \,..\, k]$ and $[k+1 \,..\, n]$.

We want to consider only the *best* parenthesization of $[1 \,..\, k]$ and the *best* parenthesization of $[k+1 \,..\, n]$.

That simplification is only justified if the CMM problem has the optimal substructure property. (It does.)

CMM: Dynamic Programming Solution

Input: matrices $A_1, \ldots, A_n$ with dimensions $p_0, p_1, \ldots, p_n$.

- Let $m[i,j]$ (where $1 \leq i \leq j \leq n$) denote the minimum number of multiplications needed to compute $\prod_{k=i}^{j} A_k$.
- Example: Minimum number of multiplies for $A_3 \cdots A_7$

$$A_1 A_2 \underbrace{A_3 A_4 A_5 A_6 A_7}_{m[3,7]} A_8 A_9$$

The product $A_3 \cdots A_7$ has dimensions $p_2 \times p_7$.

CMM: Dynamic Programming Solution

The optimal cost can be described be as follows:

- $i = j \Rightarrow$ the sequence contains only 1 matrix, so $m[i,j] = 0$
- $i < j \Rightarrow$ we consider, for each k where $i \leq k < j$, the product of $A_i \cdots A_k$ (with dimensions $p_{i-1} \times p_k$) and $A_{k+1} \cdots A_j$ (with dimensions $p_k \times p_j$)

Thus $m[i,j]$ is described by the following recursive rule:

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

Matrix Chain Order

Algorithm 3 MatrixChainOrder(p)

```
1:  n ← length[p] − 1
2:  for i ← 1 to n do
3:      m[i, i] ← 0
4:  end for
5:  for L ← 2 to n do
6:      for i ← 1 to n − L + 1 do
7:          j ← i + L − 1; m[i, j] ← ∞
8:          for k ← i to j − 1 do
9:              q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10:             if q < m[i, j] then
11:                 m[i, j] ← q
12:                 s[i, j] ← k
13:             end if
14:         end for
15:     end for
16: end for
17: return m and s
```

$m[i,j]$ contains cost of optimal multiplication of $A_i \cdots A_j$

$s[i,j]$ is the decision log. It contains the optimal "split point": $(A_i \cdots A_k)(A_{k+1} \cdots A_j)$

Runtime Analysis

There are 3 nested loops and each can iterate at most n times, so the total running time is $\Theta(n^3)$.

Example

We are multiplying the following matrices:

$$A_1\ (5 \times 4) \text{ times } A_2\ (4 \times 6) \text{ times } A_3\ (6 \times 2) \text{ times } A_4\ (2 \times 7)$$

That is, the initial sequence of dimensions is $\langle 5, 4, 6, 2, 7 \rangle$.
The optimal parenthesization is $(A_1(A_2A_3))A_4$.

m[i,j]

$p_0 \to 5$ A_1	0	120	88	158	1
$p_1 \to 4$ A_2		0	48	104	2
$p_2 \to 6$ A_3			0	84	3
$p_3 \to 2$ A_4				0	4
$p_4 \to 7$					

j: 1 2 3 4 — i: 1 2 3

s[i,j]

1	1	3	1
	2	3	2
		3	3

j: 2 3 4 — i: 1 2 3

Final order