CS624 - Analysis of Algorithms

Dynamic Programming

March 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

$\underbrace{\text{solve problems}}_{\#1}\ \underbrace{\text{efficiently}}_{\#2}$

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

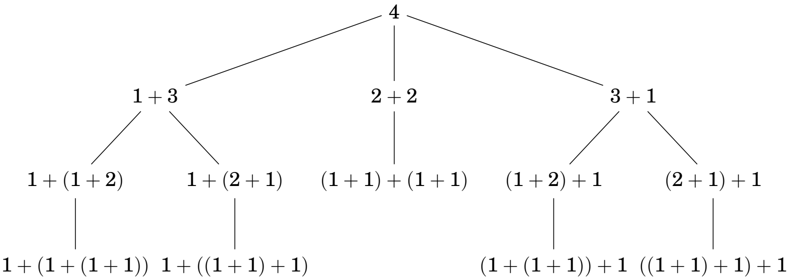
```
// minCoins : int -> int
// Returns the minimum number of coins needed for given amount.
function minCoins(amount) {
  if (amount < 0) return ∞; // ‘‘no solution’’
  else if (amount == 0) return 0;
  else if (amount == 1) return 1;
  else if (amount == 5) return 1;
  else if (amount == 10) return 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 $denominations \subset \mathbb{N}$ be fixed. Then

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

Solving a Problem with Dynamic Programming

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

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

That is the solution.

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 analyse the running time given overlapping subproblems.

Longest Common Subsequence (LCS)

Definition

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

- Each b_i is an element of A.
- If b_i occurs before b_j in B (i.e., if $i < j$) then it also occurs before b_j in A.

- We do *not* assume that the elements of B are consecutive elements of A.
- For example:"axdy" is a subsequence of "baxefdoym"

The "longest common subsequence" problem is simply this:

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

LCS – example

s p r i n g t i m e

p i o n e e r

- 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

- Try the obvious 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.
- 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?)
- Picking the longest one an $O(1)$ job, since we can keep track as we proceed of the longest subsequence that we have found so far.
- So the cost of this 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 – Optimal Substructure

- We have two strings, with possibly different lengths: $X = \{x_1, x_2, \dots, x_m\}$ and $Y = \{y_1, y_2, \dots, y_n\}$
- A *prefix* of a string is an initial segment. So we define for each i less than or equal to the length of the string the prefix of length i: $X = \{x_1, x_2, \dots, x_i\}$ and $Y = \{y_1, y_2, \dots, y_i\}$
- A solution of our problem reflects itself in solutions of prefixes of X and Y.

Theorem

Let $Z = \{z_1, z_2, \dots, z_k\}$ 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 – Optimal Substructure

Proof.

1. By assumption $x_m = y_n$. If z_k does not equal this value, then Z must be a common subsequence of X_{m-1} and Y_{n-1}, and so the sequence $Z' = \{z_1, z_2, \ldots, z_k, x_m\}$ would be a common subsequence of X and Y. But this is a longer common subsequence than Z, and this is a contradiction.
2. If $z_k \neq x_m$, 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. This is really the same as 2.

$\square$

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}.*

- Thus, the LCS problem has what is called the *optimal substructure property*: a solution contains within it the solutions to subproblems – in this case, to subproblems constructed from prefixes of the original data.
- This is one of the two keys to the success of a dynamic programming solution.

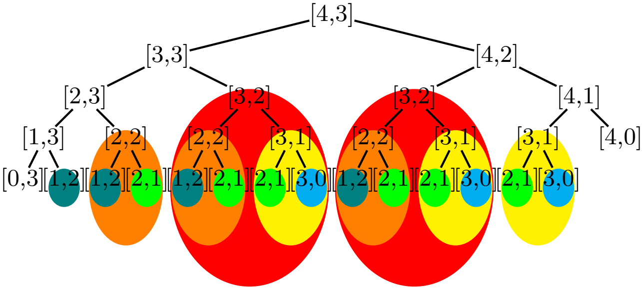
Recursive Algorithm

- 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.

Recursive Algorithm



Overlapping Substructures

- 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.

Another Algorithm

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

for $i \leftarrow 1 \ldots m$ **do**
  $c[i, 0] \leftarrow 0$
end for
for $j \leftarrow 0 \ldots n$ **do**
  $c[0,j] \leftarrow 0$
end for
for $i \leftarrow 1 \ldots m$ **do**
  for $j \leftarrow 1 \ldots n$ **do**
    if $x_i == y_j$ **then**
      $c[i,j] \leftarrow c[i-1,j-1]+1; b[i,j] \leftarrow$ "$\nwarrow$"
    else
      if $c[i-1,j] \geq c[i,j-1]$ **then**
        $c[i,j] \leftarrow c[i-1,j]; b[i,j] \leftarrow$ "$\uparrow$"
      else
        $c[i,j] \leftarrow c[i,j-1]; b[i,j] \leftarrow$ "$\leftarrow$"
      end if
    end if
  end for
end for
return c and b

LCS Table – Example

i \ j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6	A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Constructing the Actual LCS

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

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

if $i = 0$ or $j = 0$ **then**
    return
end if
if $b[i,j] ==$ "$\nwarrow$" **then**
    $PrintLCS(b, X, i-1, j-1)$
    PRINT x_i
else
    if $b[i,j] ==$ "$\uparrow$" **then**
        $PrintLCS(b, X, i-1, j)$
    else
        $PrintLCS(b, X, i, j-1)$
    end if
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 us: 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

- Given sequence $K = k_1 < k_2 < \quad < k_n$ of n sorted keys, with a search probability p_i for each key k_i.
- Want to build a binary search tree (BST) with minimum expected search cost.
- Actual cost = # of items examined.
- For key k_i, $cost = depth_T(k_i) + 1$, where $depth_T(k_i)$ = depth of k_i in BST T .
- Example – dictionary search, where not all words have equal probability to be searched.

Example

- Suppose we have a BST containing 5 words.
- We can create an additional 6 “dummy” nodes to represent searches for words not in the tree, like this (where we have arranged the words in alphabetical order:

$$d_0 \quad k_1 \quad d_1 \quad k_2 \quad d_2 \quad k_3 \quad d_3 \quad k_4 \quad d_4 \quad k_5 \quad d_5$$

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

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Cost of Searching a BST

- p_i is the probability of searching for k_i (the probability of searching for the i^{th} word)
- q_i (for $i \geq 1$) is the probability of searching for d_i (the probability of searching for a word between the i^{th} word and the $(i+1)^{th}$ word in the tree)
- q_0 is the probability of searching for a word before the first word in the tree

Of course we must have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Expected Search Cost of Each Tree

- Assume that the cost of a search is the number of nodes visited in the search.
- Denote the expected search cost for a tree T by $E(T)$.
- 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.)
- Then we have

$$E(T) = \sum_{i=1}^{n} (depth_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1) \cdot q_i$$

- Note that this can also be written as

$$E(T) = \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=0}^{n} depth_T(d_i) \cdot q_i + \left(\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i \right)$$

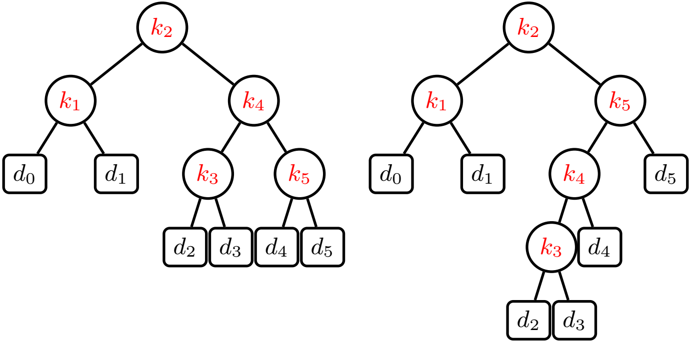
Expected Search Cost of Each Tree

- If we think of the probabilities p_i and q_i as "weights", then the total weight of the tree is $w(1, n) = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i$
- You will see below why we called this $w(1, n)$, and not just w.
- So the equation above could also be written like this:

$$E(T) = w(1, n) + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=0}^{n} depth_T(d_i) \cdot q_i$$

- As it happens, we know that $w(1, n)$ actually equals 1, but we will write some similar identities below in which this is no longer true.

Example – Two Trees with 5 Keys



Example – Expected Search Cost of Left Tree

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

The expected search cost for the other tree is 2.75. So putting the nodes of maximum probability highest is not necessarily the best thing to do.

Optimal Substructure

- The number of binary trees on n nodes is

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

- Certainly exhaustive search is not a useful way of finding the best tree in this problem.
- Substructure naturally involves subtrees.
- Our problem does exhibit optimal substructure in in the following way:
- Since our tree is a BST, any subtree contains a contiguous sequence of keys $\{k_i, \ldots, k_j\}$ and its leaves will be the contiguous set of dummy nodes $\{d_{i-1}, \ldots, d_j\}$.
- Let us denote the optimal binary search tree containing exactly these nodes by $T_{i,j}$.

Optimal Substructure

The optimal substructure property that our problem possesses is this:

Theorem (Optimal substructure for the optimal binary search tree 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 nodes.

Proof.

This is a standard cut-and-paste argument. □

Compute the Optimal Solution

- 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 $e[1, n]$.
- If the optimal binary search tree for this subproblem has k_r as its root then the problem divides into three parts:
  - The expected cost of searching the tree $T_{i,r-1}$ built from the nodes $\{i, \ldots, r-1\}$, adjusted for the fact that this is a subtree of our original tree $T_{i,j}$ and so all the depths should be 1 greater than they are in the subtree.
  - The cost of searching for the root k_r.
  - The expected cost of searching the tree $T_{r+1,j}$ built from the nodes $\{k_{r+1}, \ldots, k_j\}$, with all the depths 1 greater than they are in the subtree.

Compute the Optimal Solution

- r can take any of the values $\{i, \ldots, j\}$.
- If $r = i$ then the first subtree $T_{i,r-1}$ is empty (rather, we let it contain the dummy node d_{i-1} since there is nowhere else to put that node anyway).
- Similarly, if $r = j$ then the second subtree $T_{r+1,j}$ contains the dummy node d_j.
- In other words – the tree $T_{s,s-1}$ built from the nodes $\{k_s, \ldots k_{s-1}\}$ contains the single node d_{s-1} and its expected cost $e[s, s-1]$ will thus be q_s.
- Set $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$
- This is the sum of the probabilities of all the nodes in the tree $T_{i,j}$ built from the nodes $\{k_i, \ldots, k_j\}$.

Compute the Optimal Solution

- The tree $T_{i,r-1}$ has cost $e[i,r-1]$, but as a subtree of $T_{i,r}$, its cost has to be increased by increasing each depth number by 1 – this amounts to adding $w(i,r-1)$.
- The expected cost that the subtree $T_{i,r-1}$ contributes to the expected cost of $T_{i,j}$ is $e[i,r-1] + w(i,r-1)$.
- A similar argument applies to the other subtree $T_{r+1,j}$.
- So we get

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

Simplifying Things a Little

- Note that $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$
- So we have $e[i,j] = w(i,j) + e[i,r-1] + e[r+1,j]$
- We have to take the minimum over all possible choices of r.
- Thus we have

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

- We can use it to compute $e[1,n]$.
- However this algorithm is still exponential in cost.
- We can do better because this problem also exhibits the property of *overlapping subproblems*.
- There are only $O(n^2)$ values $e[i,j]$ with $1 \le i \le n+1$ and $0 \le j \le n$, so we can memoize the.

More Efficient Calculation

- Store pre-computed values in an array $e[1 \ldots n+1, 0 \ldots n]$.
- We can also store the values $w(i,j)$ in a table $w[1 \ldots n+1, 0 \ldots n]$.
- We have
$$w[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ w[i,j-1] + p_j + q_j & \text{otherwise} \end{cases}$$
- There are $O(n^2)$ values of $w[i,j]$ and each one takes a constant time to compute, so the total 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 $O(n^2) + O(n^3) = O(n^3)$

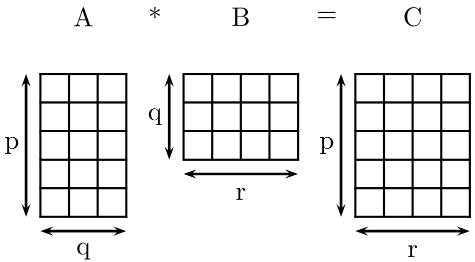
Example – Chain Operations

- Determine the optimal sequence for performing a series of operations (the general class of the problem is important in compiler design for code optimization & in databases for query optimization)
- For example: given a series of matrices: $A_1 \ldots A_n$, we can "parenthesize" this expression however we like, since matrix multiplication is associative **(but not commutative)**
- Multiply a pxq matrix by a qxr matrix B, the result will be a pxr matrix C. (# of columns of A must be equal to # of rows of B.)

Matrix Multiplications

for $1 \le i \le p$ and $1 \le j \le r$, $C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j]$

Observe that there are pr total entries in C and each takes O(q) time to compute, thus the total time to multiply 2 matrices is pqr.



Chain Matrix Multiplication (CMM)

- 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 $p_{i-1} x p_i$, determine multiplication sequence that minimizes the number of operations.
- This algorithm does not perform the multiplication, it just figures out the best order in which to perform the multiplication.

CMM – Example

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

$$Mult[((A_1A_2)A_3)] = (5x4x6) + (5x6x2) = 180$$

$$Mult[(A_1(A_2A_3))] = (4x6x2) + (5x4x2) = 88$$

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

CMM – Naive Algorithm

- If we have just 1 item, then there is only one way to parenthesize.
- If we have n items, then there are n-1 places where you could break the list with the outermost pair of parentheses, namely just after the first item, just after the 2^{nd} item, etc. and just after the $(n-1)^{th}$ item.
- 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 possibilities is L*R.

Cost of Naive Algorithm

- The number of different ways of parenthesizing n items is

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

- Specifically $P(n) = C(n-1)$.
- $C(n) = (1/(n+1)) * \binom{2n}{n} = \Omega(4^n/n^{3/2})$

DP Solution (I)

- Let $A_{i...j}$ be the product of matrices i through j. $A_{i...j}$ is a $p_{i-1}xpj$ matrix.
- At the highest level, we are multiplying two matrices together. That is, for any k, $1 \leq k \leq n-1$, $A_{1...n} = (A_{1...k})(A_{k+1...n})$
- The problem of determining the optimal sequence of multiplication is broken up into 2 parts:
  - Q: How do we decide where to split the chain (what k)?
  - A: Consider all possible values of k.
  - Q: How do we parenthesize the subchains $A_{1...k}\&Ak+1...n$?
  - A: Solve by recursively applying the same scheme.
- NOTE: this problem satisfies the “principle of optimality”
- Next, we store the solutions to the sub-problems in a table and build the table in a bottom-up manner.

DP Solution

- For $1 \leq i \leq j \leq n$, let m[i,j] denote the minimum number of multiplications needed to compute $A_{i...j}$.
- Example: Minimum number of multiplies for $A_{3...7}$

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

In terms of p_i , the product $A_{3...7}$ has dimensions $p_2 x p_7$.

DP Solution

- The optimal cost can be described be as follows:
- $i = j \Rightarrow$ the sequence contains only 1 matrix, so m[i, j] = 0.
- $i < j \Rightarrow$ This can be split by considering each k, $i \leq k < j$, as $A_{i...k}(p_{i-1}xp_k)$ times $A_{k+1..j}(p_kxp_j)$.
- This suggests the following recursive rule for computing m[i, j]:

$$m[i,i] = 0$$
$$m[i,j] = \min_{i \leq k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) \forall i < j$$

Computing m[i,j]

For a specific K:

$$\begin{aligned} & (A_i \ldots A_k)(A_{k+1} \ldots A_j) \\ =& A_{i\ldots k}(A_{k+1} \ldots A_j) && \text{(m[i, k] mults)} \\ =& A_{i\ldots k}A_{k+1\ldots j} && \text{(m[k+1, j] mults)} \\ =& A_{i\ldots j} && (p_{i-1}p_kp_j \text{ mults)} \end{aligned}$$

For solution, evaluate for all k and take minimum.

$$m[i,j] = \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j)$$

Matrix Chain Order

Algorithm 3 MatrixChainOrder(p)

```
n ← length[p] − 1
for i ← 1...n// initialization: O(n) time do
    m[i,i] ← 0
    for L ← 2...n// L = length of sub-chain do
        for i ← 1...n − L + 1 do
            j ← i + L − 1, m[i,j] ← ∞
            for k ← itoj − 1 do
                q ← m[i,k] + m[k+1,j] + p_{i−1} p_k p_j
                if q < m[i,j] then
                    m[i,j] ← q, s[i,j] ← k
                end if
            end for
        end for
    end for
end for
return m and s
```

Runtime Analysis

- The array s[i, j] is used to extract the actual sequence (see next).
- There are 3 nested loops and each can iterate at most n times, so the total running time is $\Theta(n^3)$.

Extracting Optimal Sequence

- Leave a split marker indicating where the best split is (i.e. the value of k leading to minimum values of m[i, j]).
- We maintain a parallel array s[i, j] in which we store the value of k providing the optimal split.
- If s[i, j] = k, the best way to multiply the sub-chain $Ai\ldots j$ is to first multiply the sub-chain $A_{i\ldots k}$ and then the sub-chain $A_{k+1\ldots j}$, and finally multiply them together.
- Intuitively s[i, j] tells us what multiplication to perform last.
- We only need to store s[i, j] if we have at least 2 matrices where $j > i$.

Chain Multiplication

Algorithm 4 Mult(A,i,j)

```
if i < j then
    k ← s[i,j]
    X ← Mult(A, i, k) // X = A[i]...A[k]
    Y = Mult(A, k + 1, j) // Y = A[k+1]...A[j]
    return X * Y
else
    return A[i]
end if
```

Chain Multiplication – Example

The initial set of dimensions are $< 5, 4, 6, 2, 7 >$: we are multiplying A_1 (5x4) times A_2 (4x6) times A_3 (6x2) times A_4 (2x7). Optimal sequence is $(A_1(A_2A_3))A_4$.

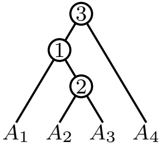
m[i,j]

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

s[i,j]

j = 2	j = 3	j = 4	i
1	1	3	1
	2	3	2
		3	3

Final order



Finding a Recursive Solution

- Figure out the "top-level" choice you have to make (e.g., where to split the list of matrices)
- List the options for that decision
- Each option should require smaller sub-problems to be solved
- Recursive function is the minimum (or max) over all the options

$$m[i,j] = \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j)$$

Steps in Dynamic Programming

- Characterize structure of an optimal solution.
- Define value of optimal solution recursively.
- Compute optimal solution values either top-down with caching or bottom-up in a table.
- Construct an optimal solution from computed values.