Class notes for CSCI 405 Spring 2024

Kameron Decker Harris

April 16, 2024

Textbook: "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein (4th ed). Note that this is the first quarter adapting to the new textbook. Please let me know if page/formula numbers are incorrect.

Chapter 9: Order stats and medians

Reading: Sections 9.1, 9.2

Intro material and proof for 4th edition

Lecture Notes for Chapter 9: Medians and Order Statistics

Chapter 9 overview

- ***i th order statistic*** is the ith smallest element of a set of n elements.
- The ***minimum*** is the first order statistic ($i = 1$).
- The ***maximum*** is the nth order statistic ($i = n$).
- A ***median*** is the "halfway point" of the set.
- When n is odd, the median is unique, at $i = (n + 1)/2$.
- When n is even, there are two medians:
  - The ***lower median***, at $i = n/2$, and
  - The ***upper median***, at $i = n/2 + 1$.
  - We mean lower median when we use the phrase "the median."

The ***selection problem***:

Input: A set A of n distinct numbers and a number i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in A. In other words, the ith smallest element of A.

Easy to solve the selection problem in $O(n \lg n)$ time:

- Sort the numbers using an $O(n \lg n)$-time algorithm, such as heapsort or merge sort.
- Then return the ith element in the sorted array.

There are faster algorithms, however.

- First, we'll look at the problem of selecting the minimum and maximum of a set of elements.
- Then, we'll look at a simple general selection algorithm with a time bound of $O(n)$ in the average case.
- Finally, we'll look at a more complicated general selection algorithm with a time bound of $O(n)$ in the worst case.

Minimum and maximum

We can easily obtain an upper bound of $n-1$ comparisons for finding the minimum of a set of n elements.

- Examine each element in turn and keep track of the smallest one.
- This is the best we can do, because each element, except the minimum, must be compared to a smaller element at least once.

The following pseudocode finds the minimum element in array $A[1:n]$:

```
MINIMUM(A, n)
  min = A[1]
  for i = 2 to n
      if min > A[i]
          min = A[i]
  return min
```

The maximum can be found in exactly the same way by replacing the $>$ with $<$ in the above algorithm.

Simultaneous minimum and maximum

Some applications need both the minimum and maximum of a set of elements.

- For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display. To do so, the program must first find the minimum and maximum of each coordinate.

A simple algorithm to find the minimum and maximum is to find each one independently. There will be $n-1$ comparisons for the minimum and $n-1$ comparisons for the maximum, for a total of $2n-2$ comparisons. This will result in $\Theta(n)$ time.

In fact, at most $3\lfloor n/2 \rfloor$ comparisons suffice to find both the minimum and maximum:

- Maintain the minimum and maximum of elements seen so far.
- Don't compare each element to the minimum and maximum separately.
- Process elements in pairs.
- Compare the elements of a pair to each other.
- Then compare the larger element to the maximum so far, and compare the smaller element to the minimum so far.

This leads to only 3 comparisons for every 2 elements.

Setting up the initial values for the min and max depends on whether n is odd or even.

- If n is even, compare the first two elements and assign the larger to max and the smaller to min. Then process the rest of the elements in pairs.
- If n is odd, set both min and max to the first element. Then process the rest of the elements in pairs.

Analysis of the total number of comparisons

- If n is even, do 1 initial comparison and then $3(n-2)/2$ more comparisons.

$$\begin{aligned}\#\text{ of comparisons} &= \frac{3(n-2)}{2} + 1 \\ &= \frac{3n-6}{2} + 1 \\ &= \frac{3n}{2} - 3 + 1 \\ &= \frac{3n}{2} - 2\,.\end{aligned}$$

- If n is odd, do $3(n-1)/2 = 3\lfloor n/2 \rfloor$ comparisons.

In either case, the maximum number of comparisons is $\leq 3\lfloor n/2 \rfloor$.

Selection in expected linear time

Selection of the ith smallest element of the array A can be done in $\Theta(n)$ time.

The function RANDOMIZED-SELECT uses RANDOMIZED-PARTITION from the quicksort algorithm in Chapter 7. RANDOMIZED-SELECT differs from quicksort because it recurses on one side of the partition only.

```
RANDOMIZED-SELECT(A, p, r, i)
  if p == r
      return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
  q = RANDOMIZED-PARTITION(A, p, r)
  k = q − p + 1
  if i == k
      return A[q]        // the pivot value is the answer
  elseif i < k
      return RANDOMIZED-SELECT(A, p, q − 1, i)
  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

After the call to RANDOMIZED-PARTITION, the array is partitioned into two subarrays $A[p:q-1]$ and $A[q+1:r]$, along with a ***pivot*** element $A[q]$.

- The elements of subarray $A[p:q-1]$ are all $\leq A[q]$.
- The elements of subarray $A[q+1:r]$ are all $> A[q]$.
- The pivot element is the kth element of the subarray $A[p:r]$, where $k = q-p+1$.
- If the pivot element is the ith smallest element (i.e., $i = k$), return $A[q]$.
- Otherwise, recurse on the subarray containing the ith smallest element.
  - If $i < k$, this subarray is $A[p:q-1]$, and we want the ith smallest element.
  - If $i > k$, this subarray is $A[q+1:r]$ and, since there are k elements in $A[p:r]$ that precede $A[q+1:r]$, we want the $(i-k)$th smallest element of this subarray.

Analysis

Worst-case running time

$\Theta(n^2)$, because we could be extremely unlucky and always recurse on a subarray that is only one element smaller than the previous subarray.

Expected running time

RANDOMIZED-SELECT works well on average. Because it is randomized, no particular input brings out the worst-case behavior consistently.

Analysis assumes that the recursion goes as deep as possible: until only one element remains.

Intuition: Suppose that each pivot is in the second or third quartiles if the elements were sorted—in the "middle half." Then at least $1/4$ of the remaining elements are ignored in all future recursive calls $\Rightarrow$ at most $3/4$ of the elements are still ***in play***: somewhere within $A[p:r]$. RANDOMIZE-PARTITION takes $\Theta(n)$ time to partition n elements $\Rightarrow$ recurrence would be $T(n) = T(3n/4) + \Theta(n) = \Theta(n)$ by case 3 of the master method.

What if the pivot is not always in the middle half? Probability that it is in the middle half is $1/2$. View selecting a pivot in the middle half as a Bernoulli trial with probability of success $1/2$. Then the number of trials before a success is a geometric distribution with expected value 2. So that half the time, $1/4$ of the elements go out of play, and the other half of the time, as few as one element (the pivot) goes out of play. But that just doubles the running time, so still expect $\Theta(n)$.

Rigorous analysis:

- Define $A^{(j)}$ as the set of elements still in play (within $A[p:r]$) after j recursive calls (i.e., after jth partitioning). $A^{(0)}$ is all the elements in A.
- $|A^{(j)}|$ is a random variable that depends on A and order statistic i, but not on the order of elements in A.
- Each partitioning removes at least one element (the pivot) $\Rightarrow$ sizes of $A^{(j)}$ strictly decrease.
- jth partitioning takes set $A^{(j-1)}$ and produces $A^{(j)}$.
- Assume a 0th "dummy" partitioning that produces $A^{(0)}$.
- jth partitioning is ***helpful*** if $|A^{(j)}| \le (3/4)|A^{(j-1)}|$. Not all partitionings are necessarily helpful. Think of a helpful partitioning as a successful Bernoulli trial.

Lemma

A partitioning is helpful with probability $\ge 1/2$.

Proof

- Whether or not a partitioning is helpful depends on the randomly chosen pivot.
- Define "middle half" of an n-element subarray as all but the smallest $\lceil n/4 \rceil - 1$ and greatest $\lceil n/4 \rceil - 1$ elements. That is, all but the first and last $\lceil n/4 \rceil - 1$ if the subarray were sorted.

- Will show that if the pivot is in the middle half, then that pivot leads to a helpful partitioning and that the probability that the pivot is in the middle half is $\geq 1/2$.
- No matter where the pivot lies, either all elements $>$ pivot or all elements $<$ pivot, and the pivot itself, are not in play after partitioning $\Rightarrow$ if the pivot is in the middle half, at least the smallest $\lceil n/4 \rceil - 1$ or greatest $\lceil n/4 \rceil - 1$ elements, plus the pivot, will not be in play after partitioning $\Rightarrow \geq \lceil n/4 \rceil$ elements not in play.
- Then, at most $n - \lceil n/4 \rceil = \lfloor 3n/4 \rfloor < 3n/4$ elements in play $\Rightarrow$ partitioning is helpful. ($n - \lceil n/4 \rceil = \lfloor 3n/4 \rfloor$ is from Exercise 3.3-2.)
- To find a lower bound on the probability that a randomly chosen pivot is in the middle half, find an upper bound on the probability that it is not:

$$\begin{aligned}\frac{2(\lceil n/4 \rceil - 1)}{n} &\leq \frac{2((n/4+1)-1)}{n} \quad \text{(inequailty (3.2))} \\ &= \frac{n/2}{n} \\ &= 1/2\,.\end{aligned}$$

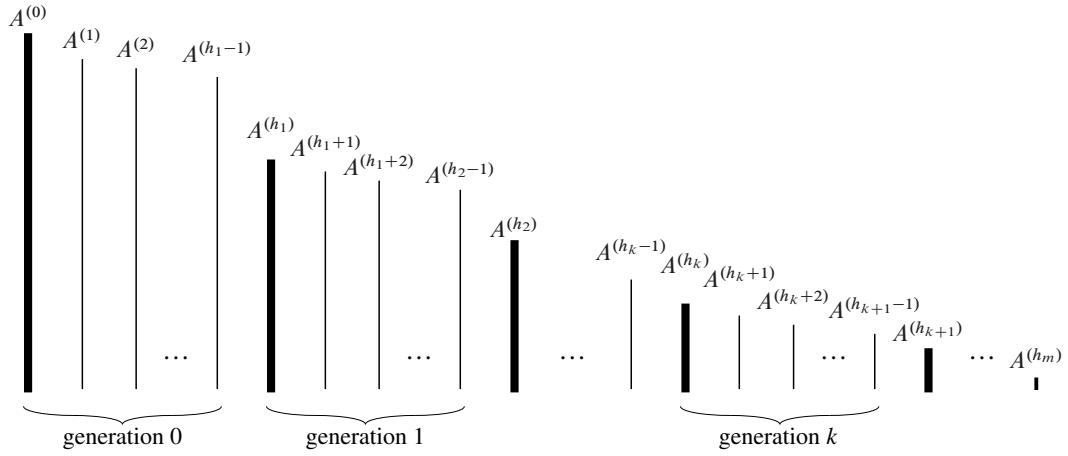
- Since the pivot has probability $\geq 1/2$ of falling into the middle half, a partitioning is helpful with probability $\geq 1/2$. ■ (lemma)

Theorem

The expected running time of RANDOMIZED-SELECT is $\Theta(n)$.

Proof

- Let the sequence of helpful partitionings be $\langle h_0, h_1, \ldots, h_m \rangle$. Consider the 0th partitioning as helpful $\Rightarrow h_0 = 0$. Can bound m, since after at most $\lceil \log_{4/3} n \rceil$ helpful partitionings, only one element remains in play.
- Define $n_k = |A^{(h_k)}|$ and $n_0 = |A^{(0)}|$, the original problem size. $n_k = |A^{(h_k)}| \leq (3/4)|A^{(h_k-1)}| = (3/4)\,n_{k-1}$ for $k = 1, 2, \ldots, m$.
- Iterating gives $n_k \leq (3/4)^k n_0$.
- Break up sets into m "generations." The sets in generation k are $A^{(h_k)}$, $A^{(h_k+1)}$, $\ldots$, $A^{(h_{k+1}-1)}$, where $A^{(h_k)}$ is the result of a helpful partitioning and $A^{(h_{k+1}-1)}$ is the last set before the next helpful partitioning.



[Height of each line indicates the set of the set (number of elements in play). Heavy lines are sets $A^{(h_k)}$, resulting from helpful partitionings and are first within their generation. Other lines are not first within their generation. A generation may contain just one set.]

- If $A^{(j)}$ is in the kth generation, then $|A^{(j)}| \le |A^{(h_k)}| = n_k \le (3/4)^k n_0$.
- Define random variable $X_k = h_{k+1} - h_k$ as the number of sets in the kth generation $\Rightarrow$ kth generation includes sets $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_k+X_k-1)}$.
- By previous lemma, a partitioning is helpful with probability $\ge 1/2$. The probability is even higher, since a partitioning is helpful even if the pivot doesn't fall into middle half, but the ith smallest element lies in the smaller side. Just use the $1/2$ lower bound $\Rightarrow \mathrm{E}[X_k] \le 2$ for $k = 0, 1, \ldots, m-1$ (by equation (C.36), expectation of a geometric distribution).
- The total running time is dominated by the comparisons during partitioning. The jth partitioning takes $A^{(j-1)}$ and compares the pivot with all the other $|A^{(j-1)}| - 1$ elements $\Rightarrow$ jth partitioning makes $< |A^{(j-1)}|$ comparisons.
- The total number of comparisons is less than

$$\begin{aligned}
\sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}| &\le \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(h_k)}| \\
&= \sum_{k=0}^{m-1} X_k \, |A^{(h_k)}| \\
&\le \sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0 \, .
\end{aligned}$$

- Since $\mathrm{E}[X_k] \le 2$, the expected total number of comparisons is less than

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0\right] &= \sum_{k=0}^{m-1} \mathrm{E}\left[X_k \left(\frac{3}{4}\right)^k n_0\right] \quad \text{(linearity of expectation)} \\
&= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \mathrm{E}[X_k] \\
&\le 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\
&< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 8n_0 \quad \text{(infinite geometric series)} \, .
\end{aligned}$$

- n_0 is the size of the original array A $\Rightarrow$ an $O(n)$ upper bound on the expected running time. For the lower bound, the first call of RANDOMIZED-PARTITION examines all n elements $\Rightarrow$ $\Theta(n)$. ■ (theorem)

Therefore, we can determine any order statistic in linear time on average, assuming that all elements are distinct.

3rd edition proof (presented in class)

Analysis

Worst-case running time

$\Theta(n^2)$, because we could be extremely unlucky and always recurse on a subarray that is only 1 element smaller than the previous subarray.

Expected running time

RANDOMIZED-SELECT works well on average. Because it is randomized, no particular input brings out the worst-case behavior consistently.

The running time of RANDOMIZED-SELECT is a random variable that we denote by $T(n)$. We obtain an upper bound on $\mathrm{E}[T(n)]$ as follows:

- RANDOMIZED-PARTITION is equally likely to return any element of A as the pivot.
- For each k such that $1 \le k \le n$, the subarray $A[p..q]$ has k elements (all $\le$ pivot) with probability $1/n$. *[Note that we're now considering a subarray that includes the pivot, along with elements less than the pivot.]*
- For $k = 1, 2, \ldots, n$, define indicator random variable

  $X_k = \mathrm{I}\{\text{subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$.
- Since $\Pr\{\text{subarray } A[p..q] \text{ has exactly } k \text{ elements}\} = 1/n$, Lemma 5.1 says that $\mathrm{E}[X_k] = 1/n$.
- When we call RANDOMIZED-SELECT, we don't know if it will terminate immediately with the correct answer, recurse on $A[p..q-1]$, or recurse on $A[q+1..r]$. It depends on whether the ith smallest element is less than, equal to, or greater than the pivot element $A[q]$.
- To obtain an upper bound, we assume that $T(n)$ is monotonically increasing and that the ith smallest element is always in the larger subarray.
- For a given call of RANDOMIZED-SELECT, $X_k = 1$ for exactly one value of k, and $X_k = 0$ for all other k.
- When $X_k = 1$, the two subarrays have sizes $k-1$ and $n-k$.
- For a subproblem of size n, RANDOMIZED-PARTITION takes $O(n)$ time. *[Actually, it takes $\Theta(n)$ time, but $O(n)$ suffices, since we're obtaining only an upper bound on the expected running time.]*
- Therefore, we have the recurrence

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) \; . \end{aligned}$$

- Taking expected values gives

$$\begin{aligned} &\mathrm{E}[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \end{aligned}$$

$$
\begin{aligned}
&= \sum_{k=1}^{n} \mathrm{E}\left[X_k \cdot T(\max(k-1, n-k))\right] + O(n) \qquad \text{(linearity of expectation)} \\
&= \sum_{k=1}^{n} \mathrm{E}\left[X_k\right] \cdot \mathrm{E}\left[T(\max(k-1, n-k))\right] + O(n) \quad \text{(equation (C.24))} \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[T(\max(k-1, n-k))\right] + O(n) \; .
\end{aligned}
$$

- We rely on X_k and $T(\max(k-1, n-k))$ being independent random variables in order to apply equation (C.24).
- Looking at the expression $\max(k-1, n-k)$, we have

$$
\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \; , \\ n-k & \text{if } k \le \lceil n/2 \rceil \; . \end{cases}
$$

  - If n is even, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears exactly twice in the summation.
  - If n is odd, these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once.

- Either way,

$$
\mathrm{E}\left[T(n)\right] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}\left[T(k)\right] + O(n) \; .
$$

- Solve this recurrence by substitution:
  - Guess that $T(n) \le cn$ for some constant c that satisfies the initial conditions of the recurrence.
  - Assume that $T(n) = O(1)$ for $n <$ some constant. We'll pick this constant later.
  - Also pick a constant a such that the function described by the $O(n)$ term is bounded from above by an for all $n > 0$.
  - Using this guess and constants c and a, we have

$$
\begin{aligned}
\mathrm{E}\left[T(n)\right] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\,\lfloor n/2 \rfloor}{2} \right) + an \\
&\le \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an
\end{aligned}
$$

$$\begin{aligned} &= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\ &\le \frac{3cn}{4} + \frac{c}{2} + an \\ &= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) . \end{aligned}$$

- To complete this proof, we choose c such that

$$\begin{aligned} cn/4 - c/2 - an &\ge 0 \\ cn/4 - an &\ge c/2 \\ n(c/4 - a) &\ge c/2 \\ n &\ge \frac{c/2}{c/4 - a} \\ n &\ge \frac{2c}{c - 4a} . \end{aligned}$$

- Thus, as long as we assume that $T(n) = O(1)$ for $n < 2c/(c-4a)$, we have $\mathrm{E}[T(n)] = O(n)$.

Therefore, we can determine any order statistic in linear time on average.

Selection in worst-case linear time

We can find the ith smallest element in $O(n)$ time *in the worst case*. We'll describe a procedure SELECT that does so.

SELECT recursively partitions the input array.

- ***Idea:*** Guarantee a good split when the array is partitioned.
- Will use the deterministic procedure PARTITION, but with a small modification. Instead of assuming that the last element of the subarray is the pivot, the modified PARTITION procedure is told which element to use as the pivot.

SELECT works on an array of $n > 1$ elements. It executes the following steps:

1. Divide the n elements into groups of 5. Get $\lceil n/5 \rceil$ groups: $\lfloor n/5 \rfloor$ groups with exactly 5 elements and, if 5 does not divide n, one group with the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups:
   - Run insertion sort on each group. Takes $O(1)$ time per group since each group has ≤ 5 elements.
   - Then just pick the median from each group, in $O(1)$ time.
3. Find the median x of the $\lceil n/5 \rceil$ medians by a recursive call to SELECT. (If $\lceil n/5 \rceil$ is even, then follow our convention and find the lower median.)
4. Using the modified version of PARTITION that takes the pivot element as input, partition the input array around x. Let x be the kth element of the array after partitioning, so that there are $k-1$ elements on the low side of the partition and $n-k$ elements on the high side.

Chapter 14: Dynamic programming

Reading: Sections 14.1–14.4

Lecture Notes for Chapter 14: Dynamic Programming

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.
  - Minimization or maximization. (We'll see both.)

Four-step method

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod cutting

How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integer number of inches.

Input: A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.

Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

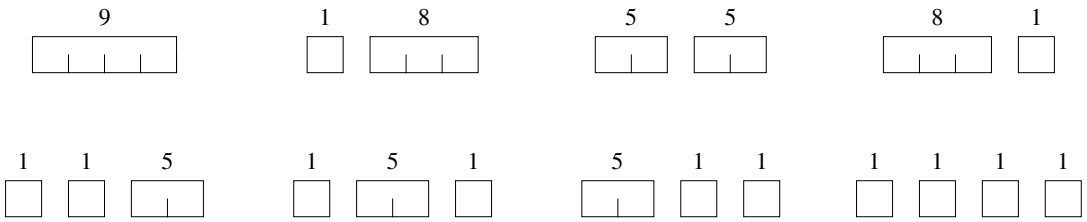
If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

Example: *[Using the first* 8 *values from the example in the textbook.]*

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.

Here are all 8 ways to cut a rod of length 4, with the costs from the example:



The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$.

Let r_i be the maximum revenue for a rod of length i. Can express a solution as a sum of individual rod lengths.

Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

Can determine optimal revenue r_n by taking the maximum of

- p_n: the revenue from not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
- $r_{n-1} + r_1$.

That is,

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} \ .$$

Optimal substructure: To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, two subproblems remain. The optimal solution to the original problem incorporates optimal solutions to the subproblems. May solve the subproblems independently.

Example: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. Need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

A simpler way to decompose the problem: Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.

- Need to divide only the remainder, not the first piece.
- Leaves only one subproblem to solve, rather than two subproblems.
- Say that the solution with no cuts has first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.
- Gives a simpler version of the equation for r_n:

  $$r_n = \max\{p_i + r_{n-i} : 1 \le i \le n\} \ .$$

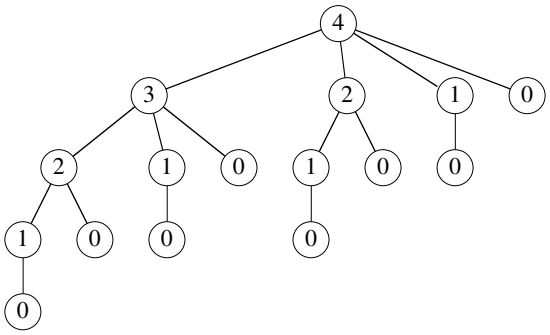
Recursive top-down solution

Direct implementation of the simpler equation for r_n.
The call CUT-ROD(p, n) returns the optimal revenue r_n:

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = -∞
  for i = 1 to n
      q = max{q, p[i] + CUT-ROD(p, n - i)}
  return q
```

This procedure works, but it is terribly *inefficient*. If you code it up and run it, it could take more than an hour for $n = 40$. Running time approximately doubles each time n increases by 1.

Why so inefficient?: CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



Lots of repeated subproblems. Solves the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

Exponential growth: Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \,, \\ 1 + \displaystyle\sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 \,. \end{cases}$$

Summation counts calls where second parameter is $j = n - i$.
Solution to recurrence is $T(n) = 2^n$.

Dynamic-programming solution

Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
"Store, don't recompute" $\Rightarrow$ time-memory trade-off.
Can turn an exponential-time solution into a polynomial-time solution.
Two basic approaches: top-down with memoization, and bottom-up.

Top-down with memoization

Solve recursively, but store each result in a table.
To find the solution to a subproblem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.

Memoizing is remembering what has been computed previously. *["Memoizing," not "memorizing."]*

Memoized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:

```
MEMOIZED-CUT-ROD(p, n)
  let r[0:n] be a new array          // will remember solution values in r
  for i = 0 to n
      r[i] = -∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0                        // already have a solution for length n?
      return r[n]
  if n == 0
      q = 0
  else q = -∞
      for i = 1 to n                 // i is the position of the first cut
          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
  r[n] = q                           // remember the solution value for length n
  return q
```

Bottom-up

Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems needed.

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] be a new array        // will remember solution values in r
  r[0] = 0
  for j = 1 to n                   // for increasing rod length j
      q = -∞
      for i = 1 to j               // i is the position of the first cut
          q = max{q, p[i] + r[j - i]}
      r[j] = q                     // remember the solution value for length j
  return r[n]
```

Running time

Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.

- Bottom-up: Doubly nested loops. Number of iterations of inner **for** loop forms an arithmetic series.
- Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop iterates n times $\Rightarrow$ over all recursive calls, total number of iterations forms an arithmetic series. *[Actually using aggregate analysis, which Chapter 16 covers.]*

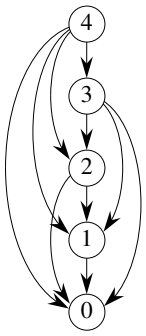
Subproblem graphs

How to understand the subproblems involved and how they depend on each other.

Directed graph:

- One vertex for each distinct subproblem.
- Has a directed edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y.

Example: For rod-cutting problem with $n = 4$:



Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.

Subproblem graph can help determine running time. Because each subproblem is solved just once, running time is sum of times needed to solve each subproblem.

- Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

When these conditions hold, running time is linear in number of vertices and edges.

Reconstructing a solution

So far, have focused on computing the *value* of an optimal solution, rather than the *choices* that produced an optimal solution.

Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] and s[1:n] be new arrays
  r[0] = 0
  for j = 1 to n                    // for increasing rod length j
      q = -∞
      for i = 1 to j                // i is the position of the first cut
          if q < p[i] + r[j - i]
              q = p[i] + r[j - i]
              s[j] = i              // best cut location so far for length j
      r[j] = q                      // remember the solution value for length j
  return r and s
```

Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.

To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]          // cut location for length n
      n = n - s[n]        // length of the remainder of the rod
```

Example: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$		1	2	3	2	2	6	1	2

A call to PRINT-CUT-ROD-SOLUTION$(p, 8)$ calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

Matrix-chain multiplication

Problem: Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$ using standard matrix multiplication (not Strassen's method) while minimizing the number of scalar multiplications.

Lecture Notes for Chapter 14: Dynamic Programming

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.
  - Minimization or maximization. (We'll see both.)

Four-step method

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod cutting

How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integer number of inches.

Input: A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.

Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

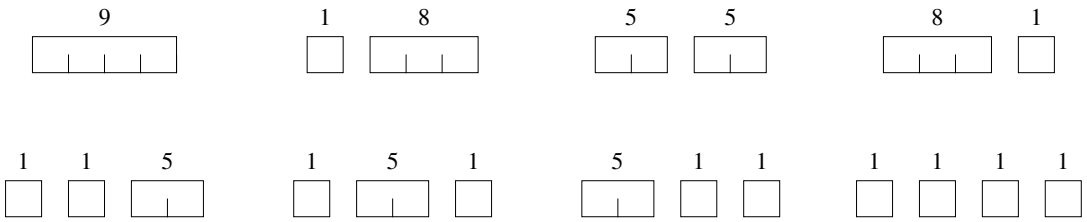
If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

Example: *[Using the first* 8 *values from the example in the textbook.]*

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.

Here are all 8 ways to cut a rod of length 4, with the costs from the example:



The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$.

Let r_i be the maximum revenue for a rod of length i. Can express a solution as a sum of individual rod lengths.

Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

Can determine optimal revenue r_n by taking the maximum of

- p_n: the revenue from not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
- $r_{n-1} + r_1$.

That is,

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} \ .$$

Optimal substructure: To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, two subproblems remain. The optimal solution to the original problem incorporates optimal solutions to the subproblems. May solve the subproblems independently.

Example: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. Need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

A simpler way to decompose the problem: Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.

- Need to divide only the remainder, not the first piece.
- Leaves only one subproblem to solve, rather than two subproblems.
- Say that the solution with no cuts has first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.
- Gives a simpler version of the equation for r_n:

  $$r_n = \max\{p_i + r_{n-i} : 1 \le i \le n\} \ .$$

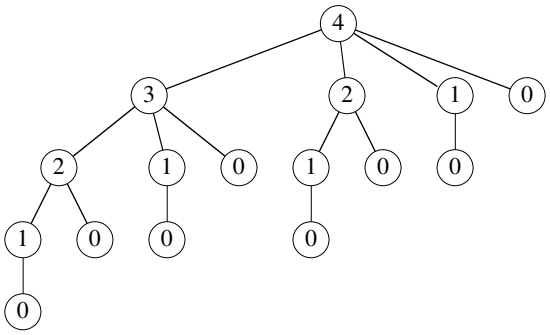
Recursive top-down solution

Direct implementation of the simpler equation for r_n.
The call CUT-ROD(p, n) returns the optimal revenue r_n:

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = -∞
  for i = 1 to n
      q = max{q, p[i] + CUT-ROD(p, n - i)}
  return q
```

This procedure works, but it is terribly *inefficient*. If you code it up and run it, it could take more than an hour for $n = 40$. Running time approximately doubles each time n increases by 1.

Why so inefficient?: CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



Lots of repeated subproblems. Solves the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

Exponential growth: Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \,, \\ 1 + \displaystyle\sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 \,. \end{cases}$$

Summation counts calls where second parameter is $j = n - i$.
Solution to recurrence is $T(n) = 2^n$.

Dynamic-programming solution

Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
"Store, don't recompute" $\Rightarrow$ time-memory trade-off.
Can turn an exponential-time solution into a polynomial-time solution.
Two basic approaches: top-down with memoization, and bottom-up.

Top-down with memoization

Solve recursively, but store each result in a table.
To find the solution to a subproblem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.

Memoizing is remembering what has been computed previously. *["Memoizing," not "memorizing."]*

Memoized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:

```
MEMOIZED-CUT-ROD(p, n)
  let r[0:n] be a new array          // will remember solution values in r
  for i = 0 to n
      r[i] = -∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0                        // already have a solution for length n?
      return r[n]
  if n == 0
      q = 0
  else q = -∞
      for i = 1 to n                 // i is the position of the first cut
          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
  r[n] = q                           // remember the solution value for length n
  return q
```

Bottom-up

Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems needed.

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] be a new array        // will remember solution values in r
  r[0] = 0
  for j = 1 to n                   // for increasing rod length j
      q = -∞
      for i = 1 to j               // i is the position of the first cut
          q = max{q, p[i] + r[j - i]}
      r[j] = q                     // remember the solution value for length j
  return r[n]
```

Running time

Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.

- Bottom-up: Doubly nested loops. Number of iterations of inner **for** loop forms an arithmetic series.
- Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop iterates n times $\Rightarrow$ over all recursive calls, total number of iterations forms an arithmetic series. *[Actually using aggregate analysis, which Chapter 16 covers.]*

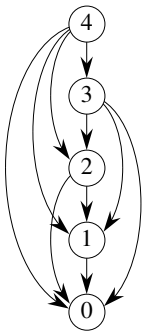
Subproblem graphs

How to understand the subproblems involved and how they depend on each other.

Directed graph:

- One vertex for each distinct subproblem.
- Has a directed edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y.

Example: For rod-cutting problem with $n = 4$:



Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.

Subproblem graph can help determine running time. Because each subproblem is solved just once, running time is sum of times needed to solve each subproblem.

- Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

When these conditions hold, running time is linear in number of vertices and edges.

Reconstructing a solution

So far, have focused on computing the *value* of an optimal solution, rather than the *choices* that produced an optimal solution.

Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] and s[1:n] be new arrays
  r[0] = 0
  for j = 1 to n                    // for increasing rod length j
      q = -∞
      for i = 1 to j                // i is the position of the first cut
          if q < p[i] + r[j - i]
              q = p[i] + r[j - i]
              s[j] = i              // best cut location so far for length j
      r[j] = q                      // remember the solution value for length j
  return r and s
```

Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.

To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]          // cut location for length n
      n = n - s[n]        // length of the remainder of the rod
```

Example: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$		1	2	3	2	2	6	1	2

A call to PRINT-CUT-ROD-SOLUTION$(p, 8)$ calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

Matrix-chain multiplication

Problem: Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$ using standard matrix multiplication (not Strassen's method) while minimizing the number of scalar multiplications.

How to parenthesize the product to minimize the number of scalar multiplications?

Suppose multiplying matrices A and B: $C = A \cdot B$. *[The textbook has a procedure to compute $C = C + A \cdot B$, but it's easier in a lecture situation to just use $C = A \cdot B$.]* The matrices must be compatible: number of columns of A equals number of rows of B. If A is $p \times q$ and B is $q \times r$, then C is $p \times r$ and takes pqr scalar multiplications.

Example: $A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$. Compute $A_1 A_2 A_3$, which is 10×50.

- Try parenthesizing by $((A_1 A_2) A_3)$. First perform $10 \cdot 100 \cdot 5 = 5000$ multiplications, then perform $10 \cdot 5 \cdot 50 = 2500$, for a total of 7500.
- Try parenthesizing by $(A_1 (A_2 A_3))$. First perform $100 \cdot 5 \cdot 50 = 25{,}000$ multiplications, then perform $10 \cdot 100 \cdot 50 = 50{,}000$, for a total of 75,000.
- The first way is 10 times faster.

Input to the problem: Let A_i be $p_{i-1} \times p_i$. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

Note: Not actually multiplying matrices. Just deciding an order with the lowest cost.

Counting the number of parenthesizations

Let $P(n)$ denote the number of ways to parenthesize a product of n matrices. $P(1) = 1$.

When $n \geq 2$, can split anywhere between A_k and A_{k+1} for $k = 1, 2, \ldots, n-1$. Then have to split the subproducts. Get

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

The solution is $P(n) = \Omega(4^n / n^{3/2})$. *[The textbook does not prove the solution to this recurrence.]* So brute force is a bad strategy.

Step 1: Structure of an optimal solution

Let $A_{i:j}$ be the matrix product $A_i A_{i+1} \cdots A_j$.

If $i < j$, then must split between A_k and A_{k+1} for some $i \leq k < j \Rightarrow$ compute $A_{i:k}$ and $A_{k+1:j}$ and then multiply them together. Cost is

cost of computing $A_{i:k}$
\+ cost of computing $A_{k+1:j}$
\+ cost of multiplying them together .

Optimal substructure: Suppose that optimal parenthesization of $A_{i:j}$ splits between A_k and A_{k+1}. Then the parenthesization of $A_{i:k}$ must be optimal. Otherwise, if there's a less costly way to parenthesize it, you'd use it and get a parenthesization of $A_{i:j}$ with a lower cost. Same for $A_{k+1:j}$.

Therefore, to build an optimal solution to $A_{i:j}$, split it into how to optimally parenthesize $A_{i:k}$ and $A_{k+1:j}$, find optimal solutions to these subproblems, and then combine the optimal solutions. Need to consider all possible splits.

Step 2: A recursive solution

Define the cost of an optimal solution recursively in terms of optimal subproblem solutions.

Let $m[i, j]$ be the minimum number of scalar multiplications to compute $A_{i:j}$. For the full problem, want $m[1, n]$.

If $i = j$, then just one matrix $\Rightarrow m[i, i] = 0$ for $i = 1, 2, \ldots, n$.

If $i < j$, then suppose the optimal split is between A_k and A_{k+1}, where $i \le k < j$. Then $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_i p_j$.

But that's assuming you know the value of k. Have to try all possible values and pick the best, so that

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j : i \le k < j\} & \text{if } i < j \ . \end{cases}$$

That formula gives the cost of an optimal solution, but not how to construct it. Define $s[i, j]$ to be a value of k to split $A_{i:j}$ in an optimal parenthesization. Then $s[i, j] = k$ such that $m[i, j] = m[i, k] + m[k_1, j] + p_{i-1} p_k p_j$.

Step 3: Compute the optimal costs

Could implement a recursive algorithm based on the above equation for $m[i, j]$. *Problem:* It would take exponential time.

There are not all that many subproblems: just one for each i, j such that $1 \le i \le j \le n$. There are $\binom{n}{2} + n = \Theta(n^2)$ of them. Thus, a recursive algorithm would solve the same subproblems over and over.

In other words, this problem has overlapping subproblems.

Here is a tabular, bottom-up method to solve the problem. It solves subproblems in order of increasing chain length. The variable $l = j - i + 1$ indicates the chain length.

```
MATRIX-CHAIN-ORDER(p, n)
  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
  for i = 1 to n                          // chain length 1
      m[i, i] = 0
  for l = 2 to n                          // l is the chain length
      for i = 1 to n − l + 1              // chain begins at A_i
          j = i + l − 1                   // chain ends at A_j
          m[i, j] = ∞
          m[i, j] = ∞
          for k = i to j − 1
              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
              if q < m[i, j]
                  m[i, j] = q             // remember this cost
                  s[i, j] = k             // remember this index
  return m and s
```

All n chains of length 1 are initialized so that $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. Then $n - 1$ chains of length 2 are computed, then $n - 2$ chains of length 3, and so on, up to 1 chain of length n.

[We don't include an example here because the arithmetic is hard for students to process in real time.]

Time: $O(n^3)$, from triply nested loops. Also $\Omega(n^3) \Rightarrow \Theta(n^3)$.

Step 4: Construct an optimal solution

With the s table filled in, recursively print an optimal solution.

```
PRINT-OPTIMAL-PARENS(s, i, j)
  if i == j
      print "A"_i
  else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```

Initial call is PRINT-OPTIMAL-PARENS$(s, 1, n)$

Longest common subsequence

[The textbook has the section on elements of dynamic programming next, but these lecture notes reserve that section for the end of the chapter so that it may refer to two more examples of dynamic programming.]

Problem: Given two sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

```
MATRIX-CHAIN-ORDER(p, n)
  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
  for i = 1 to n                          // chain length 1
      m[i, i] = 0
  for l = 2 to n                          // l is the chain length
      for i = 1 to n − l + 1              // chain begins at A_i
          j = i + l − 1                   // chain ends at A_j
          m[i, j] = ∞
          m[i, j] = ∞
          for k = i to j − 1
              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
              if q < m[i, j]
                  m[i, j] = q             // remember this cost
                  s[i, j] = k             // remember this index
  return m and s
```

All n chains of length 1 are initialized so that $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. Then $n - 1$ chains of length 2 are computed, then $n - 2$ chains of length 3, and so on, up to 1 chain of length n.

[We don't include an example here because the arithmetic is hard for students to process in real time.]

Time: $O(n^3)$, from triply nested loops. Also $\Omega(n^3) \Rightarrow \Theta(n^3)$.

Step 4: Construct an optimal solution

With the s table filled in, recursively print an optimal solution.

```
PRINT-OPTIMAL-PARENS(s, i, j)
  if i == j
      print "A"_i
  else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```

Initial call is PRINT-OPTIMAL-PARENS$(s, 1, n)$

Longest common subsequence

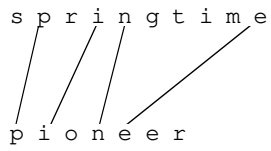
[The textbook has the section on elements of dynamic programming next, but these lecture notes reserve that section for the end of the chapter so that it may refer to two more examples of dynamic programming.]

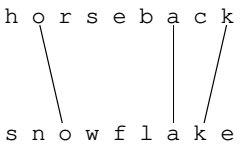
Problem: Given two sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

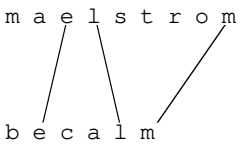
[To come up with examples of longest common subsequences, search the dictionary for all words that contain the word you are looking for as a subsequence. On a UNIX system, for example, to find all the words with `pine` as a subsequence, use the command `grep '.*p.*i.*n.*e.*' dict`, where `dict` is your local dictionary. Then check if that word is actually a longest common subsequence. Working C code for finding a longest common subsequence of two strings appears at http://www.cs.dartmouth.edu/~thc/code/lcs.c The comments in the code refer to the second edition of the textbook, but the code is correct.]

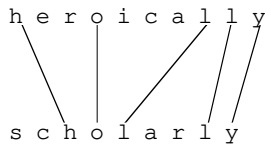
Examples

[The examples are of different types of trees.]









Brute-force algorithm:

For every subsequence of X, check whether it's a subsequence of Y.

Time: $\Theta(n2^m)$.

- 2^m subsequences of X to check.
- Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

Step 1: Characterize an LCS

Notation:

X_i = prefix $\langle x_1, \ldots, x_i \rangle$

Y_i = prefix $\langle y_1, \ldots, y_i \rangle$

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ 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 $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Proof

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = \langle z_1, \ldots, z_k, x_m \rangle$. It's a common subsequence of X and Y and has length $k + 1 \Rightarrow Z'$ is a longer common subsequence than $Z \Rightarrow$ contradicts Z being an LCS.

   Now show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. Clearly, it's a common subsequence. Now suppose there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence W' by appending x_m to W. W' is common subsequence of X and Y, has length $\geq k + 1$ $\Rightarrow$ contradicts Z being an LCS.
2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$. Then W is a common subsequence of X and $Y \Rightarrow$ contradicts Z being an LCS.
3. Symmetric to 2. ■ (theorem)

Therefore, an LCS of two sequences contains as a prefix an LCS of prefixes of the sequences.

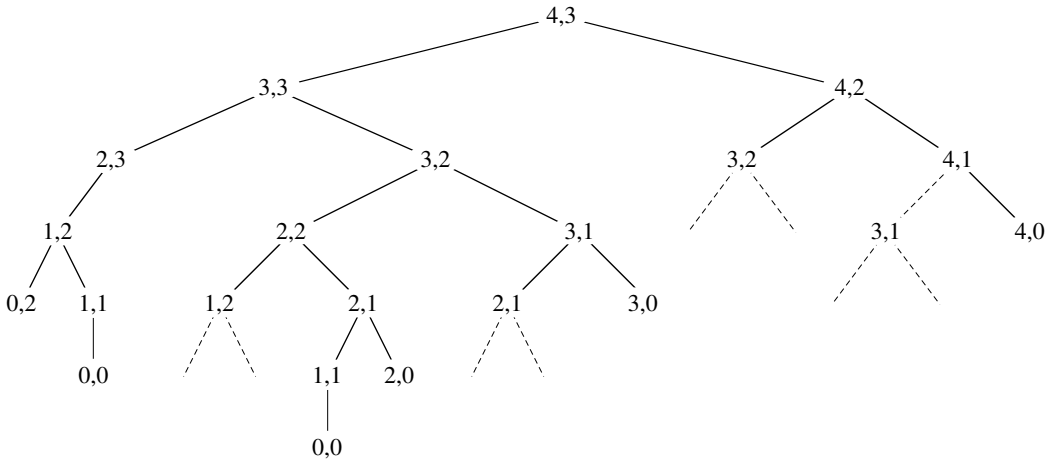
Step 2: Recursively define an optimal solution

Define $c[i, j]$ = length of LCS of X_i and Y_j. Want $c[m, n]$.

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

Again, could write a recursive algorithm based on this formulation.

Try with $X = \langle a, t, o, m \rangle$ and $Y = \langle a, n, t \rangle$. Numbers in nodes are values of i, j in each recursive call. Dashed lines indicate subproblems already computed.



- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

Step 3: Compute the length of an LCS

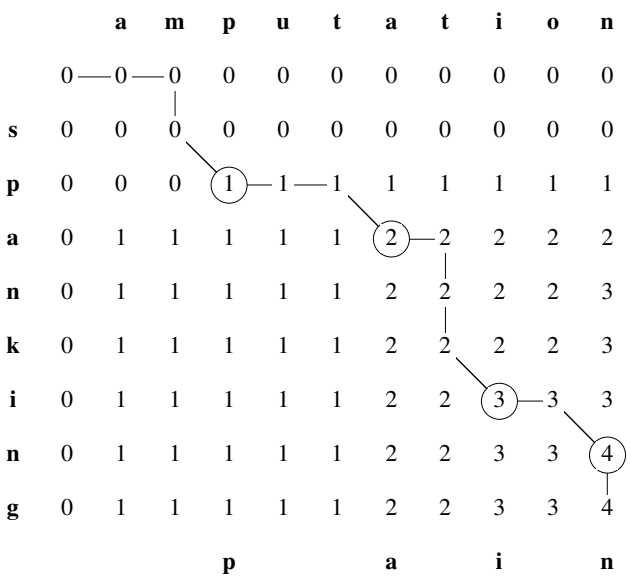
```
LCS-LENGTH(X, Y, m, n)
  let b[1:m, 1:n] and c[0:m, 0:n] be new tables
  for i = 1 to m
      c[i, 0] = 0
  for j = 0 to n
      c[0, j] = 0
  for i = 1 to m          // compute table entries in row-major order
      for j = 1 to n
          if x_i == y_j
              c[i, j] = c[i − 1, j − 1] + 1
              b[i, j] = "↖"
          else if c[i − 1, j] ≥ c[i, j − 1]
                  c[i, j] = c[i − 1, j]
                  b[i, j] = "↑"
          else c[i, j] = c[i, j − 1]
                  b[i, j] = "←"
  return c and b
```

```
PRINT-LCS(b, X, i, j)
  if i == 0 or j = 0
      return              // the LCS has length 0
  if b[i, j] == "↖"
      PRINT-LCS(b, X, i − 1, j − 1)
      print x_i           // same as y_j
  elseif b[i, j] == "↑"
      PRINT-LCS(b, X, i − 1, j)
  else PRINT-LCS(b, X, i, j − 1)
```

- Initial call is PRINT-LCS(b, X, m, n).
- $b[i, j]$ points to table entry whose subproblem was used in solving LCS of X_i and Y_j.
- When $b[i, j] = \nwarrow$, LCS extended by one character. So longest common subsequence = entries with $\nwarrow$ in them.

Demonstration

What do `spanking` and `amputation` have in common? *[Show only $c[i, j]$.]*



Answer: `pain`.

Time

$\Theta(mn)$

Improving the code

Don't really need the b table. $c[i, j]$ depends only on $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given $c[i, j]$, can determine in constant time which of the three values was used to compute $c[i, j]$. *[Exercise 14.4-2.]*

Or, if only need the length of an LCS, and don't need to construct the LCS itself, can get away with storing only one row of the c table plus a constant amount of additional entries. *[Exercise 14.4-4.]*

Optimal binary search trees

- Given sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys, sorted ($k_1 < k_2 < \cdots < k_n$).
- Want to build a binary search tree from the keys.
- For k_i, have probability p_i that a search is for k_i.
- Want BST with minimum expected search cost.
- Actual cost = # of items examined.

  For key k_i, cost = $\text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T.

E [search cost in T]

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

Lecture Notes for Chapter 14: Dynamic Programming

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.
  - Minimization or maximization. (We'll see both.)

Four-step method

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod cutting

How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integer number of inches.

Input: A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.

Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

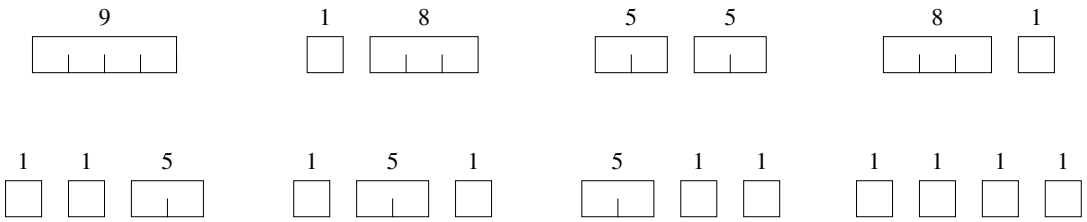
If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

Example: *[Using the first* 8 *values from the example in the textbook.]*

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.

Here are all 8 ways to cut a rod of length 4, with the costs from the example:



The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$.

Let r_i be the maximum revenue for a rod of length i. Can express a solution as a sum of individual rod lengths.

Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

Can determine optimal revenue r_n by taking the maximum of

- p_n: the revenue from not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
- $r_{n-1} + r_1$.

That is,

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} \ .$$

Optimal substructure: To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, two subproblems remain. The optimal solution to the original problem incorporates optimal solutions to the subproblems. May solve the subproblems independently.

Example: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. Need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

A simpler way to decompose the problem: Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.

- Need to divide only the remainder, not the first piece.
- Leaves only one subproblem to solve, rather than two subproblems.
- Say that the solution with no cuts has first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.
- Gives a simpler version of the equation for r_n:

  $$r_n = \max\{p_i + r_{n-i} : 1 \le i \le n\} \ .$$

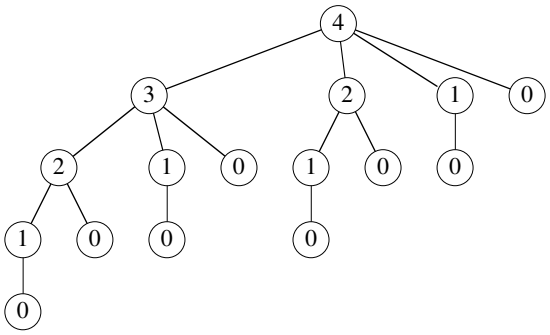
Recursive top-down solution

Direct implementation of the simpler equation for r_n.
The call CUT-ROD(p, n) returns the optimal revenue r_n:

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = -∞
  for i = 1 to n
      q = max{q, p[i] + CUT-ROD(p, n - i)}
  return q
```

This procedure works, but it is terribly *inefficient*. If you code it up and run it, it could take more than an hour for $n = 40$. Running time approximately doubles each time n increases by 1.

Why so inefficient?: CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



Lots of repeated subproblems. Solves the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

Exponential growth: Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \,, \\ 1 + \displaystyle\sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 \,. \end{cases}$$

Summation counts calls where second parameter is $j = n - i$.
Solution to recurrence is $T(n) = 2^n$.

Dynamic-programming solution

Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
"Store, don't recompute" $\Rightarrow$ time-memory trade-off.
Can turn an exponential-time solution into a polynomial-time solution.
Two basic approaches: top-down with memoization, and bottom-up.

Top-down with memoization

Solve recursively, but store each result in a table.
To find the solution to a subproblem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.

Memoizing is remembering what has been computed previously. *["Memoizing," not "memorizing."]*

Memoized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:

```
MEMOIZED-CUT-ROD(p, n)
  let r[0:n] be a new array          // will remember solution values in r
  for i = 0 to n
      r[i] = -∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0                        // already have a solution for length n?
      return r[n]
  if n == 0
      q = 0
  else q = -∞
      for i = 1 to n                 // i is the position of the first cut
          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
  r[n] = q                           // remember the solution value for length n
  return q
```

Bottom-up

Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems needed.

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] be a new array        // will remember solution values in r
  r[0] = 0
  for j = 1 to n                   // for increasing rod length j
      q = -∞
      for i = 1 to j               // i is the position of the first cut
          q = max{q, p[i] + r[j - i]}
      r[j] = q                     // remember the solution value for length j
  return r[n]
```

Running time

Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.

- Bottom-up: Doubly nested loops. Number of iterations of inner **for** loop forms an arithmetic series.
- Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop iterates n times $\Rightarrow$ over all recursive calls, total number of iterations forms an arithmetic series. *[Actually using aggregate analysis, which Chapter 16 covers.]*

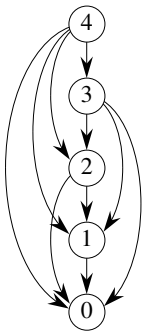
Subproblem graphs

How to understand the subproblems involved and how they depend on each other.

Directed graph:

- One vertex for each distinct subproblem.
- Has a directed edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y.

Example: For rod-cutting problem with $n = 4$:



Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.

Subproblem graph can help determine running time. Because each subproblem is solved just once, running time is sum of times needed to solve each subproblem.

- Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

When these conditions hold, running time is linear in number of vertices and edges.

Reconstructing a solution

So far, have focused on computing the *value* of an optimal solution, rather than the *choices* that produced an optimal solution.

Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0:n] and s[1:n] be new arrays
  r[0] = 0
  for j = 1 to n                    // for increasing rod length j
      q = -∞
      for i = 1 to j                // i is the position of the first cut
          if q < p[i] + r[j - i]
              q = p[i] + r[j - i]
              s[j] = i              // best cut location so far for length j
      r[j] = q                      // remember the solution value for length j
  return r and s
```

Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.

To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]          // cut location for length n
      n = n - s[n]        // length of the remainder of the rod
```

Example: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$		1	2	3	2	2	6	1	2

A call to PRINT-CUT-ROD-SOLUTION$(p, 8)$ calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

Matrix-chain multiplication

Problem: Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$ using standard matrix multiplication (not Strassen's method) while minimizing the number of scalar multiplications.

How to parenthesize the product to minimize the number of scalar multiplications?

Suppose multiplying matrices A and B: $C = A \cdot B$. *[The textbook has a procedure to compute $C = C + A \cdot B$, but it's easier in a lecture situation to just use $C = A \cdot B$.]* The matrices must be compatible: number of columns of A equals number of rows of B. If A is $p \times q$ and B is $q \times r$, then C is $p \times r$ and takes pqr scalar multiplications.

Example: $A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$. Compute $A_1 A_2 A_3$, which is 10×50.

- Try parenthesizing by $((A_1 A_2) A_3)$. First perform $10 \cdot 100 \cdot 5 = 5000$ multiplications, then perform $10 \cdot 5 \cdot 50 = 2500$, for a total of 7500.
- Try parenthesizing by $(A_1 (A_2 A_3))$. First perform $100 \cdot 5 \cdot 50 = 25{,}000$ multiplications, then perform $10 \cdot 100 \cdot 50 = 50{,}000$, for a total of 75,000.
- The first way is 10 times faster.

Input to the problem: Let A_i be $p_{i-1} \times p_i$. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

Note: Not actually multiplying matrices. Just deciding an order with the lowest cost.

Counting the number of parenthesizations

Let $P(n)$ denote the number of ways to parenthesize a product of n matrices. $P(1) = 1$.

When $n \geq 2$, can split anywhere between A_k and A_{k+1} for $k = 1, 2, \ldots, n-1$. Then have to split the subproducts. Get

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

The solution is $P(n) = \Omega(4^n / n^{3/2})$. *[The textbook does not prove the solution to this recurrence.]* So brute force is a bad strategy.

Step 1: Structure of an optimal solution

Let $A_{i:j}$ be the matrix product $A_i A_{i+1} \cdots A_j$.

If $i < j$, then must split between A_k and A_{k+1} for some $i \leq k < j \Rightarrow$ compute $A_{i:k}$ and $A_{k+1:j}$ and then multiply them together. Cost is

cost of computing $A_{i:k}$
\+ cost of computing $A_{k+1:j}$
\+ cost of multiplying them together .

Optimal substructure: Suppose that optimal parenthesization of $A_{i:j}$ splits between A_k and A_{k+1}. Then the parenthesization of $A_{i:k}$ must be optimal. Otherwise, if there's a less costly way to parenthesize it, you'd use it and get a parenthesization of $A_{i:j}$ with a lower cost. Same for $A_{k+1:j}$.

Therefore, to build an optimal solution to $A_{i:j}$, split it into how to optimally parenthesize $A_{i:k}$ and $A_{k+1:j}$, find optimal solutions to these subproblems, and then combine the optimal solutions. Need to consider all possible splits.

Step 2: A recursive solution

Define the cost of an optimal solution recursively in terms of optimal subproblem solutions.

Let $m[i, j]$ be the minimum number of scalar multiplications to compute $A_{i:j}$. For the full problem, want $m[1, n]$.

If $i = j$, then just one matrix $\Rightarrow m[i, i] = 0$ for $i = 1, 2, \ldots, n$.

If $i < j$, then suppose the optimal split is between A_k and A_{k+1}, where $i \le k < j$. Then $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_i p_j$.

But that's assuming you know the value of k. Have to try all possible values and pick the best, so that

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j : i \le k < j\} & \text{if } i < j \ . \end{cases}$$

That formula gives the cost of an optimal solution, but not how to construct it. Define $s[i, j]$ to be a value of k to split $A_{i:j}$ in an optimal parenthesization. Then $s[i, j] = k$ such that $m[i, j] = m[i, k] + m[k_1, j] + p_{i-1} p_k p_j$.

Step 3: Compute the optimal costs

Could implement a recursive algorithm based on the above equation for $m[i, j]$. *Problem:* It would take exponential time.

There are not all that many subproblems: just one for each i, j such that $1 \le i \le j \le n$. There are $\binom{n}{2} + n = \Theta(n^2)$ of them. Thus, a recursive algorithm would solve the same subproblems over and over.

In other words, this problem has overlapping subproblems.

Here is a tabular, bottom-up method to solve the problem. It solves subproblems in order of increasing chain length. The variable $l = j - i + 1$ indicates the chain length.

```
MATRIX-CHAIN-ORDER(p, n)
  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
  for i = 1 to n                          // chain length 1
      m[i, i] = 0
  for l = 2 to n                          // l is the chain length
      for i = 1 to n − l + 1              // chain begins at A_i
          j = i + l − 1                   // chain ends at A_j
          m[i, j] = ∞
          m[i, j] = ∞
          for k = i to j − 1
              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
              if q < m[i, j]
                  m[i, j] = q             // remember this cost
                  s[i, j] = k             // remember this index
  return m and s
```

All n chains of length 1 are initialized so that $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. Then $n - 1$ chains of length 2 are computed, then $n - 2$ chains of length 3, and so on, up to 1 chain of length n.

[We don't include an example here because the arithmetic is hard for students to process in real time.]

Time: $O(n^3)$, from triply nested loops. Also $\Omega(n^3) \Rightarrow \Theta(n^3)$.

Step 4: Construct an optimal solution

With the s table filled in, recursively print an optimal solution.

```
PRINT-OPTIMAL-PARENS(s, i, j)
  if i == j
      print "A"_i
  else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```

Initial call is PRINT-OPTIMAL-PARENS$(s, 1, n)$

Longest common subsequence

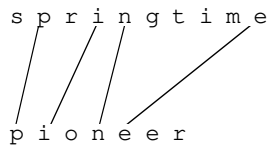
[The textbook has the section on elements of dynamic programming next, but these lecture notes reserve that section for the end of the chapter so that it may refer to two more examples of dynamic programming.]

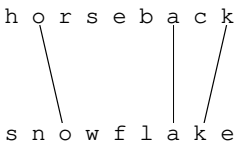
Problem: Given two sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

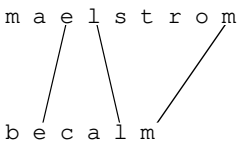
[To come up with examples of longest common subsequences, search the dictionary for all words that contain the word you are looking for as a subsequence. On a UNIX system, for example, to find all the words with `pine` as a subsequence, use the command `grep '.*p.*i.*n.*e.*' dict`, where `dict` is your local dictionary. Then check if that word is actually a longest common subsequence. Working C code for finding a longest common subsequence of two strings appears at http://www.cs.dartmouth.edu/~thc/code/lcs.c The comments in the code refer to the second edition of the textbook, but the code is correct.]

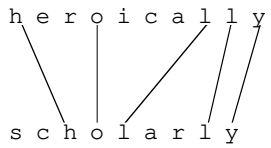
Examples

[The examples are of different types of trees.]









Brute-force algorithm:

For every subsequence of X, check whether it's a subsequence of Y.

Time: $\Theta(n2^m)$.

- 2^m subsequences of X to check.
- Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

Step 1: Characterize an LCS

Notation:

X_i = prefix $\langle x_1, \ldots, x_i \rangle$

Y_i = prefix $\langle y_1, \ldots, y_i \rangle$

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ 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 $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Proof

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = \langle z_1, \ldots, z_k, x_m \rangle$. It's a common subsequence of X and Y and has length $k + 1 \Rightarrow Z'$ is a longer common subsequence than $Z \Rightarrow$ contradicts Z being an LCS.

   Now show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. Clearly, it's a common subsequence. Now suppose there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence W' by appending x_m to W. W' is common subsequence of X and Y, has length $\geq k + 1$ $\Rightarrow$ contradicts Z being an LCS.
2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$. Then W is a common subsequence of X and $Y \Rightarrow$ contradicts Z being an LCS.
3. Symmetric to 2. ■ (theorem)

Therefore, an LCS of two sequences contains as a prefix an LCS of prefixes of the sequences.

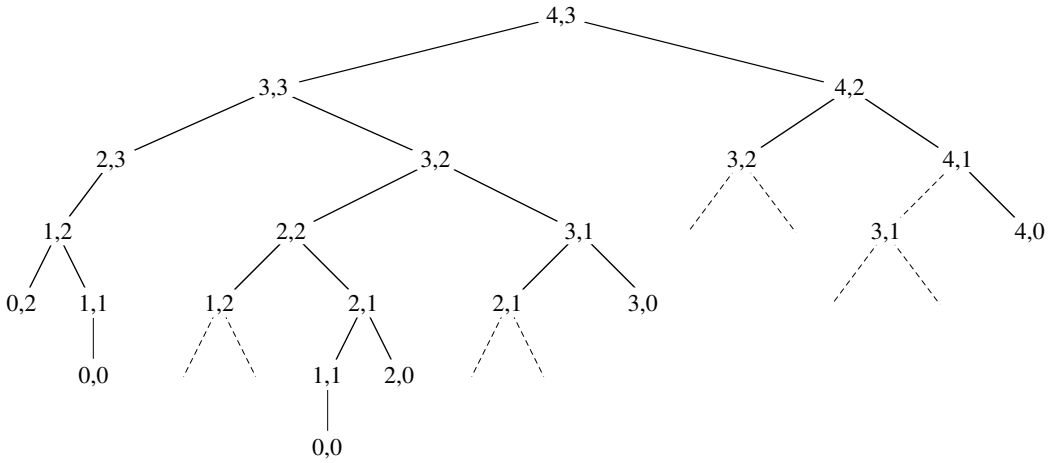
Step 2: Recursively define an optimal solution

Define $c[i, j]$ = length of LCS of X_i and Y_j. Want $c[m, n]$.

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

Again, could write a recursive algorithm based on this formulation.

Try with $X = \langle a, t, o, m \rangle$ and $Y = \langle a, n, t \rangle$. Numbers in nodes are values of i, j in each recursive call. Dashed lines indicate subproblems already computed.



- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

Step 3: Compute the length of an LCS

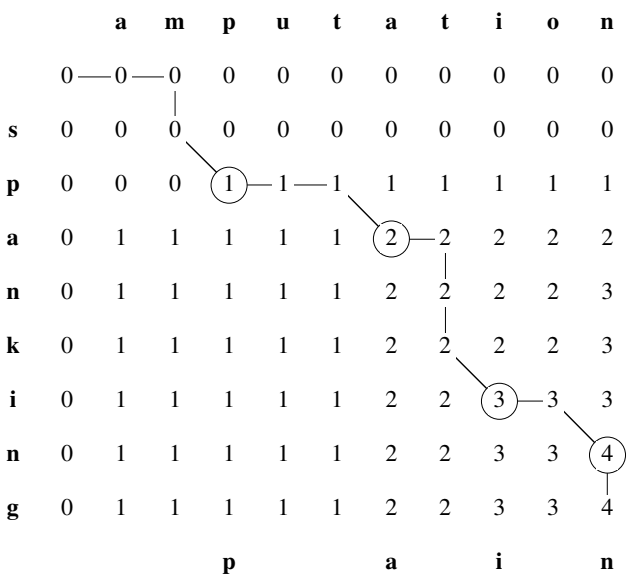
```
LCS-LENGTH(X, Y, m, n)
  let b[1:m, 1:n] and c[0:m, 0:n] be new tables
  for i = 1 to m
      c[i, 0] = 0
  for j = 0 to n
      c[0, j] = 0
  for i = 1 to m          // compute table entries in row-major order
      for j = 1 to n
          if x_i == y_j
              c[i, j] = c[i − 1, j − 1] + 1
              b[i, j] = "↖"
          else if c[i − 1, j] ≥ c[i, j − 1]
                  c[i, j] = c[i − 1, j]
                  b[i, j] = "↑"
          else c[i, j] = c[i, j − 1]
                  b[i, j] = "←"
  return c and b
```

```
PRINT-LCS(b, X, i, j)
  if i == 0 or j = 0
      return              // the LCS has length 0
  if b[i, j] == "↖"
      PRINT-LCS(b, X, i − 1, j − 1)
      print x_i           // same as y_j
  elseif b[i, j] == "↑"
      PRINT-LCS(b, X, i − 1, j)
  else PRINT-LCS(b, X, i, j − 1)
```

- Initial call is PRINT-LCS(b, X, m, n).
- $b[i, j]$ points to table entry whose subproblem was used in solving LCS of X_i and Y_j.
- When $b[i, j] = \nwarrow$, LCS extended by one character. So longest common subsequence = entries with $\nwarrow$ in them.

Demonstration

What do `spanking` and `amputation` have in common? *[Show only $c[i, j]$.]*



Answer: `pain`.

Time

$\Theta(mn)$

Improving the code

Don't really need the b table. $c[i, j]$ depends only on $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given $c[i, j]$, can determine in constant time which of the three values was used to compute $c[i, j]$. *[Exercise 14.4-2.]*

Or, if only need the length of an LCS, and don't need to construct the LCS itself, can get away with storing only one row of the c table plus a constant amount of additional entries. *[Exercise 14.4-4.]*

Optimal binary search trees

- Given sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys, sorted ($k_1 < k_2 < \cdots < k_n$).
- Want to build a binary search tree from the keys.
- For k_i, have probability p_i that a search is for k_i.
- Want BST with minimum expected search cost.
- Actual cost = # of items examined.

  For key k_i, cost = $\text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T.

E [search cost in T]

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

		j					
	w	0	1	2	3	4	5
	1	0	.25	.45	.5	.7	1.0
	2		0	.2	.25	.45	.75
	3			0	.05	.25	.55
i	4				0	.2	.5
	5					0	.3
	6						0

		j				
	root	1	2	3	4	5
	1	1	1	1	2	2
	2		2	2	2	4
i	3			3	4	5
	4				4	5
	5					5

Time

$O(n^3)$: for loops nested 3 deep, each loop index takes on $\leq n$ values. Can also show $\Omega(n^3)$. Therefore, $\Theta(n^3)$.

Step 4: Construct an optimal binary search tree

[Exercise 14.5-1 asks to write this pseudocode.]

```
CONSTRUCT-OPTIMAL-BST(root)
  r = root[1, n]
  print "k"_r "is the root"
  CONSTRUCT-OPT-SUBTREE(1, r − 1, r, "left", root)
  CONSTRUCT-OPT-SUBTREE(r + 1, n, r, "right", root)

CONSTRUCT-OPT-SUBTREE(i, j, r, dir, root)
  if i ≤ j
      t = root[i, j]
      print "k"_t "is" dir "child of k"_r
      CONSTRUCT-OPT-SUBTREE(i, t − 1, t, "left", root)
      CONSTRUCT-OPT-SUBTREE(t + 1, j, t, "right", root)
```

Elements of dynamic programming

Mentioned already:

- optimal substructure
- overlapping subproblems

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution. *[We find that students often have trouble understanding the relationship between optimal substructure and determining which choice is made in an optimal solution. One way that helps them understand optimal substructure is to imagine that the dynamic-programming gods tell you what was the last choice made in an optimal solution.]*
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste:
  - Suppose that one of the subproblem solutions is not optimal.
  - *Cut* it out.
  - *Paste* in an optimal solution.
  - Get a better solution to the original problem. Contradicts optimality of problem solution.

That was optimal substructure.

Need to ensure that you consider a wide enough range of choices and subproblems that you get them all. *[The dynamic-programming gods are too busy to tell you what that last choice really was.]* Try all the choices, solve all the subproblems resulting from each choice, and pick the choice whose solution, along with subproblem solutions, is best.

How to characterize the space of subproblems?

- Keep the space as simple as possible.
- Expand it as necessary.

Examples

Rod cutting

- Space of subproblems was rods of length $n - i$, for $1 \le i \le n$.
- No need to try a more general space of subproblems.

Matrix-chain multiplication

- Suppose we had tried to constrain the space of subproblems to parenthesizing $A_1 A_2 \cdots A_j$.
- An optimal parenthesization splits at some matrix A_k.
- Get subproblems for $A_1 \cdots A_k$ and $A_{k+1} \cdots A_j$.
- Unless we could guarantee that $k = j - 1$, so that the subproblem for $A_{k+1} \cdots A_j$ has only A_j, then this subproblem is *not* of the form $A_1 A_2 \cdots A_j$.
- Thus, needed to allow the subproblems to vary at both ends—allow both i and j to vary.

Longest commmon subsequence

- Space of subproblems for $\langle x_1, \ldots, x_i \rangle$ and $\langle y_1, \ldots, y_j \rangle$ was just $\langle x_1, \ldots, x_{i-1} \rangle$ and $\langle y_1, \ldots, y_{j-1} \rangle$.
- No need to try a more general space of subproblems.

Optimal binary search trees

- Similar to matrix-chain multiplication.
- Suppose we had tried to constrain space of subproblems to subtrees with keys $k_1, k_2, \ldots, k_j$.
- An optimal BST would have root k_r, for some $1 \le r \le j$.
- Get subproblems $k_1, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$.
- Unless we could guarantee that $r = j$, so that subproblem with $k_{r+1}, \ldots, k_j$ is empty, then this subproblem is *not* of the form $k_1, k_2, \ldots, k_j$.
- Thus, needed to allow the subproblems to vary at "both ends," i.e., allow both i and j to vary.

Optimal substructure varies across problem domains:

1. *How many subproblems* are used in an optimal solution.
2. *How many choices* in determining which subproblem(s) to use.

- Rod cutting:
  - 1 subproblem (of size $n - i$)
  - n choices
- Matrix-chain multiplication:
  - 2 subproblems ($A_i \cdots A_k$ and $A_{k+1} \cdots A_j$)
  - $j - i$ choices for A_k in $A_i, A_{i+1}, \ldots, A_{j-1}$. Having found optimal solutions to subproblems, choose from among the $j - i$ candidates for A_k.
- Longest common subsequence:
  - 1 subproblem
  - Either
    - 1 choice (if $x_i = y_j$, LCS of X_{i-1} and Y_{j-1}), or
    - 2 choices (if $x_i \ne y_j$, LCS of X_{i-1} and Y, and LCS of X and Y_{j-1})
- Optimal binary search tree:
  - 2 subproblems ($k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$)
  - $j - i + 1$ choices for k_r in $k_i, \ldots, k_j$. Having found optimal solutions to subproblems, choose from among the $j - i + 1$ candidates for k_r.

Informally, running time depends on (# of subproblems overall) $\times$ (# of choices).

- Rod cutting: $\Theta(n)$ subproblems, $\le n$ choices for each
  $\Rightarrow O(n^2)$ running time.
- Matrix-chain multiplication: $\Theta(n^2)$ subproblems, $O(n)$ choices for each
  $\Rightarrow O(n^3)$ running time.

- Longest common subsequence: $\Theta(mn)$ subproblems, ≤ 2 choices for each $\Rightarrow \Theta(mn)$ running time.
- Optimal binary search tree: $\Theta(n^2)$ subproblems, $O(n)$ choices for each $\Rightarrow O(n^3)$ running time.

Can use the subproblem graph to get the same analysis: count the number of edges.

- Each vertex corresponds to a subproblem.
- Choices for a subproblem are vertices that the subproblem has edges going to.
- For rod cutting, subproblem graph has n vertices and $\leq n$ edges per vertex $\Rightarrow O(n^2)$ running time.
  In fact, can get an exact count of the edges: for $i = 0, 1, \ldots, n$, vertex for subproblem size i has out-degree $i \Rightarrow$ # of edges $= \sum_{i=0}^{n} i = n(n+1)/2$.
- Subproblem graph for matrix-chain multiplication has $\Theta(n^2)$ vertices, each with degree $\leq n - 1$
  $\Rightarrow O(n^3)$ running time.

Dynamic programming uses optimal substructure *bottom up*.

- *First* find optimal solutions to subproblems.
- *Then* choose which to use in optimal solution to the problem.

When we look at greedy algorithms, we'll see that they work *top down*: *first* make a choice that looks best, *then* solve the resulting subproblem.

Don't be fooled into thinking optimal substructure applies to all optimization problems. It doesn't.

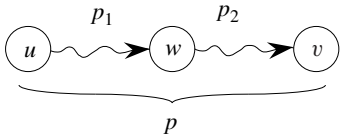
Here are two problems that look similar. In both, we're given an *unweighted, directed graph* $G = (V, E)$.

- V is a set of *vertices*.
- E is a set of *edges*.

And we ask about finding a ***path*** (sequence of connected edges) from vertex u to vertex v.

- ***Shortest path***: find a path $u \rightsquigarrow v$ with fewest edges. Must be ***simple*** (no *cycles*), since removing a cycle from a path gives a path with fewer edges.
- ***Longest simple path***: find a *simple* path $u \rightsquigarrow v$ with most edges. If didn't require simple, could repeatedly traverse a cycle to make an arbitrarily long path.

Shortest path has optimal substructure.



- Suppose p is shortest path $u \rightsquigarrow v$.
- Let w be any vertex on p.
- Let p_1 be the portion of p going $u \rightsquigarrow w$.
- Then p_1 is a shortest path $u \rightsquigarrow w$.

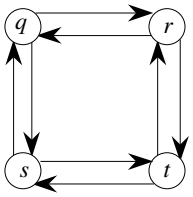
Proof Suppose there exists a shorter path p_1' going $u \rightsquigarrow w$. Cut out p_1, replace it with p_1', get path $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ with fewer edges than p. ■

Therefore, can find shortest path $u \rightsquigarrow v$ by considering all intermediate vertices w, then finding shortest paths $u \rightsquigarrow w$ and $w \rightsquigarrow v$.

Same argument applies to p_2.

Does longest path have optimal substructure?

- It seems like it should.
- It does *not*.



Consider $q \rightarrow r \rightarrow t$ = longest path $q \rightsquigarrow t$. Are its subpaths longest paths?

No!

- Subpath $q \rightsquigarrow r$ is $q \rightarrow r$.
- Longest simple path $q \rightsquigarrow r$ is $q \rightarrow s \rightarrow t \rightarrow r$.
- Subpath $r \rightsquigarrow t$ is $r \rightarrow t$.
- Longest simple path $r \rightsquigarrow t$ is $r \rightarrow q \rightarrow s \rightarrow t$.

Not only isn't there optimal substructure, but can't even assemble a legal solution from solutions to subproblems.

Combine longest simple paths:

$q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$

Not simple!

In fact, this problem is NP-complete (so it probably has no optimal substructure to find.)

What's the big difference between shortest path and longest path?

- Shortest path has ***independent*** subproblems.
- Solution to one subproblem does not affect solution to another subproblem of the same problem.
- Longest simple path: subproblems are *not* independent.
- Consider subproblems of longest simple paths $q \rightsquigarrow r$ and $r \rightsquigarrow t$.
- Longest simple path $q \rightsquigarrow r$ uses s and t.
- Cannot use s and t to solve longest simple path $r \rightsquigarrow t$, since if you do, the path isn't simple.
- But you *have* to use t to find longest simple path $r \rightsquigarrow t$!

- Using resources (vertices) to solve one subproblem renders them unavailable to solve the other subproblem.

  [For shortest paths, for a shortest path $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, no vertex other than w can appear in p_1 and p_2. Otherwise, get a cycle.]

Independent subproblems in our examples:

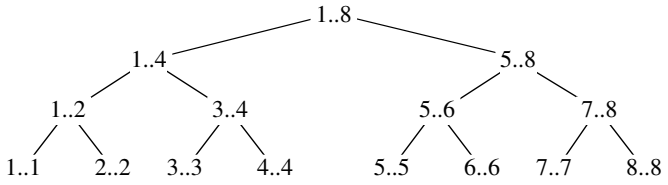
- Rod cutting and longest common subsequence
  - 1 subproblem $\Rightarrow$ automatically independent.
- Matrix-chain multiplication
  - $A_i \cdots A_k$ and $A_{k+1} \cdots A_j \Rightarrow$ independent.
- Optimal binary search tree
  - $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j \Rightarrow$ independent.

Overlapping subproblems

These occur when a recursive algorithm revisits the same problem over and over.

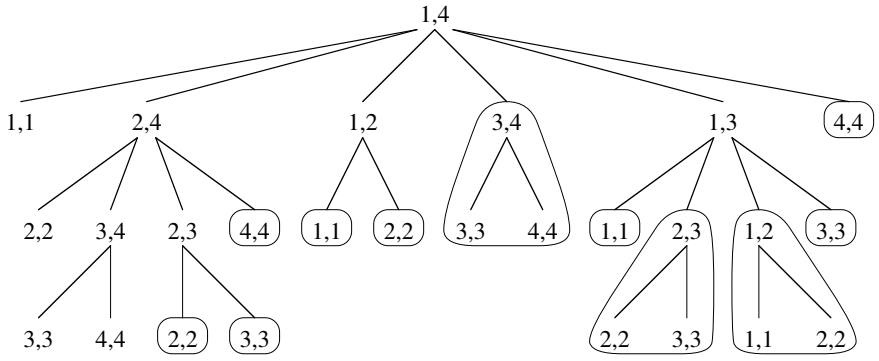
Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

Example: merge sort



Alternative approach to dynamic programming: ***memoization***

- "Store, don't recompute."
- Make a table indexed by subproblem.
- When solving a subproblem:
  - Lookup in table.
  - If answer is there, use it.
  - Else, compute answer, then store it.
- For matrix-chain multiplication:



Each node has the parameters i and j. Computations performed in highlighted subtrees are replaced by a single table lookup if computing recursively with memoization.

- In bottom-up dynamic programming, we go one step further. Determine in what order to access the table, and fill it in that way.

Chapter 15: Greedy algorithms

Reading: 15.1–15.3

Lecture Notes for Chapter 15: Greedy Algorithms

[The fourth edition removed the starred sections on matroids and task scheduling (an application of matroids). These sections were replaced by a new, unstarred section covering offline caching, which had been the subject of Problem 16-5 in the third edition.]

Chapter 15 overview

Similar to dynamic programming.

Used for optimization problems.

Idea

When you have a choice to make, make the one that looks best *right now*. Make a *locally optimal choice* in hope of getting a *globally optimal solution*.

Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions. We then study two other applications of the greedy method: Huffman coding and offline caching. *[Later chapters use the greedy method as well: minimum spanning tree, Dijkstra's algorithm for single-source shortest paths, and a greedy set-covering heuristic.]*

Activity selection

n ***activities*** require *exclusive* use of a common resource. For example, scheduling the use of a classroom.

Set of activities $S = \{a_1, \ldots, a_n\}$.

a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where $s_i =$ start time and $f_i =$ finish time.

Goal

Select the largest possible set of nonoverlapping (***mutually compatible***) activities.

Could have many other objectives:

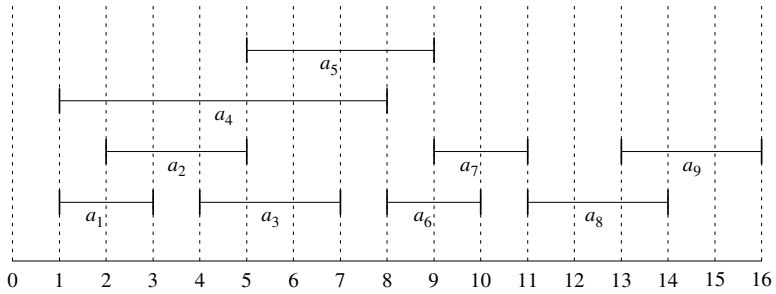
- Schedule room for longest time.
- Maximize income rental fees.

Assume that activities are sorted by finish time: $f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n$.

Example

S sorted by finish time: *[Leave on board]*

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



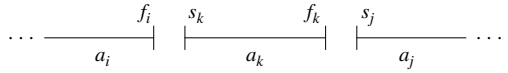
Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_1, a_3, a_6, a_9\}$, $\{a_1, a_3, a_7, a_8\}$, $\{a_1, a_3, a_7, a_9\}$, $\{a_1, a_5, a_7, a_8\}$, $\{a_1, a_5, a_7, a_9\}$, $\{a_2, a_5, a_7, a_8\}$, $\{a_2, a_5, a_7, a_9\}$.

Optimal substructure of activity selection

$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$ *[Leave on board]*

$=$ activities that start after a_i finishes and finish before a_j starts .



Activities in S_{ij} are compatible with

- all activities that finish by f_i, and
- all activities that start no earlier than s_j.

Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij}.

Let $a_k \in A_{ij}$ be some activity in A_{ij}. Then we have two subproblems:

- Find mutually compatible activities in S_{ik} (activities that start after a_i finishes and that finish before a_k starts).
- Find mutually compatible activities in S_{kj} (activities that start after a_k finishes and that finish before a_j starts).

Let

$A_{ik} = A_{ij} \cap S_{ik} =$ activities in A_{ij} that finish before a_k starts ,

$A_{kj} = A_{ij} \cap S_{kj} =$ activities in A_{ij} that start afer a_k finishes .

Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

$\Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

Claim

Optimal solution A_{ij} must include optimal solutions for the two subproblems for S_{ik} and S_{kj}.

Proof of claim Use the usual cut-and-paste argument. Will show the claim for S_{kj}; proof for S_{ik} is symmetric.

Suppose we could find a set A'_{kj} of mutually compatible activities in S_{kj}, where $|A'_{kj}| > |A_{kj}|$. Then use A'_{kj} instead of A_{kj} when solving the subproblem for S_{ij}. Size of resulting set of mutually compatible activities would be $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A|$. Contradicts assumption that A_{ij} is optimal. ■ (claim)

One recursive solution

Since optimal solution A_{ij} must include optimal solutions to the subproblems for S_{ik} and S_{kj}, could solve by dynamic programming.

Let $c[i, j] =$ size of optimal solution for S_{ij}. Then

$$c[i, j] = c[i, k] + c[k, j] + 1 \ .$$

But we don't know which activity a_k to choose, so we have to try them all:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \ , \\ \max \{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset \ . \end{cases}$$

Could then develop a recursive algorithm and memoize it. Or could develop a bottom-up algorithm and fill in table entries.

Instead, we will look at a greedy approach.

Making the greedy choice

Choose an activity to add to optimal solution *before* solving subproblems. For activity-selection problem, we can get away with considering only the greedy choice: the activity that leaves the resource available for as many other activities as possible.

Question: Which activity leaves the resource available for the most other activities? Answer: The first activity to finish. (If more than one activity has earliest finish time, can choose any such activity.)

Since activities are sorted by finish time, just choose activity a_1.

That leaves only one subproblem to solve: finding a maximum size set of mutually compatible activities that start after a_1 finishes. (Don't have to worry about activities that finish before a_1 starts, because $s_1 < f_1$ and no activity a_i has finish time $f_i < f_1 \Rightarrow$ no activity a_i has $f_i \leq s_1$.)

Since have only subproblem to solve, simplify notation:

$$S_k = \{a_i \in S : s_i \geq f_k\} = \text{activities that start after } a_k \text{ finishes} \ .$$

Making greedy choice of $a_1 \Rightarrow S_1$ remains as only subproblem to solve. *[Slight abuse of notation: referring to S_k not only as a set of activities but as a subproblem consisting of these activities.]*

By optimal substructure, if a_1 is in an optimal solution, then an optimal solution to the original problem consists of a_1 plus all activities in an optimal solution to S_1.

But need to prove that a_1 is always part of some optimal solution.

Theorem

If S_k is nonempty and a_m has the earliest finish time in S_k, then a_m is included in some optimal solution.

Proof Let A_k be an optimal solution to S_k, and let a_j have the earliest finish time of any activity in A_k. If $a_j = a_m$, done. Otherwise, let $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but with a_m substituted for a_j.

Claim

Activities in A'_k are disjoint.

Proof of claim Activities in A_k are disjoint, a_j is first activity in A_k to finish, and $f_m \le f_j$. ■ (claim)

Since $|A'_k| = |A_k|$, conclude that A'_k is an optimal solution to S_k, and it includes a_m. ■ (theorem)

So, don't need full power of dynamic programming. Don't need to work bottom-up.

Instead, can just repeatedly choose the activity that finishes first, keep only the activities that are compatible with that one, and repeat until no activities remain.

Can work top-down: make a choice, then solve a subproblem. Don't have to solve subproblems before making a choice.

Recursive greedy algorithm

Start and finish times are represented by arrays s and f, where f is assumed to be already sorted in monotonically increasing order.

To start, add fictitious activity a_0 with $f_0 = 0$, so that $S_0 = S$, the entire set of activities.

Procedure RECURSIVE-ACTIVITY-SELECTOR takes as parameters the arrays s and f, index k of current subproblem, and number n of activities in the original problem.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
  m = k + 1
  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
      m = m + 1
  if m ≤ n
      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
  else return ∅
```

Initial call

RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Idea

The **while** loop checks $a_{k+1}, a_{k+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_k (need $s_m \geq f_k$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve S_m, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

Go through example given earlier. Should get $\{a_1, a_3, a_6, a_8\}$.

Time

$\Theta(n)$—each activity examined exactly once, assuming that activities are already sorted by finish times.

Iterative greedy algorithm

Can convert the recursive algorithm to an iterative one. It's already almost tail recursive.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
  A = {a_1}
  k = 1
  for m = 2 to n
      if s[m] ≥ f[k]          // is a_m in S_k?
          A = A ∪ {a_m}       // yes, so choose it
          k = m               // and continue from there
  return A
```

Go through example given earlier. Should again get $\{a_1, a_3, a_6, a_8\}$.

Time

$\Theta(n)$, if activities are already sorted by finish times.

For both the recursive and iterative algorithms, add $O(n \lg n)$ time if activities need to be sorted.

Elements of the greedy strategy

The choice that seems best at the moment is the one we go with.

What did we do for activity selection?

1. Determine the optimal substructure.

2. Develop a recursive solution.
3. Show that if you make the greedy choice, only one subproblem remains.
4. Prove that it's always safe to make the greedy choice.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.

At first, it looked like dynamic programming. In the activity-selection problem, we started out by defining subproblems S_{ij}, where both i and j varied. But then found that making the greedy choice allowed us to restrict the subproblems to be of the form S_k.

Could instead have gone straight for the greedy approach: in our first crack at defining subproblems, use the S_k form. Could then have proven that the greedy choice a_m (the first activity to finish), combined with optimal solution to the remaining compatible activities S_m, gives an optimal solution to S_k.

Typically, we streamline these steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

No general way to tell whether a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.

Greedy-choice property

Can assemble a globally optimal solution by making locally optimal (greedy) choices.

Dynamic programming

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Solve *bottom-up* (unless memoizing).

Greedy

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*.

Typically show the greedy-choice property by what we did for activity selection:

- Look at an optimal solution.
- If it includes the greedy choice, done.
- Otherwise, modify the optimal solution to include the greedy choice, yielding another solution that's just as good.

Can get efficiency gains from greedy-choice property.

- Preprocess input to put it into greedy order.
- Or, if dynamic data, use a priority queue.

Optimal substructure

Just show that optimal solution to subproblem and greedy choice $\Rightarrow$ optimal solution to problem.

Greedy vs. dynamic programming

The knapsack problem is a good example of the difference.

0-1 knapsack problem

- n items.
- Item i is worth $\$v_i$, weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- Have to either take an item or not take it—can't take part of it.

Fractional knapsack problem

Like the 0-1 knapsack problem, but can take fraction of an item.

Both have optimal substructure.

But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.

To solve the fractional problem, rank items by value/weight: v_i/w_i. Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i. Take items in decreasing order of value/weight. Will take all of the items with the greatest value/weight, and possibly a fraction of the next item.

```
FRACTIONAL-KNAPSACK(v, w, W)
  load = 0
  i = 1
  while load < W and i ≤ n
      if w_i ≤ W − load
          take all of item i
      else take (W − load)/w_i of item i
      add what was taken to load
      i = i + 1
```

Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

Greedy doesn't work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50$.

Greedy solution:

- Take items 1 and 2.
- value = 160, weight = 30.

Have 20 pounds of capacity left over.

Optimal solution:

- Take items 2 and 3.
- value = 220, weight = 50.

No leftover capacity.

Huffman codes

Goal: Compress a data file made up of characters. You know how often each character appears in the file—its ***frequency***. Each character is represented by some bit sequence: a ***codeword***. Use as few bits as possible to represent the file.

Fixed-length code: All codewords have the same number of bits. For $n \geq 2$ characters, need $\lceil \lg n \rceil$ bits.

Variable-length code: Represent different characters with differing numbers of bits. In particular, give frequently occurring characters shorter codewords and infrequently occurring characters longer codewords.

Example: For a data file of 100,000 characters:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

For a fixed-length code, need 3 bits per character. For 100,000 characters, need 300,000 bits. For this variable-length code, need

$$\begin{aligned} 45{,}000 \cdot 1 &= 45{,}000 \\ +\ 13{,}000 \cdot 3 &= 39{,}000 \\ +\ 12{,}000 \cdot 3 &= 36{,}000 \\ +\ 16{,}000 \cdot 3 &= 48{,}000 \\ +\ 9{,}000 \cdot 4 &= 36{,}000 \\ +\ 5{,}000 \cdot 4 &= 20{,}000 \\ \hline &= 224{,}000 \text{ bits} \end{aligned}$$

Prefix-free codes

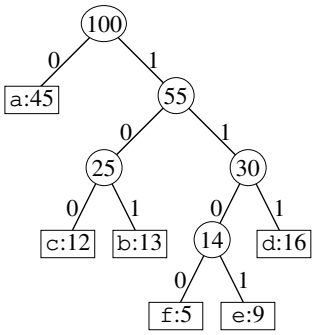
No codeword is also a prefix of any other codeword. *[Called "prefix codes" in earlier editions of the book. Changed to "prefix-free codes" in the fourth edition because each codeword is free of prefixes of other codes.]* A prefix-free code can always achieve the optimal compression.

Encoding: Just concatenate codewords for each character in the file. ***Example:*** To encode `face`: $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where $\cdot$ is concatenation.

Decoding: Since no codeword is a prefix of any other codeword, just process bits until you get a match. Then discard the bits and go from the rest of the compressed file. ***Example:*** If encoding is 100011001101, get a match on $100 =$ `c`. That leaves 011001101. Get a match on $0 =$ `a`. That leaves 11001101. Get a match on $1100 =$ `f`. That leaves 1101. Get a match on $1101 =$ `e`. So the encoded file represents `cafe`.

Binary tree representation

Use a binary tree whose leaves are the characters. The codeword for a character is given by the simple path from the root down to that character's leaf, where going left is 0 and going right is 1.



Here, each leaf has its character and frequency (in thousands). Each internal node holds the sum of the frequencies of the leaves in its subtree.

An optimal code is always given by a full binary tree: each internal node has 2 children $\Rightarrow$ if C is the alphabet for the characters, then the tree has $|C|$ leaves and $|C| - 1$ internal nodes.

How to compute the number of bits to encode a file for alphabet C given tree T: For each character $c \in C$, denote its frequency by $c.freq$. Denote the depth of c in T by $d_T(c)$, which equals the length of c's codeword. Then the number of bits to encode the file, the ***cost*** of T, is

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \ .$$

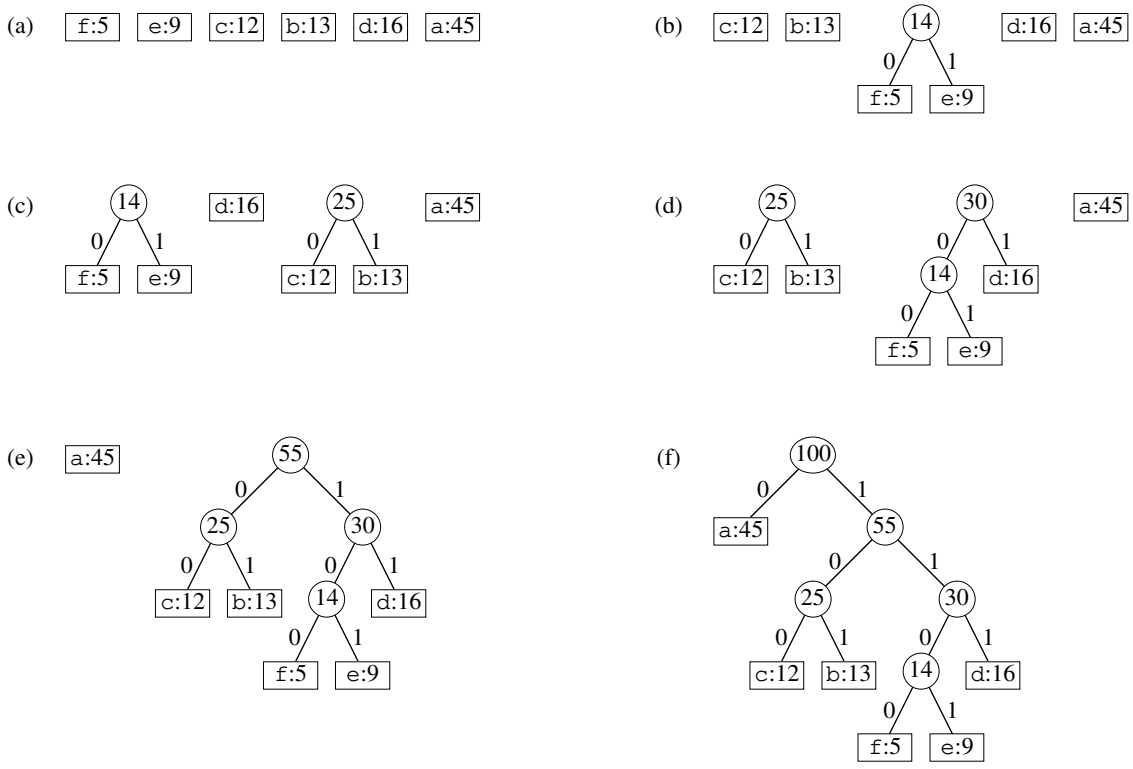
Constructing a Huffman code

[Named after David Huffman.] The algorithm builds tree T bottom-up. It repeatedly selects two nodes with the lowest frequency and makes them children of

a new node whose frequency is the sum of the two nodes' frequencies. It uses a min-priority queue Q keyed on the *freq* attribute, which all nodes have.

```
HUFFMAN(C)
  n = |C|
  Q = C
  for i = 1 to n − 1
      allocate a new node z
      x = EXTRACT-MIN(Q)
      y = EXTRACT-MIN(Q)
      z.left = x
      z.right = y
      z.freq = x.freq + y.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

Example: Using the frequencies from before:



Running time: Let $n = |C|$. The running time depends on how the min-priority queue Q is implemented. If with a binary min-heap, can initialize Q in $O(n)$ time. The **for** loop runs $n - 1$ times, and each INSERT and EXTRACT-MIN call takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$ time in all.

Correctness

Show the greedy-choice and optimal-substructure properties.

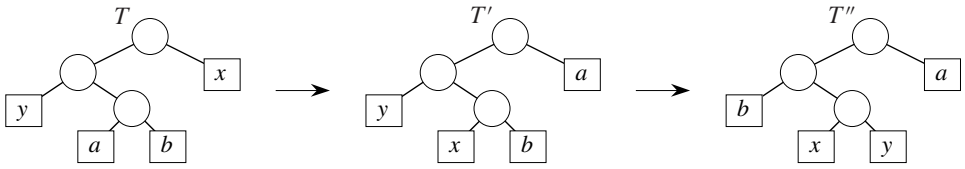
Lemma (Greedy-choice property)
For alphabet C, let x and y be the two characters with the lowest frequencies. Then there exists an optimal prefix-free code for C where the codewords for x and y have the same length and differ only in the last bit.

Proof Given a tree T for some optimal prefix-free code, modify it so that x and y are sibling leaves of maximum depth. Then the codewords for x and y will have the same length and differ in the last bit.

Let a, b be two characters that are sibling leaves of maximum depth in T. Assume wlog that $a.freq \le b.freq$ and $x.freq \le y.freq$. Must have $x.freq \le a.freq$ and $y.freq \le b.freq$.

Could have $x.freq = a.freq$ or $y.freq = b.freq$. If $x.freq = b.freq$, then $a.freq = b.freq = x.freq = y.freq$ (Exercise 15.3-1), and the lemma is trivially true. So assume that $x.freq \ne b.freq \Rightarrow x \ne b$.

In T: exchange a and x, producing T'.
In T': exchange b and y, producing T''.
In T'', x and y are sibling leaves of maximum depth.



Claim
$B(T') \le B(T)$. (Exchanging a and x does not increase the cost.)

Proof of claim

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \ge 0 \,.
\end{aligned}$$

The last line follows because $x.freq \le a.freq$ and a is a maximum-depth leaf $\Rightarrow$ $d_T(a) \ge d_T(x)$. ■ (claim)

Similarly, $B(T'') \le B(T')$ because exchanging y and b doesn't increase the cost. Therefore, $B(T'') \le B(T') \le B(T)$. T is optimal $\Rightarrow B(T) \le B(T'') \Rightarrow B(T'') = B(T) \Rightarrow T''$ is optimal, and x and y are sibling leaves of maximum depth. ■

The lemma shows that to build up an optimal tree, can begin with the greedy choice of merging the two characters with lowest frequency. Greedy because the cost of a merger is the sum of the frequencies of its children and the cost of a tree equals the sum of the costs of its mergers (Exercise 15.3-4).

Lemma (Optimal-substructure property)

For alphabet C, let x, y be the two characters with minimum frequency. Let $C' = (C - \{x, y\}) \cup z$ for a new character z with $z.freq = x.freq + y.freq$. Let T' be a tree representing an optimal prefix-free code for C', and T be T' with the leaf for z replaced by an internal node with children x and y. Then T represents an optimal prefix-free code for C.

Proof $c \in C - \{x, y\} \Rightarrow d_T(c) = d_{T'}(c) \Rightarrow c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. $d_T(x) = d_T(y) = d_{T'}(z) + 1 \Rightarrow$

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) , \end{aligned}$$

so that $B(T) = B(T') + x.freq + y.freq$, which is equivalent to $B(T') = B(T) - x.freq - y.freq$.

Now suppose T doesn't represent an optimal prefix-free code for C. Then $B(T'') < B(T)$ for some optimal tree T''. By the previous lemma, without loss of generality, T'' has x and y as siblings. Replace the common parent of x and y by a leaf z with $z.freq = x.freq + y.freq$ and call the resulting tree T'''. Then,

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

so that T' was not optimal, a contradiction. ■

Theorem

HUFFMAN produces an optimal prefix-free code.

Proof The greedy-choice and optimal-substructure properties both apply. ■

Offline caching

In a computer, a ***cache*** is memory that is smaller but faster than main memory. It holds a small subset of what's in main memory. Caches store data in ***blocks***, also known as ***cache lines***, usually 32, 64, or 128 bytes. *[We use the term* blocks *in this discussion, rather than* cache lines.*]*

A program makes a sequence of memory requests to blocks. Each block usually has several requests to some data that it holds.

The cache size is limited to k blocks, starting out empty before the first request. Each request causes either 0 or 1 block to enter the cache, and either 0 or 1 block to be evicted. A request for block b may have one of three outcomes:

1. b is already in the cache due to some previous request $\Rightarrow$ ***cache hit***. The cache remains unchanged.
2. b is not already in the cache, but the cache is not yet full (contains $< k$ blocks). b goes into the cache, so that the cache now contains one more block than before the request.