References

There are very many algorithms textbooks in print.
The one I shall regularly refer to (as **CLRS**) is:

Introduction to Algorithms (Second Edition)
by Cormen, Leiserson, Rivest and Stein,
The MIT Press, 2001.

Two others worth exploring for this course:

Fundamentals of Algorithmics
by Brassard and Bratley,
Prentice Hall, 1996.

Algorithmics: The Spirit of Computing (Third Edition)
by Harel and Feldman,
Addison Wesley, 2004.

Readings from CLRS

Introduction (Slides 3-17)
Chapters 1-3

Divide-and-Conquer (Slides 18–41)
Chapter 4 (not Section 4.4).
Chapter 28, Section 28.2.
Chapter 33, Section 33.4.

Greedy Algorithms (Slides 42–62)
Chapter 16, Sections 16.1–16.3.
Chapter 23, Section 23.2 (pp 567-570).

Dynamic Programming (Slides 63–80)
Chapter 15.
Chapter 25, pp 620-622 and Section 25.2.

Mathematical Functions

I shall assume you are comfortable with standard math functions, like exponentiation b^x and its inverse $\log_b x$.

$\log_b a$ is the number x such that $b^x = a$.

We shall usually work with binary logarithms $\lg x = \log_2 x$.

Some Useful Identities

$$\log_b(xy) = \log_b x + \log_b y \qquad \log_b(x^y) = y \log_b x \qquad \frac{\log_b x}{\log_c x} = \log_b c$$

We shall often use ***floor*** $\lfloor x \rfloor$ and ***ceiling*** $\lceil x \rceil$ functions:

$\lfloor x \rfloor$ is the largest integer $\leq x$, e.g., $\lfloor 5.3 \rfloor = 5$
$\lceil x \rceil$ is the smallest integer $\geq x$, e.g., $\lceil 5.3 \rceil = 6$

as well as summation notation:

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_2 + \cdots + a_n.$$

Basic Definitions

Model of Computation: An abstract sequential computer called a *Random Access Machine (RAM)*.

Computational Problem: A specification in general terms of *inputs* and *outputs* and the desired input/output relationship.

Problem Instance: An actual set of inputs for a given problem.

Algorithm: A method of solving a problem which can be implemented on a computer (in particular, a RAM).

- A *program* is a particular *implementation* of some algorithm.
  A program is <u>not</u> the same as an algorithm.
  (In this course, you shall not be implementing any algorithms.)
- There will always be many different algorithms for any given problem.

How to describe an algorithm

Pseudocode notation

- Similar to any typical imperative programming language, such as Pascal, C, Modula, ...
- Liberal use of English.
- Use of indentation for block structure.
- Employs any clear and concise expressive methods.
- Typically not concerned with software engineering issues such as:
  - ∗ error handling.
  - ∗ data abstraction.
  - ∗ modularity.

An Example Algorithm

Problem: Sorting (numbers).
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots a'_n \rangle$ of the input such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

```
INSERTION-SORT(A)
for j ← 2 to length(A) do
    key ← A[j]
    ▷ Insert A[j] into sorted sequence A[1..j−1].
    i ← j−1
    while i > 0 and A[i] > key do
        A[i+1] ← A[i]
        i ← i−1
    A[i+1] ← key
```


Algorithm Analysis

Predicting the amount of resource required from the *size* of the input.

We must have some quantity to count. Typically:

- runtime.
- memory.
- number of basic operations, such as:
  - arithmetic operations (eg, for multiplying matrices).
  - bit operations (eg, for multiplying integers).
  - comparisons (eg, for sorting and searching).

Types of Analysis:

- worst-case.
- average-case.
- best-case.

Various Runtime Bounds

Given a *problem*, a function $T(n)$ reflecting its runtime (as a function of *input size*) is an:

Upper Bound: if there is an algorithm which solves the problem and has worst–case runtime $T(n)$.

Average–case bound: if there is an algorithm which solves the problem and has average–case runtime $T(n)$.

Lower Bound: if *every* algorithm which solves the problem must use at least $T(n)$ time on some instance of size n for infinitely many n.

Two Revealing Tables

Time to solve a problem instance of size n using a $T(n)$-time algorithm.

T(n)	$n=10$	$n=20$	$n=50$	$n=100$	$n=500$	$n=1000$
$n \lg n$	33 µs	86 µs	282 µs	664 µs	4.5 ms	10 ms
n^2	100 µs	400 µs	2.5 ms	10 ms	250 ms	1 s
2^n	1 ms	1 s	36 yr	10^{16} yr	10^{137} yr	10^{287} yr
$n!$	4 s	10^5 yr	10^{51} yr	10^{144} yr	10^{1120} yr	10^{2554} yr

Largest problem instance solvable in 1 minute.

$T(n)$		2× faster machine	10^3× faster machine
$n \lg n$	3.9×10^6	7.5×10^6	2.7×10^9
n^2	7 745	10 954	244 948
2^n	25	26	35
$n!$	11	11	13

Analyzing Algorithms: How *not* to do it

	INSERTION-SORT(A)	*cost*	*repetitions*
1	**for** $j \leftarrow 2$ **to** *length*(A) **do**	c_1	n
2	$\quad key \leftarrow A[j]$	c_2	$n-1$
3	$\quad \triangleright$ Insert $A[j]$ into sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4	$\quad i \leftarrow j-1$	c_4	$n-1$
5	$\quad$ **while** $i > 0$ **and** $A[i] > key$ **do**	c_5	$\sum_{j=2}^{n} t_j$
6	$\qquad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j-1)$
7	$\qquad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j-1)$
8	$\quad A[i+1] \leftarrow key$	c_8	$n-1$

where:

- $n = \mathit{length}(A)$, and
- t_j = number of times the **while**-loop test in line 5 is executed in the jth iteration of the **for**-loop.

The Exact Analysis

Best possible situation: $t_j = 1$ for all j, i.e., when A is already sorted.

Runtime: $$(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Worst possible situation: $t_j = j$ for all j, i.e., when A is sorted in *reverse* order.

Runtime: $$\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8).$$

Average situation: $t_j = \frac{j}{2}$ for all j, i.e., every permutation is equally likely, so expected value of t_j is $\frac{j}{2}$.

Runtime: $$\left(\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{4} - \frac{3c_6}{4} - \frac{3c_7}{4} + c_8\right)n - \left(c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right).$$

Growth of Functions

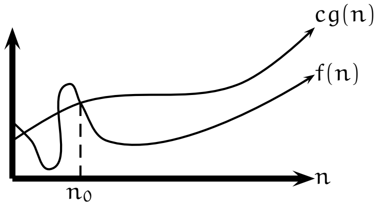
We consider only functions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$.

O-notation: $O(g(n))$ is the set of all functions $f(n)$ for which there are positive constants c and n_0 such that
$$f(n) \leq c g(n) \qquad \text{for all } n \geq n_0.$$

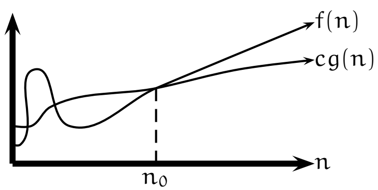
Ω-notation: $\Omega(g(n))$ is the set of all functions $f(n)$ for which there are positive constants c and n_0 such that
$$f(n) \geq c g(n) \qquad \text{for all } n \geq n_0.$$

Θ-notation: $\Theta(g(n))$ is the set of all functions $f(n)$ for which there are positive constants c_1, c_2 and n_0 such that
$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \text{for all } n \geq n_0.$$

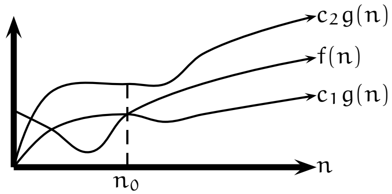
Picturing Asymptotic Growth



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

Useful (abuse of) notation

We write

$$f(n) \;=\; O(g(n))$$

to mean

$$f(n) \;\in\; O(g(n)).$$

Similarly for Ω and Θ.

Very useful, eg:

$$5n^3 + 2n - 4 \quad = \quad 5n^3 + \Theta(n) \quad = \quad \Theta(n^3).$$

Note:

$$\sum_{i=1}^{m} O(i^5) \quad \text{means} \quad \sum_{i=1}^{m} f(i) \quad \text{for some } f(n) \in O(n^5).$$

It does *not* mean $O(1^5) \;+\; O(2^5) \;+\; \cdots \;+\; O(m^5)$.

Example Analysis

```
INSERTION-SORT(A)
1  for j ← 2 to length(A) do
2     key ← A[j]
3     ▷ Insert A[j] into sorted sequence A[1..j−1].
4     i ← j−1
5     while i > 0 and A[i] > key do
6         A[i+1] ← A[i]
7         i ← i−1
8     A[i+1] ← key
```

The **for**-loop on line 1 is executed $O(n)$ times; and each statement costs constant time, except for the **while**-loop on lines 5-7 which costs $O(n)$.

Thus overall runtime is: $O(n) \times O(n) \;=\; O(n^2)$.

Note: In fact, the worst-case runtime is $\Theta(n^2)$.

Another Example

Problem: Evaluating Polynomials.
Input: A sequence $a = \langle a_0, a_1, \ldots, a_n \rangle$ of real values, and a real value x.
Output: The value of the nth-degree polynomial
$A(x) \;=\; a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$.

```
POLY-EVAL(a, x)
xi ← 1
result ← a_0
for i ← 1 to n do
    xi ← xi × x            ▷ xi = x^i
    result ← result + a_i × xi
return result
```

This naïve algorithm performs $2n$ multiplications and n additions.

Can we do better? Yes!

A Better Solution

Horner's Rule expresses the polynomial $A(x)$ as:

$$A(x) \;=\; a_0 + x(a_1 + \cdots + x(a_{n-1} + x(a_n))\cdots).$$

This gives rise to the following algorithm for computing $A(x)$.

HORNER(a, x)
1 *result* $\leftarrow a_n$
2 **for** $i \leftarrow n{-}1$ **downto** 0 **do**
3 *result* $\leftarrow$ *result* $\times\, x \;+\; a_i$
4 **return** *result*

This algorithm performs n additions and n multiplications.

Note: It can be shown that at least n additions and n multiplications are *necessary* in the worst case to evaluate $A(x)$ for *any* arithmetic-based algorithm: this is thus a lower bound on the problem.

Hence this algorithm is provably optimal.

Divide-and-Conquer

Principle: Divide a problem into simpler subproblems, and solve the subproblems recursively.

Example: Find the minimum *and* maximum of a list A of $n{>}0$ numbers.

NAIVE-MIN-MAX(A)
1 *least* $\leftarrow A[1]$
2 **for** $i \leftarrow 2$ **to** *length*(A) **do**
3 **if** $A[i] <$ *least* **then** *least* $\leftarrow A[i]$
4 *greatest* $\leftarrow A[1]$
5 **for** $i \leftarrow 2$ **to** *length*(A) **do**
6 **if** $A[i] >$ *greatest* **then** *greatest* $\leftarrow A[i]$
7 **return** (*least*, *greatest*)

The **for**-loop on line 2 makes $n{-}1$ comparisons, as does the **for**-loop on line 5, making a total of $2n{-}2$ comparisons.

Can we do better? Yes!

Divide-and-Conquer Min-Max

Initially called with MIN-MAX$\big(A, 1,$ *length*$(A)\big)$.

MIN-MAX(A, p, q)
1 **if** $p = q$ **then** **return** $(A[p], A[q])$
2 **if** $p = q - 1$ **then**
3 **if** $A[p] < A[q]$ **then** **return** $(A[p], A[q])$
4 **else** **return** $(A[q], A[p])$
5 $r \leftarrow \lfloor (p+q)/2 \rfloor$
6 $(min1, max1) \leftarrow$ MIN-MAX(A, p, r)
7 $(min2, max2) \leftarrow$ MIN-MAX$(A, r{+}1, q)$
8 **return** $\big(\min(min1, min2), \max(max1, max2)\big)$

Let $T(n)$ be the number of comparisons made by MIN-MAX(A, p, q), where $n = q{-}p{+}1$. Then $T(1) = 0$, $T(2) = 1$, and for $k > 2$:

$$T(k) \;=\; T(\lceil k/2 \rceil) \;+\; T(\lfloor k/2 \rfloor) \;+\; 2.$$

Solving the Min-Max Recurrence

Claim: $T(n) = \frac{3}{2}n - 2$ for $n = 2^k \geq 2$ a power of 2.

Proof: By induction on n.

Base case: true for $n{=}2$, as $T(2) = 1 = \frac{3}{2} \cdot 2 - 2$.

Induction step: assuming $T(\frac{n}{2}) = \frac{3}{2}(\frac{n}{2}) - 2$,

$$T(n) \;=\; 2T(\tfrac{n}{2}) + 2 \;=\; 2\left(\tfrac{3}{2}(\tfrac{n}{2}) - 2\right) + 2 \;=\; \tfrac{3}{2}n - 2 \qquad \square$$

Note: If we replace line 5 of the algorithm by $r \leftarrow p{+}1$, then the resulting runtime $T'(n)$ satisfies $T'(n) \;=\; \left\lceil \frac{3n}{2} \right\rceil - 2$ for all $n > 0$. (For example, $T'(6) = 7$ whereas $T(6) = 8$.)

Note: It can be shown that at least $\left\lceil \frac{3n}{2} \right\rceil - 2$ comparisons are *necessary* in the worst case to find the maximum and minimum of n numbers for *any* comparison-based algorithm: this is thus a lower bound on the problem.

Hence this (last) algorithm is provably optimal.

Another Example: Merge-Sort

MERGE-SORT(A, p, r)
if $p < r$ **then**
  $q \leftarrow \lfloor (p + r)/2 \rfloor$
  MERGE-SORT(A, p, q)
  MERGE-SORT$(A, q + 1, r)$
  MERGE(A, p, q, r)

The runtime $T(n)$, where $n = r - p + 1 > 1$, satisfies:

$$T(n) = 2T(n/2) + \Theta(n)$$

We can show that $T(n) = \Theta(n \lg n)$.

Note: It can be shown that $\Omega(n \lg n)$ comparisons are *necessary* in the worst case to sort n numbers for *any* comparison-based algorithm: this is thus an (asymptotic) lower bound on the problem.

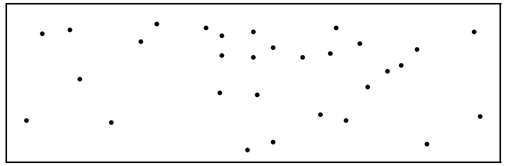
Hence this algorithm is provably (asymptotically) optimal.

Another Example: Closest-Points

Input: A set P of $n \geq 2$ points $p = (x, y)$.

Output: The two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ which are closest together, that is, which minimize

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

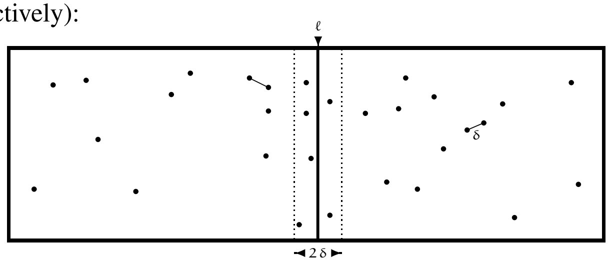


The naïve algorithm for computing this computes the distance between all $\frac{n(n-1)}{2}$ pairs, and thus runs in time $O(n^2)$.

Can we do better? Yes!

Divide-and-Conquer Closest Points

1. Sort the points P by their x coordinates, and recursively solve the problem for the points in the left and right halves (P_L and P_R, respectively):



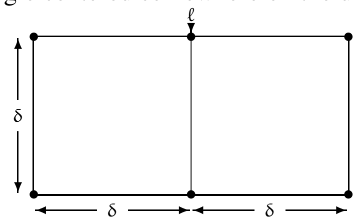
   If the minimum distance found on either side is δ, any closer pair of points must lie within the central 2δ strip.

2. Look in the central 2δ strip for a closer pair.

   It may be that *all* of the points lie in this strip; hence naïvely we may still need to check $n/2 \times n/2 = O(n^2)$ pairs.

The Conquer Step

If $p \in P_L$ and $q \in P_R$ have distance less than δ, then they must lie within some $2\delta \times \delta$ rectangle centered somewhere on the dividing line ℓ.



There can be at most 8 points in such a rectangle.

(The only way to get 4 points on either half with a distance of δ between each pair is to put them on the corners of the square. Hence we cannot fit more than 8 points into the rectangle.)

Hence for each point in the 2δ strip, we only need to compare its distance with the 7 points immediately above it in the strip.

The Closest-Pair Algorithm

```
CLOSEST-PAIR(P, X, Y)
 1  if P = {p} then return ∞
 2  if P = {p, q} (p ≠ q) then return d(p, q)
 3  Split P according to X into equal halves P_L and P_R
 4  Split X and Y likewise into X_L, X_R, Y_L and Y_R
 5  δ_L ← CLOSEST-PAIR(P_L, X_L, Y_L)
 6  δ_R ← CLOSEST-PAIR(P_R, X_R, Y_R)
 7  δ ← min(δ_L, δ_R)
 8  P′ ← { p ∈ Y : ℓ−δ ≤ p_x ≤ ℓ+δ }
 9  for i ← 0 to |P′|−2 do
10      for j ← i+1 to min(i+7, |P′|−1) do
11          δ ← min(δ, d(P′[i], P′[j]))
12  return δ
```

The program is invoked with the list of points P, along with the list sorted by x coordinates X and sorted by y coordinates Y.

Analysis of CLOSEST-PAIR

The running time $T(n)$ for CLOSEST-PAIR (given P sorted into X and Y):

3.	Split P into P_L and P_R	$O(n)$
		+
4.	Split X and Y into X_L, X_R and Y_L, Y_L	$O(n)$
		+
5-6.	Compute δ_L and δ_R	$2T(n/2)$
		+
8.	Create P'	$O(n)$
		+
9-11.	Compute minimum distance	$O(n)$

Hence, $T(n) = 2T(n/2) + O(n)$.

We can show that this gives $T(n) = O(n \lg n)$.

Yet Another Example: Matrix Multiplication

Input: Two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$.

Output: An $n \times n$ matrix $C = (c_{ij})$ where $C = AB$, ie, $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

The usual naïve algorithm is as follows.

```
MATRIX-MULT(A, B)
for i ← 1 to n do
    for j ← 1 to n do
        c_ij ← 0
        for k ← 1 to n do
            c_ij ← c_ij + a_ik × b_kj
```

This algorithm requires $\Theta(n^3)$ arithmetic operations.

Can we do better? Yes!

A Divide-and-Conquer Solution

Assume that n is a power of 2 (by padding out rows and columns with 0s).

Partition A, B and C into $\frac{n}{2} \times \frac{n}{2}$ matrices as such:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad B = \begin{pmatrix} e & g \\ f & h \end{pmatrix} \quad C = \begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} ae+bf & ag+bh \\ ce+df & cg+dh \end{pmatrix}$$

This reduces the $n \times n$ problem to 8 $\frac{n}{2} \times \frac{n}{2}$ problems, with an overhead of $4\left(\frac{n}{2}\right)^2$ additions.

The number $T(n)$ of arithmetic operations used to compute C is thus:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 8T(\frac{n}{2}) + n^2 & \text{if } n > 1. \end{cases}$$

We can show that $T(n) = \Theta(n^3)$.

Hence this is asymptotically no better than the naïve algorithm.

Strassen's Algorithm

Recursively form the following 7 products (first performing $10\left(\frac{n}{2}\right)^2$ additions):

$$
\begin{aligned}
P_1 &= (a+d)(e+h) & & \\
P_2 &= (c+d)e & P_5 &= (a+b)h \\
P_3 &= a(g-h) & P_6 &= (-a+c)(e+g) \\
P_4 &= d(-e+f) & P_7 &= (b-d)(f+h)
\end{aligned}
$$

Then with $8\left(\frac{n}{2}\right)^2$ more additions, we can compute:

$$
\begin{aligned}
r &= P_1 + P_4 - P_5 + P_7 \\
s &= P_3 + P_5 \\
t &= P_2 + P_4 \\
u &= P_1 + P_3 - P_2 + P_6
\end{aligned}
$$

These are easily verified. For example,

$$s = ag + bh = a(g-h) + (a+b)h = P_3 + P_5.$$

Analysis of Strassen's Algorithm

The number $T(n)$ of arithmetic operations used to compute C is thus:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 7T(\frac{n}{2}) + \frac{9}{2}n^2 & \text{if } n > 1. \end{cases}$$

We can show that $T(n) = \Theta(n^{\lg 7})$.

Note: $\lg 7 \approx 2.807$. Hence we have beaten the $O(n^3)$ upper bound.

Note: In a similar fashion, if we can multiply $k \times k$ matrices using m multiplications (not assuming commutativity of multiplication), then we can multiply $n \times n$ matrices in time $O(n^{\log_k m})$.

Note: The best known algorithm runs in time $O(n^{2.376})$.

Note: The best known asymptotic lower bound is $\Omega(n^2)$. (This is a trivial result, as there are n^2 entries c_{ij} to compute.)

Solving Recurrences: The Substitution Method

Guess an asymptotic bound and verify the guess with an induction proof.

For example, a good guess for

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

would be

$$T(n) = O(n \lg n).$$

We can then show by induction that for some appropriately-chosen positive constants c and n_0,

$$T(n) \le cn \lg n \quad \text{for all } n \ge n_0.$$

Note: The constants c and n_0 would need to be discovered in the process of carrying out the induction proof, and would rely on the boundary value $T(1)$.

Example

Given $T(n) = 4T(n/2) + n$, we might guess that $T(n) = O(n^3)$.

To demonstrate this, we want to show that

$$T(n) \le cn^3 \quad \text{for all } n \ge n_0,$$

where c and n_0 are as yet unspecified.

Assuming that $T(k) \le ck^3$ for all $k < n$,

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4c(n/2)^3 + n \\
&= \tfrac{c}{2}n^3 + n \\
&= cn^3 - \left(\tfrac{c}{2}n^3 - n\right) \\
&\le cn^3 \qquad \textit{if } c \ge 2 \textit{ and } n \ge 1.
\end{aligned}
$$

We also need $T(1) \le c \cdot 1^3$, so we set:

$$c = \max\left(2, T(1)\right) \quad \text{and} \quad n_0 = 1.$$

A Better Guess

In fact, $T(n) = O(n^2)$, but the proof is trickier.

Suppose we try to demonstrate that

$$T(n) \le cn^2 \qquad \text{for all } n \ge n_0,$$

where c and n_0 are as yet unspecified.

Assuming that $T(k) \le ck^2$ for all $k < n$,

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4c(n/2)^2 + n \\ &= cn^2 + n \\ &\not\le cn^2 \qquad \textit{for } \underline{\textit{any}} \quad c \ge 0. \end{aligned}$$

What went wrong?

We must *strengthen* the inductive hypothesis, by *subtracting* a lower-order term.

Another Attempt

We try instead to demonstrate that

$$T(n) \le cn^2 - dn \qquad \text{for all } n \ge n_0,$$

where c, d and n_0 are as yet unspecified.

Assuming that $T(k) \le ck^2 - dk$ for all $k < n$,

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4\left(c(n/2)^2 - d(n/2)\right) + n \\ &= cn^2 - 2dn + n \\ &= cn^2 - dn - (dn - n) \\ &\le cn^2 - dn \qquad \textit{if} \quad d \ge 1. \end{aligned}$$

So we can choose $d = 1$.

We also need $T(1) \le c \cdot 1^2 - 1 \cdot 1$, so we set:

$$c = \max(1, T(1) + 1) \qquad \text{and} \qquad n_0 = 1.$$

Solving Recurrences: The Iteration Method

Unfold the recurrence and look for a pattern.

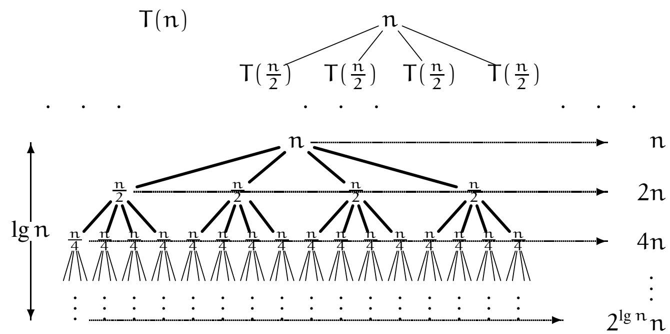
For our example, we might proceed as follows:

$$\begin{aligned} T(n) &= n + 4T(n/2) \\ &= n + 4(n/2 + 4T(n/4)) \\ &= n + 2n + 4^2(n/4 + 4T(n/8)) \\ &= n + 2n + 4n + 4^3(n/8 + 4T(n/16)) \\ &= n + 2n + \cdots + 2^{\lg n - 1}n + 4^{\lg n}T(1) \\ &= n(1 + 2 + 4 + \cdots + 2^{\lg n - 1}) + n^2\Theta(1) \\ &= n(2^{\lg n} - 1) + \Theta(n^2) \\ &= n^2 - n + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

Solving Recurrences: Recursion Trees

Draw the unfoldings of the recurrence

$$T(n) = n + 4T(n/2).$$



Total: $\Theta(2^{\lg n}n)$
$= \Theta(n^2)$

Master Theorem – Simplified Version –

Let $a \geq 1$ and $b > 1$ and $c \geq 0$ be constants.

Let $T(n)$ be defined by the recurrence

$$T(n) = aT(n/b) + \Theta(n^c),$$

where n/b represents either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

Then $T(n)$ is bounded asymptotically as follows:

1. If $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$.
2. If $c = \log_b a$ then $T(n) = \Theta(n^c \lg n)$.
3. If $c > \log_b a$ then $T(n) = \Theta(n^c)$.

(**General version:** CLRS, Thm 4.1, p73.)

Using the Master Theorem

- The runtime for MIN-MAX satisfies the recurrence:

  $T(n) = 2T(n/2) + \Theta(1)$.

  The Master Theorem (case 1) applies:

  $a = b = 2$ and $c = 0 < 1 = \log_b a$,

  giving $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

- The runtime for MERGE-SORT satisfies the recurrence:

  $T(n) = 2T(n/2) + \Theta(n)$.

  The Master Theorem (case 2) applies:

  $a = b = 2$ and $c = 1 = \log_b a$,

  giving $T(n) = \Theta(n^c \lg n) = \Theta(n \lg n)$.

More Examples: Matrix Multiplication

- The runtime for MATRIX-MULT satisfies the recurrence:

  $T(n) = 8T(n/2) + n^2$.

  The Master Theorem (case 1) applies:

  $a = 8$, $b = 2$ and $c = 2 < 3 = \log_b a$,

  giving $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$.

- Strassen's Algorithm satisfies the recurrence:

  $T(n) = 7T(n/2) + \Theta(n^2)$.

  The Master Theorem (case 1) applies:

  $a = 7$, $b = 2$ and $c = 2 < \log_b a \approx 2.801$,

  giving $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$.

What's Happening

For the recurrences:

$$T_1(n) = 4T(n/2) + n$$
$$T_2(n) = 4T(n/2) + n^2$$
$$T_3(n) = 4T(n/2) + n^3$$

The Master Theorem (case i) applies:

$a = 4$ and $b = 2$ (so $\log_b a = 2$), and $c = i$,

giving

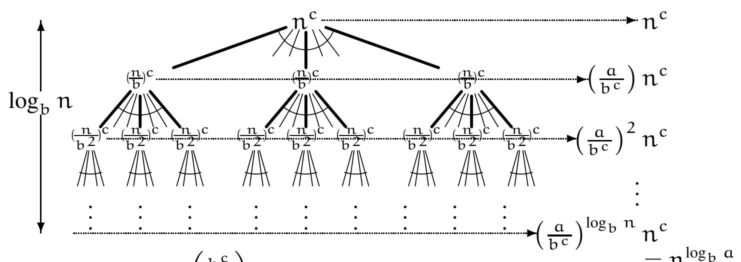
$T_1(n) = \Theta(n^2)$, $T_2(n) = \Theta(n^2 \lg n)$, and $T_3(n) = \Theta(n^3)$.

Case 1: applies if the overhead cost (n^c) is negligible compared to the number and size of the subproblems.

Case 2: applies if the overhead cost (n^c) is as costly as the subproblems.

Case 3: applies if the overhead cost (n^c) is the dominating factor.

The General Picture



1. If $c < \log_b a$ then $\left(\frac{b^c}{a}\right) < 1$, and

   $T(n) < n^{\log_b a} \left(1 + \left(\frac{b^c}{a}\right) + \left(\frac{b^c}{a}\right)^2 + \cdots\right) = n^{\log_b a} \left(\frac{1}{1-\left(\frac{b^c}{a}\right)}\right) = \Theta(n^{\log_b a}).$
2. If $c = \log_b a$ then $\left(\frac{a}{b^c}\right) = 1$, and

   $T(n) = n^c \log_b n = \Theta(n^c \log_b n).$
3. If $c > \log_b a$ then $\left(\frac{a}{b^c}\right) < 1$, and

   $T(n) < n^c \left(1 + \left(\frac{a}{b^c}\right) + \left(\frac{a}{b^c}\right)^2 + \cdots\right) = n^c \left(\frac{1}{1-\left(\frac{a}{b^c}\right)}\right) = \Theta(n^c).$

Greedy Algorithms

Idea: In order to find a globally optimal solution, repeatedly choose locally optimal solutions.

Example Problem: Making Change.

Input: A list of integers representing coin denominations, plus another positive integer representing an amount of money.

Output: A minimal collection of coins of the given denominations which sum to the given amount.

Greedy Strategy: Repeatedly include in the solution the largest coin whose value doesn't exceed the remaining amount.

E.g.: If the denominations are (25,10,5,1) and the amount is 87, then

$$87 \quad = \quad 25 + 25 + 25 + 10 + 1 + 1.$$

Applicability of Greedy Algorithms

Greedy Algorithms don't always work.

For example, in Making Change, if the denominations were (25,11,5,1), then

$$15 \quad = \quad 11 + 1 + 1 + 1 + 1 \quad = \quad 5 + 5 + 5.$$

However, quite often they do, or they come close enough to the optimal solution to make the outcome acceptable.

This, and the fact that they are quite easy to implement, make them an attractive alternative for many hard problems.

Well-known instances of the use of Greedy Algorithms are known for the following problems.

- Minimum Spanning Trees. (Kruskal's Algorithm.)
- Activity Selection.
- Data Compression. (Huffman Codes.)

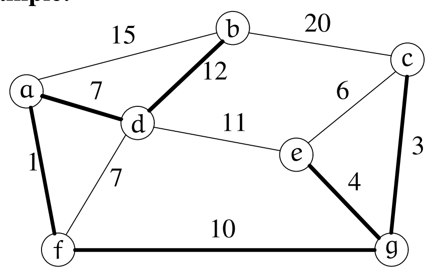
Minimum Spanning Trees (MST)

Input: An undirected connected graph $G = (V, E)$, and a (positive) weight $w(e) \in \mathbb{R}^+$ on each edge $e \in E$.

Output: A subset $F \subseteq E$ of edges which connects all of the vertices V of G, has no cycles, and minimizes the total edge weight:

$$w(F) \quad = \quad \sum_{e \in F} w(e).$$

Example:



Kruskal's Algorithm

The following classic algorithm due to Kruskal is a greedy algorithm.

KRUSKAL-MST(G, w)
1 sort edges E of G by nondecreasing weight w
2 $F \leftarrow \emptyset$
3 **for** each edge $e \in E$ **do**
4 **if** $F \cup \{e\}$ is acyclic **then**
5 $F \leftarrow F \cup \{e\}$
6 **return** F

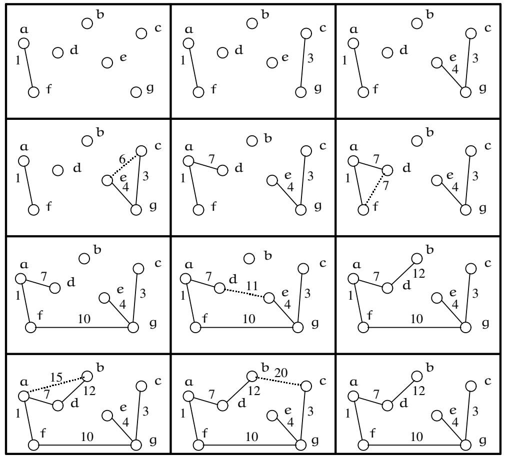
At each stage, the algorithm greedily chooses the cheapest edge and adds it to the partial solution F, provided it satisfies the acyclicity criterion.

The running time of the algorithm is dominated by the sorting of edges in line 1*. Hence, by using an optimal sorting algorithm, the running time of this algorithm is $\Theta(E \lg E)$.

*assuming we can do the test in line 4 efficiently.

Kruskal's Algorithm Illustrated

$$
\begin{aligned}
w(a,f) &= 1\\
w(a,d) &= 7\\
w(b,d) &= 12\\
w(c,g) &= 3\\
w(d,f) &= 7\\
w(a,b) &= 15\\
w(e,g) &= 4\\
w(f,g) &= 10\\
w(b,c) &= 20\\
w(c,e) &= 6\\
w(d,e) &= 11
\end{aligned}
$$

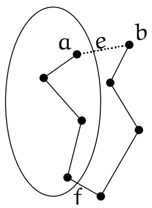


Correctness of Kruskal's Algorithm

Fact: At any time during the exectution of Kruskal's Algorithm, F is a subset of a MST.

Proof: By induction on $|F|$. This is clearly true (initially) when $F = \emptyset$.

Suppose $F \neq \emptyset$, and let $e = (a, b)$ be the most recently added edge. Let T be a MST which (by induction) contains $F - \{e\}$. Let f be the first edge on the (unique) path in T going from vertex a to vertex b which is not in $F - \{e\}$, and assume that $f \neq e$.



We must have $w(e) \leq w(f)$ (for otherwise f would have been included in F rather than e), and hence we could get another MST by replacing f by e in T. Hence F is a subset of a MST. $\square$

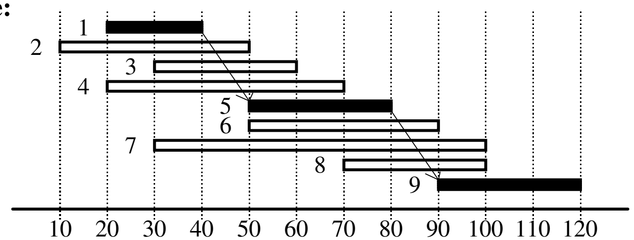
Corollary: Kruskal's Algorithm computes a MST.

Activity Selection

Input: A set $S = \{1, 2, \ldots, n\}$ of n activities, each with a start time s_i and a finish time $f_i \geq s_i$.

Output: A maximal-sized set of mutually-compatible activities. (Activities i and j are *compatible* if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.)

Example:



Solutions: $\{1, 5, 9\}$ or $\{1, 6, 9\}$ or $\{2, 5, 9\}$ or $\{2, 6, 9\}$.

A Greedy Algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  sort activities so that f_1 ≤ f_2 ≤ ··· ≤ f_n
2  A ← {1};  j ← 1
3  for i ← 2 to n do
4      if s_i ≥ f_j then
5          A ← A ∪ {i};  j ← i
6  return A
```

At each stage, the algorithm greedily chooses for inclusion in A the earliest-finishing activity compatible with the activities already chosen.

The running time of the algorithm is dominated by the sorting of activities in line 1, giving it a running time of $\Theta(n \lg n)$ (assuming an optimal sorting algorithm is used).

If the activities are already sorted, the algorithm (from line 2 onward) runs in time $\Theta(n)$.

Correctness of the Algorithm

Fact: At any point, A is a subset of a solution.

Proof: By induction on $|A|$. This is clearly true (initially) when $A = \emptyset$.

Suppose $A \neq \emptyset$; let k be the most recently added activity, and B be a solution which (by induction) contains $A-\{k\}$ but not k.

By induction, the activities of $A-\{k\}$ are mutually-compatible; and k, by being added, is compatible with the activities of $A-\{k\}$. Thus the activities of A must be mutually compatible.

Choose the $i \in B-A$ with the least finish time.

We must have $f_k \leq f_i$ (for otherwise i would have been added to A rather than k).

But then we can get another solution by replacing i by k in B. □

Corollary: The algorithm is correct.

When Greedy Algorithms Work

Not every optimization problem can be solved using a greedy algorithm. (For example, Making Change with a poor choice of coins.)

There are two vital components to a problem which make a greedy algorithm appropriate:

Greedy-choice property: *A globally optimal solution to the problem can be obtained by making a locally-optimal (greedy) choice.*

(A greedy algorithm does not look ahead nor backtrack; hence a single bad choice, no matter how attractive it was when made, will lead to a suboptimal solution.)

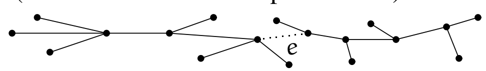
Optimal substructure property: *An optimal solution to the problem contains optimal solutions to subproblems.*

(A greedy algorithm works by iteratively finding optimal solutions to these subproblems, having made its initial greedy choice.)

MST and Activity Selection Revisited

Greedy-choice property for MST: If T is a MST, then T contains the edge e with the least weight. (Otherwise we could replace some edge in T with e and arrive at a better solution.)

Optimal substructure property for MST: If T is a MST, then removing the edge e with the least weight leaves two MSTs of smaller graphs. (Otherwise we could improve on T.)



Greedy-choice property for Activity Selection: If A is an optimal solution, then we can assume that it contains 1. (Otherwise we can replace the first activity in A by 1.)

Optimal substructure property for Activity Selection: If A is an optimal solution, then $A-\{1\}$ is an optimal solution to $\{i : s_i \geq f_1\}$. (Otherwise we could improve on A.)

Data Compression

We wish to compress a text file (a string of characters) using a binary code of variable length, that is, a code in which the number of bits needed for the encoding may vary from character to character.

We restrict our attention to *prefix* codes, that is, codes in which no codeword is a prefix of any other codeword.

For example, the code

$$\texttt{a} \mapsto \texttt{0} \qquad \texttt{b} \mapsto \texttt{10} \qquad \texttt{c} \mapsto \texttt{11}$$

is a prefix code, whereas the code

$$\texttt{a} \mapsto \texttt{0} \qquad \texttt{b} \mapsto \texttt{11} \qquad \texttt{c} \mapsto \texttt{111}$$

is not.

(In the latter case, there is no way of telling whether `111111` encodes `bbb` or `cc`.)

Why Variable-Length Prefix Codes

It would be wasteful to use a fixed-length code. For example, a fixed-length code for the three characters `a`, `b` and `c` as above would require (at least) two bits for every character; by encoding `a` with only one bit, we save one bit for every occurrence of `a`.

(Presumably `a` occurs more often than either `b` or `c`; the more frequently-occurring characters should have shorter codes.)

Prefix codes allow easy encoding and decoding.

- To encode a text we simply replace each character with its code and concatenate them.
- To decode the text, we identify the initial codeword, translate it back to the original character, remove it from the encoded file, and repeat the decoding procedure.

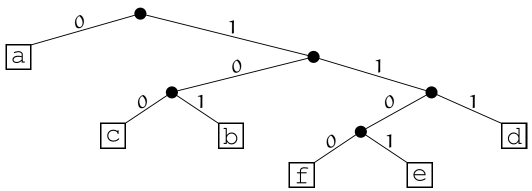
Eg, using the above prefix code, the string `0110101011` uniquely corresponds to `acabbc`.

The Tree Representation of a Code

Any prefix code is represented in an obvious way by a binary tree. For example, the code

$$\texttt{a} \mapsto 0 \quad \texttt{b} \mapsto 101 \quad \texttt{c} \mapsto 100 \quad \texttt{d} \mapsto 111 \quad \texttt{e} \mapsto 1101 \quad \texttt{f} \mapsto 1100$$

is represented by the following binary tree.



Using the tree, it is easy to decode any encoded text. For example:

`10101111111011101111`

is easily decoded, character-by-character, to

101	0	111	111	1101	1101	111
b	a	d	d	e	e	d

A More Efficient Code

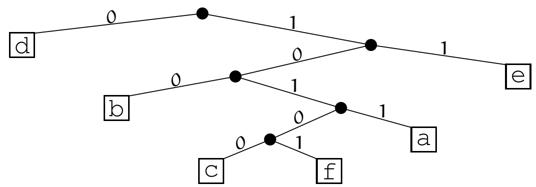
Using the previous code, the text `baddeed` is encoded by a bit string of length 21.

If instead we use the code

$$\texttt{a} \mapsto 1011 \quad \texttt{b} \mapsto 100 \quad \texttt{c} \mapsto 10100 \quad \texttt{d} \mapsto 0 \quad \texttt{e} \mapsto 11 \quad \texttt{f} \mapsto 10101$$

then the text `baddeed` would instead be encoded as the 14-bit string `10010110011110`, rather than as a 21-bit string.

The tree representation of this code is as follows.



Can we improve on this? No!

Computing the encoded length

Given a prefix code tree T, we can compute the number of bits required to encode a given text.

For alphabet C, let $f(c)$ be the frequency (number of occurrences) of the character $c \in C$ in the text, and let $d_T(c)$ be the depth of the leaf labelled c in T (that is, the length of the code for c).

Then the number of bits required to encode the text is

$$B(T) \;=\; \sum_{c \in C} f(c) d_T(c).$$

For example, the number of bits required to encode `baddeed` is:

$$\begin{aligned} B(T) &= f(a)d_T(a) + \cdots + f(f)d_T(f) \\ &= 1\cdot 4 + 1\cdot 3 + \cdots + 2\cdot 2 + 0\cdot 5 \\ &= 4 + 3 + 0 + 3 + 4 + 0 \;=\; 14. \end{aligned}$$

The Data Compression Problem

Input: A set C of characters (which possibly appear in some text to be compressed), along with a function $f : C \to \mathbb{N}$ indicating the number of times each character appears.

Output: A binary code which provides an optimal compression of the text.

A Greedy Algorithm: Construct the prefix code tree bottom-up, starting with all characters as leaves, and successively merging the two lowest-frequency sub-trees.

(Thus high-frequency characters are merged after the lower-frequency characters, giving them the tendency to end up higher up in the tree.)

This is the idea underlying **Huffman Codes**.

Constructing a Huffman Code

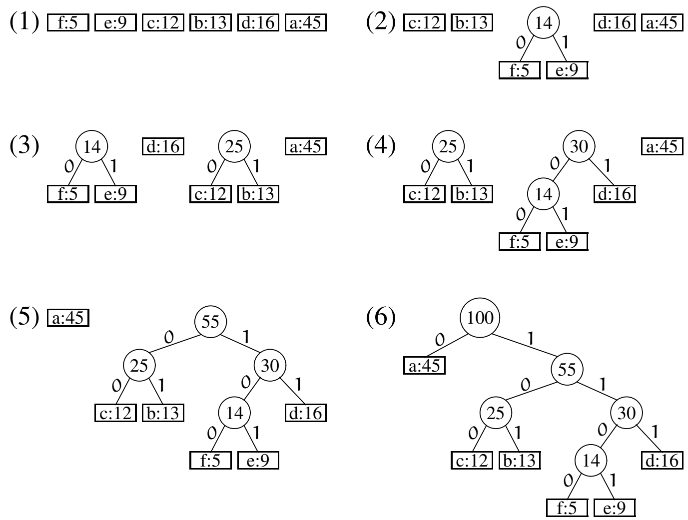
```
HUFFMAN(C)
1  n ← |C| ;  Q ← C
2  for i ← 1 to n − 1 do
3        z ← ALLOCATE-NODE()
4        x ← left(z) ← EXTRACT-MIN(Q)
5        y ← right(z) ← EXTRACT-MIN(Q)
6        f(z) ← f(x) + f(y)
7        INSERT(Q, z)
8  return EXTRACT-MIN(Q)
```

Q is a priority queue, keyed on f.

Assuming the queue is implemented as a binary heap, its initialization (line 2) can be performed in $O(n)$ time; and each heap operation takes time $O(\lg n)$ time.

Thus the total running time is $O(n \lg n)$.

Huffman Coding Illustrated



The Greedy-Choice Property for Huffman's Algorithm

Let:
- T represent an optimal prefix code;
- $x, y \in C$ be the two least-frequent characters;
- $a, b \in C$ be siblings with the longest codes;
- T' be T with $x \leftrightarrow a$ and $y \leftrightarrow b$ exchanged.

Fact: $B(T) - B(T') \geq 0$, so T' also represents an optimal prefix code.

Proof:

$$\begin{aligned}
B(T) - B(T') &= \textstyle\sum_{c \in C} f(c)d_T(c) \;-\; \sum_{c \in C} f(c)d_{T'}(c) \\
&= f(x)d_T(x) + f(y)d_T(y) + f(a)d_T(a) + f(b)d_T(b) \\
&\quad - f(x)d_{T'}(x) - f(y)d_{T'}(y) - f(a)d_{T'}(a) - f(b)d_{T'}(b) \\
&= f(x)d_T(x) + f(y)d_T(y) + f(a)d_T(a) + f(b)d_T(b) \\
&\quad - f(x)d_T(a) - f(y)d_T(b) - f(a)d_T(x) - f(b)d_T(y) \\
&= (f(a) - f(x))(d_T(a) - d_T(x)) \\
&\quad + (f(b) - f(y))(d_T(b) - d_T(y)) \quad \geq \quad 0.
\end{aligned}$$

$\square$

Corollary: The greedy-choice property holds.

The Optimal Substructure Property for Huffman's Algorithm

Let:
- T represent an optimal prefix code;
- $x, y \in C$ be siblings with parent z;
- $T' = T - \{x, y\}$; $C' = C - \{x, y\} \cup \{z\}$; and $f(z) = f(x) + f(y)$.

Fact: T' represents an optimal prefix code for C'.

Proof: $B(T) = B(T') + f(x) + f(y)$.

Hence if there were a more optimal tree, then replacing x and y under z in this tree would provide a more optimal solution to the original problem. $\square$

Corollary: The optimal substructure property holds.

Corollary: The algorithm is correct.

When Greedy Algorithms Fail: Making Change Revisited

Suppose we want to solve the Making Change problem of paying 9 cents with 1, 4 and 6 cent coins.

The greedy algorithm would give 6+1+1+1 rather than the optimal 4+4+1.

To solve the general Making Change problem, we can do as follows:

- In order to pay the sum of N cents using n distinct coins $\langle d_1, d_2, \ldots, d_n \rangle$, we set up an $n \times (N+1)$ table $c[1..n, 0..N]$.
- In this table, $c[i, j]$ will hold the minimum number of coins required to pay the amount j using only coins $d_1, \ldots, d_i$.
  (If no arrangement of such coins makes up N cents, then we shall have $c[i, j] = \infty$.)
- The solution will then be contained in $c[n, N]$.

It then suffices to find a way to fill in this table.

Bookkeeping for Making Change

To fill out the table, we can proceed as follows:

- Clearly $c[i, 0] \leftarrow 0$ for every i.
- Also, for every j, $c[1, j] \leftarrow \begin{cases} j \text{ div } d_1 & \text{if } j \bmod d_1 = 0, \\ \infty & \text{otherwise.} \end{cases}$
  (Whenever we cannot make change for amount j using coins $d_1, \ldots, d_i$, we let $c[i, j] = \infty$.)
- For $c[i, j]$ ($i > 1, j > 0$), we may either:
  - pay j cents using only coins $d_1, \ldots, d_{i-1}$: $c[i, j] \leq c[i-1, j]$; or
  - use (at least) one coin d_i, and reduce the problem to that of paying $j - d_i$: $c[i, j] \leq 1 + c[i, j - d_i]$.
- As we want to minimize the number of coins, we choose the better of these two options:

$$c[i, j] \leftarrow \min\big(c[i-1, j], 1 + c[i, j - d_i]\big).$$

The Making-Change Algorithm

```
MAKING-CHANGE(N, d[1..n])                ▷ Running time O(nN)
for i ← 1 to n do c[i, 0] ← 0
for j ← 1 to N do
if (j mod d_1)=0 then c[1, j] ← j div d_1 else c[1, j] ← ∞
for i ← 2 to n do
    for j ← 1 to N do
        if j < d_i then c[i, j] ← c[i−1, j]
        else c[i, j] ← min(c[i−1, j], 1 + c[i, j−d_i])
return c
```

Example: Paying 9 cents using 6, 1 and 4 cent coins (order irrelevant).

Amount	0	1	2	3	4	5	6	7	8	9
$d_1 = 6$	0	∞	∞	∞	∞	∞	1	∞	∞	∞
$d_2 = 1$	$\boxed{0}$	$\boxed{1}$	2	3	4	5	1	2	3	4
$d_3 = 4$	0	$\boxed{1}$	2	3	1	$\boxed{2}$	1	2	2	$\boxed{3}$

Determining the optimal solution

The algorithm MAKING-CHANGE tells us that only $c[3, 9] = 3$ coins are necessary to make up 9 cents; but it doesn't say *which* three coins to use.

The following algorithm will answer that question, by retracing the solution through the c table.

```
PAY-OUT(c[1..n, 0..N], d[1..n])
i ← n ;  j ← N ;  coins ← ⟨⟩
while j > 0 do
    if c[i, j] = c[i−1, j] then i ← i−1
    else coins ← d_i :: coins ;  j ← j−d_i
return coins
```

This algorithm involves stepping back n rows, and making $c[n, N]$ jumps to the left. Hence it runs in time $O(n + N)$, and is thus a negligible addition to the $O(nN)$ algorithm MAKING-CHANGE.

Why the Greedy Algorithm fails

As with problems which can be solved by greedy algorithms, MAKING-CHANGE has the

Optimal Substructure Property: An optimal solution to the problem contains optimal solutions to subproblems.

However, the *greedy-choice property* fails. We now have to consider many potential solutions, which requires added bookkeeping: we need to remember past decisions, and also build solutions from the bottom up.

We do have something though, namely the

Overlapping Subproblems Property: The space of subproblems is small, so an otherwise-obvious top-down, divide-and-conquer recursive algorithm would solve the same subproblems over and over.

Dynamic Programming

The presence of

- **Optimal Substructure Property** and
- **Overlapping Subproblems Property**

characterises **Dynamic Programming**.

With dynamic programming, we take a natural recursive definition, and instead of computing it in a *top-down* fashion, we compute it *bottom-up*, thus exploiting the **overlapping subproblems property** by only solving each subproblem once.

For example, a top-down algorithm for computing Fibonacci numbers from their definition:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

would run in exponential time, while a bottom-up algorithm, computing $F_0, F_1, F_2, F_3, F_4, \ldots, F_{n-1}, F_n$ would run in linear time.

Matrix Chain Multiplication

Assumption: Computing $A \times B$ where A is a $p \times q$ matrix and B is a $q \times r$ matrix requires pqr scalar multiplications.

Question: What is the most efficient order for multiplying $A_1 \times A_2 \times \cdots \times A_n$ where A_i is a $p_{i-1} \times p_i$ matrix?

Example: Let A_1, A_2, A_3 and A_4 be matrices of dimensions 6×2, 2×5, 5×4 and 4×3. Then $A_1 A_2 A_3 A_4$ can be computed in five ways:

- $((A_1 A_2) A_3) A_4 \implies 252$ scalar multiplications.
- $(A_1 A_2)(A_3 A_4) \implies 210$ scalar multiplications.
- $(A_1 (A_2 A_3)) A_4 \implies 160$ scalar multiplications.
- $A_1 ((A_2 A_3) A_4) \implies 100$ scalar multiplications.
- $A_1 (A_2 (A_3 A_4)) \implies 126$ scalar multiplications.

So the right choice can make a big difference.

But we generally cannot make this choice by exhaustive search as in the above example, as there are exponentially-many possibilities.

A Dynamic Programming Solution

Let $m[i,j]$ be the cost (number of scalar multiplications) of computing $A_i \times A_{i+1} \times \cdots \times A_j$ using an optimal order of multiplying the matrices. Then

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

If we implement this as a top-down recursive algorithm, it would run in exponential time. However, there are only a quadratic number of subproblems. (The recursive algorithm solves these subproblems over and over.) Hence, we use a bottom-up dynamic-programming algorithm.

Note though: $m[1,n]$ only tells us how many scalar multiplications are necessary in the optimal solution, not what the optimal solution is (ie, the order in which to multiply the matrices). For this, we can maintain a second table s, and record in $s[i,j]$ the value of k from which the minimal value of $m[i,j]$ was computed.

The Algorithm

```
MATRIX-CHAIN-ORDER(p)                ▷ Running time O(n^3)
  n ← length(p)−1
  for i ← 1 to n do m[i,i] ← 0
  for l ← 2 to n do
     for i ← 1 to n−l+1 do
        j ← i+l−1 ;  m[i,j] ← ∞
        for k ← i to j−1 do
            q ← m[i,k] + m[k+1,j] + p_{i−1} p_k p_j
            if q < m[i,j] then m[i,j] ← q ;  s[i,j] ← k
  return (m, s)
```

For our example, $p = \langle 6,2,5,4,3 \rangle$, and the tables computed are:

m	1	2	3	4
1	0	60	88	100
2	–	0	40	64
3	–	–	0	60
4	–	–	–	0

s	1	2	3	4
1	0	1	1	1
2	–	0	2	3
3	–	–	0	3
4	–	–	–	0

Printing the Matrix-Chain

Using the s table, we can then reconstruct and print out an optimal Matrix Chain as follows.

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

We would then invoke this algorithm initially by

MATRIX-CHAIN-PRINT$(s, 1, n)$.

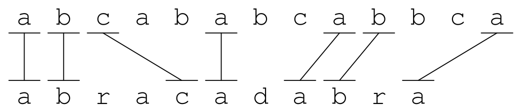
For our example, we would get the output: $(A_1((A_2 A_3) A_4))$.

We could equally modify this algorithm to carry out the matrix multiplications in an optimal way.

Longest Common Subsequence (LCS)

Problem: Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find the longest common subsequence.

Example: The longest common subsequence of the sequences



is `abcaaba` and is of length 7.

There are 2^m subsequences of X, so we cannot check them all against Y.

However, we can express this as a dynamic programming problem, and derive an efficient algorithm to solve it.

Dynamic Programming LCS

Let $c[i,j]$ denote the length of the LCS of $x_1 \cdots x_i$ and $y_1 \cdots y_j$.

Then $c[m,n]$ is the length of the LCS we are trying to compute.

A recursive definition for c, exploiting an obvious optimal subproblem property, is given as follows.

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

Computing the length of the LCS

The following is a dynamic-programming algorithm which computes $c[i,j]$, as well as a further table $b[i,j]$ to use for constructing the LCS.

```
LCS-LENGTH(X, Y)    ▷ Running time O(mn)
m ← length(X) ;  n ← length(Y)
for i ← 0 to m do c[i, 0] ← 0
for j ← 0 to n do c[0, j] ← 0
for i ← 1 to m do
    for j ← 1 to n do
        if x_i = y_j then
            c[i, j] ← c[i−1, j−1] + 1 ;  b[i, j] ← "↖"
        else if c[i−1, j] ≥ c[i, j−1] then
            c[i, j] ← c[i−1, j] ;  b[i, j] ← "↑"
        else c[i, j] ← c[i, j−1] ;  b[i, j] ← "←"
return(c, b)
```


Printing the LCS

```
PRINT-LCS(b, X, i, j)    ▷ Running time O(m + n)
1  if i=0 or j=0 then return
2  else if b[i, j] = "↑" then PRINT-LCS(b, X, i−1, j)
3  else if b[i, j] = "←" then PRINT-LCS(b, X, i, j−1)
4  else PRINT-LCS(b, X, i−1, j−1) ;  print x_i
```

Example:

LCS = BCBA

	j	0	1	2	3	4	5	6	7	8
i	y_j		**B**	D	**C**	B	B	A	**B**	**A**
0	x_i	0	0	0	0	0	0	0	0	0
1	A	**0**	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	**B**	0	↖ **1**	← **1**	← 1	↖ 1	↖ 1	↑ 1	↖ 2	← 2
3	**C**	0	↑ 1	↑ 1	↖ **2**	← **2**	← **2**	← **2**	↑ 2	↑ 2
4	**B**	0	↖ 1	↑ 1	↑ 2	↖ 3	↖ 3	← 3	↖ **3**	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ **3**	↑ 3
6	**A**	0	↑ 1	↑ 2	↑ 2	↑ 3	↑ 3	↖ 4	← 4	↖ **4**

All-Pairs Shortest Paths

Problem: Calculating the shortest route between any two cities from a given set of cities $1, 2, \ldots, n$.

Input: A matrix $d_{i,j}$ ($1 \leq i, j \leq n$) of nonnegative values indicating the length of the direct route from i to j.

Note: $d_{i,i} = 0$ for all i; and if there is no direct route from i to j, then $d_{i,j} = \infty$.

Output: A shortest distance matrix $s_{i,j}$ indicating the length of the shortest route from i to j.

We shall give a recursive definition for s, which can be computed by a dynamic-programming algorithm.

The Floyd-Warshall Algorithm

Let $s^{(k)}_{i,j}$ denote the shortest distance from i to j which only passes through cities $1, 2, \ldots, k$. A recursive definition for $s^{(k)}_{i,j}$ is given as follows.

$$s^{(k)}_{i,j} = \begin{cases} d_{i,j} & \text{if } k = 0, \\ \min\left(s^{(k-1)}_{i,j}, s^{(k-1)}_{i,k} + s^{(k-1)}_{k,j}\right) & \text{if } k > 0. \end{cases}$$

```
FLOYD-WARSHALL-1(d, n)
s^(0) ← d
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
                s^(k)_{i,j} ← min( s^(k−1)_{i,j},  s^(k−1)_{i,k} + s^(k−1)_{k,j} )
```

This algorithm runs in $O(n^3)$ time and space.
However, we can safely remove the superscripts from s (can you see why?), and achieve $O(n^2)$ space.

Constructing Shortest Paths

To construct the shortest paths, we maintain a *predecessor matrix* $\pi_{i,j}$ in which $\pi_{i,j}$ denotes the predecessor of j on some shortest path from i to j. (If $i = j$ or there is no such path, then $\pi_{i,j} = \text{NIL}$.)

The final algorithm for computing s and π is thus as follows.

```
FLOYD-WARSHALL(d, n)
s ← d
for i ← 1 to n do
    for j ← 1 to n do
        if i=j or d_{i,j}=∞ then π_{i,j} = NIL else π_{i,j} = i
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
                x ← s_{i,k} + s_{k,j}
                if x < s_{i,j} then s_{i,j} ← x ;  π_{i,j} ← π_{k,j}
```


Example

$s^{(0)}/\pi^{(0)} = s^{(1)}/\pi^{(1)}$

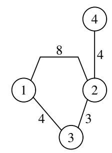
	1	2	3	4
1	0/NIL	8/1	4/1	∞/NIL
2	8/2	0/NIL	3/2	4/2
3	4/3	3/3	0/NIL	∞/NIL
4	∞/NIL	4/4	∞/NIL	0/NIL

$s^{(2)}/\pi^{(2)}$

	1	2	3	4
1	0/NIL	8/1	4/1	12/2
2	8/2	0/NIL	3/2	4/2
3	4/3	3/3	0/NIL	7/2
4	12/2	4/4	7/2	0/NIL

$s^{(3)}/\pi^{(3)} = s^{(4)}/\pi^{(4)}$

	1	2	3	4
1	0/NIL	7/3	4/1	11/2
2	7/3	0/NIL	3/2	4/2
3	4/3	3/3	0/NIL	7/2
4	11/3	4/4	7/2	0/NIL



d	1	2	3	4
1	0	8	4	∞
2	8	0	3	4
3	4	3	0	∞
4	∞	4	∞	0