Algorithm Design and Analysis

Sergey Bereg

2016

Contents

Chapter 1

Introduction

1.1 Computational problems and algorithms

In a computational problem, we are given an input, an output and an input/output relationship. Example: the sorting problem can be defined as follows.

Input: A sequence of n numbers $s_1, s_2, \ldots, s_n$.

Output: A permutation of the input numbers in non-decreasing order.

For example, given the input sequence 7,3,5,1,8,2, the corresponding (correct) output is 1,2,3,5,7,8. Such an input sequence is called an instance of the sorting problem. In general, an instance of a computational problem gives specific input values to the problem.

Another example of a computational problem is the integer multiplication.

Input: Two n-digit numbers x and y.

Output: Product $x \cdot y$.

An algorithm is a step-by-step procedure for solving a computational problem. It takes some values as input and returns some values as output. An algorithm is correct if it halts with the correct output.

1.2 Insertion sort

Insertion sort is a simple algorithm that inserts one number at a time into a sorted sequence. *InsertionSort* sorts n numbers *in place*: the numbers are rearranged within an array $S[0..n-1]$. At the beginning the sorted sequence contains only one number $S[0]$, so $S[1]$ is the first number to insert.

```
$InsertionSort(S)$
// Input: $S[0..n-1]$ is the array of $n$ numbers.
// Output: sorted array $S$.
1   for $i = 1$ to $n-1$
2      $x = S[i]$
       // Insert $t$ into the sorted sequence $S[0..i-1]$
3      for ( $j = i-1$; $j \geq 0$ && $S[j] > s$; $j--$ )
4         $S[j+1] = S[j]$
5      $S[j+1] = x$
```

Example. Sort 8,6,5,14,15,4,10,12 using *InsertionSort*.

input	8	6	5	14	15	4	10	12
$i = 1$	6	8	5	14	15	4	10	12
$i = 2$	5	6	8	14	15	4	10	12
$i = 3$	5	6	8	14	15	4	10	12
$i = 4$	5	6	8	14	15	4	10	12
$i = 5$	4	5	6	8	14	15	10	12
$i = 6$	4	5	6	8	10	14	15	12
$i = 7$	4	5	6	8	10	12	14	15

Consider one iteration of the "for" loop. Suppose that, at the beginning (line 2), the subarray $S[0..i-1]$ is sorted. The "while" loop (line 4) shifts the elements larger than x (there could be no shifts if x is the largest element). Then x is inserted in the right place and the subarray $S[0..i]$ is sorted. By repeating this argument $n-1$ times, the entire array $S[..]$ is sorted. Therefore *InsertionSort* is correct.

This proof can be viewed as the proof of the loop invariant: at line 2, the subarray $S[0..i-1]$ is sorted. It is similar to a proof by induction. Difference: the loop terminates but the induction is infinite (usually).

Analysis of insertion sort

Let t be the number of shifts, .i.e. the number of times line 5 is executed. We count the number of times for each line:

	Line	Total number of times
1	for $i = 1$ to $n-1$	$\leq n$
2	$x = S[i]$	$n-1$
3	for loop $\leq t+n$	$t+n$
4	$S[j+1] = S[j]$	t
5	$S[j+1] = x$	$n-1$

Since every line is executed in $O(1)$ time, the total time is $T(n) = O(n+t)$. The running time depends not only on n but also on t.

Running time cases

For a given n, there are many instances of the input and the running time varies.

- *Best case*. An input instance with the least running time.
- *Worst case*. An input instance with the maximum running time.
- *Average case*. It captures "average" behavior and is difficult to define formally. Instead, we will use expected running time for randomized algorithms (which use random choices).

Running time of *InsertionSort*. The best case occurs if the input sequence is already sorted. Then $t = 0$ and the best-case running time is $O(n)$. The worst-case input for the insertion sort is a decreasing sequence. In this case i elements are shifted when $S[i]$ is inserted.Then $t = 1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2} \leq n^2$. The worst-case running time is $O(n^2)$.

1.3 O-notation

Definition. We say that a function $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$. We write $f(n) = O(g(n))$.

Example: Show by the definition that $4n + 7$ is $O(n)$.

Solution: $f(n) = 4n + 7$ and $g(n) = n$. We want to find c and n_0 such that $4n + 7 \leq cn$ for all $n \geq n_0$. We can choose $c = 5$ and $n_0 = 7$.

Scratch paper.

$$\begin{aligned} 4n + 7 &\leq cn \\ cn - 4n &\geq 7 \qquad \text{(assume that } c > 4) \\ n &\geq \frac{7}{c-4} \qquad \text{(suppose that } c = 5) \\ n &\geq 7. \end{aligned}$$

Example: Show by the definition that $2n^3 + 3n^2 + 7$ is $O(n^3)$.

Solution: $f(n) = 2n^3 + 3n^2 + 7$ and $g(n) = n^3$. We want to find c and n_0 such that $2n^3 + 3n^2 + 7 \leq cn$ for all $n \geq n_0$. We can choose $c = 12$ and $n_0 = 1$.

Scratch paper.

$$\begin{aligned} 3n^2 &\leq 3n^3 \quad \text{if } n \geq 1 \\ 7 &\leq 7n^3 \quad \text{if } n \geq 1 \\ 2n^3 + 3n^2 + 7 &\leq (2+3+7)n^3 = 12n^3 \quad \text{if } n \geq 1 \end{aligned}$$

We also define relationships Ω, Θ, o and ω. We assume that $f(n)$ and $g(n)$ are nonnegative functions.

Definition. We say that $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that

$$f(n) \geq c \cdot g(n)$$

for all $n \geq n_0$.

Meaning: $f()$ is asymptotically "greater than or equal to" $g()$ up to a constant factor. In other words, $f()$ "grows no slower than" $g()$.

Example: Show $5n - 3$ is $\Omega(n)$.

Solution: We use the definition for $f(n) = 5n - 3$ and $g(n) = n$. We want to find c and n_0 such that $5n - 3 \geq cn$ for all $n \geq n_0$. It is true, if we choose $c = 1$ and $n_0 = 1$.

Scratch paper.

$$\begin{aligned} 5n - 3 &\geq cn \\ (5 - c)n &\geq 3 \qquad \text{(assume that } 5 - c > 0) \\ n &\geq \frac{3}{5 - c} \qquad \text{(suppose that } c = 1) \\ n &\geq \frac{3}{4}. \qquad \text{(it follows from } n \geq 1) \end{aligned}$$

Example: Show $2n^3 - 5n^2 - 3 = \Omega(n^3)$.

Solution: We use the definition for $f(n) = 2n^3 - 5n^2 - 3$ and $g(n) = n^3$. We want to find c and n_0 such that $2n^3 - 5n^2 - 3 \geq cn^3$ for all $n \geq n_0$. We can pick any $c \in (0, 2)$, say $c = 1$. Then we pick $n_0 = 10$.

Scratch paper. $2n^3 - 5n^2 - 3 \geq n^3$ or $n^3 \geq 5n^2 + 3$ follows from $n^3/2 \geq 5n^2$ and $n^3/2 \geq 3$.
$n^3/2 \geq 5n^2$ simplifies to $n \geq 10$
$n^3/2 \geq 3$ or $n^3 \geq 6$ follows from $n \geq 2$.

Definition. We say that $f(n)$ is $\Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$; that is, there exist positive constants c_1, c_2 and n_0 such that, for all $n \geq n_0$,

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

Meaning: $f()$ is asymptotically "equal to" $g()$ up to a constant factor. In other words, $f(n)$ and $g(n)$ have the same growth rate.

Example: Show that $5n - 3$ is $\Theta(n)$.

Solution: First, $5n - 3 = O(n)$ since $5n - 3 \leq cn$ for $c = 5$ and all $n \geq 0$. $5n - 3 = \Theta(n)$ since $5n - 3 = \Omega(n)$ from the example above.

Definition. We say that $f(n)$ is $o(g(n))$ if, for any constant $c > 0$, there is a constant n_0 such that, for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n).$$

meaning: $f()$ is asymptotically "less than" $g()$ up to a constant factor. In other words, $f(n)$ "grows slower than" $g(n)$. o is stronger than O: If $f(n) = o(g(n))$ then $f(n) = O(g(n))$.

Example: Show that $5n - 3$ is $o(n^2)$.

Solution: Let $c > 0$ be any constant. We choose $n_0 = 5/c$. Note that $n \geq n_0$ means $cn \geq 5$ or $5 \leq cn$. Then $f(n) = 5n - 3 \leq 5n \leq cn^2 = c \cdot g(n)$ for all $n \geq n_0$.

Definition. We say that $f(n)$ is $\omega(g(n))$ if, for any constant $c > 0$, there is a constant n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

It means that $f(n)$ is asymptotically "greater than" $g(n)$ up to a constant factor. In other words, $f(n)$ "grows faster than" $g(n)$.

Example: Show that $5n - 3$ is $\omega(\sqrt{n})$.

Solution: Let $c > 0$ be any constant. We choose $n_0 = \max(1, c^2)$. Note that $n \geq 1$ implies $5n - 3 \geq n$ and $n \geq c^2$ implies $\sqrt{n} \geq c$. Then $f(n) = 5n - 3 \geq n \geq c\sqrt{n} = c \cdot g(n)$ for all $n \geq n_0$.

1.4 Growth Rates of Functions

Limit Theorem. If $f(n) \geq 0, g(n) > 0$ for all n and

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 \\ c > 0 \\ \infty \end{cases} \text{, then } f(n) = \begin{cases} o(g(n)) \\ \Theta(g(n)) \\ \omega(g(n)) \end{cases} .$$

Example: Show that $n/\lg n = o(n)$ (in these notes $\lg$ is actually $\log_2$).

Solution: It follows by the Limit Theorem since $\lim_{n\to\infty} \frac{n/\lg n}{n} = \lim_{n\to\infty} \frac{1}{\lg n} = 0$.

L'Hopital's Rule. If $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n) = \infty$, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$.

Recall the derivatives $(a^x)' = a^x \cdot \ln a$ and $(\log_a x)' = \frac{1}{x \ln a}$.

Example: Prove that $n^3 = o(2^n)$.

Solution: We can apply the L'Hopital's Rule three times.

$$\lim_{n\to\infty} \frac{n^3}{2^n} = \lim_{n\to\infty} \frac{3n^2}{2^n \cdot \ln 2} = \lim_{n\to\infty} \frac{6n}{2^n \cdot (\ln 2)^2} = \lim_{n\to\infty} \frac{6}{2^n \cdot (\ln 2)^3} = 0.$$

Another way. To prove that $\frac{f(n)}{g(n)} \to 0$ it suffices to show that $\left(\frac{f(n)}{g(n)}\right)^{1/3} \to 0$. Then we apply the L'Hopital's Rule just one time $\lim_{n\to\infty} \left(\frac{n^3}{2^n}\right)^{1/3} = \lim_{n\to\infty} \frac{n}{2^{n/3}} = \lim_{n\to\infty} \frac{1}{2^{n/3} \cdot \ln 2/3} = 0$.

Example: Show that $\lg n = o(n)$.

Solution: By the L'Hopital's Rule $\lim_{n\to\infty} \frac{\lg n}{n} = \lim_{n\to\infty} \frac{1/(n \ln 2)}{1} = 0$.

Theorem. Let $f(n), g(n), h(n), t(n)$ be nonnegative functions.

1. If $f = O(g)$ then $cf = O(g)$ for any constant $c > 0$.
2. If $f = O(g)$ and $h = O(t)$ then $f + h = O(g + t)$ and $f \cdot h = O(g \cdot t)$.
3. If f is a polynomial of degree d then $f = \Theta(n^d)$.
4. $n^a = o(b^n)$ for any constants $a > 0$ and $b > 1$.
5. $\log^a n = o(n^b)$ for any constants $a > 0$ and $b > 0$.

Useful facts

Transitivity:
$f(n) = O((g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O((h(n))$.
$f(n) = \Omega((g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega((h(n))$.
$f(n) = \Theta((g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta((h(n))$.
$f(n) = o((g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o((h(n))$.
$f(n) = \omega((g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega((h(n))$.

Reflexivity:
$f(n) = O((f(n))$, $f(n) = \Omega((f(n))$, $f(n) = \Theta((f(n))$.

Symmetry:
$f(n) = \Theta((g(n))$ if and only if $g(n) = \Theta((f(n))$.

o is stronger than O: If $f(n) = o(g(n))$ then $f(n) = O(g(n))$.

Table of Growth Rates

Function	Name
c	constant
$\lg n$	logarithmic
$\lg^2 n$	log-squared
n	linear
$n \lg n$	
n^2	quadratic
n^3	cubic
2^n	exponential

A list of common functions encountered when analyzing the running time of an algorithm. The slower-growing functions are listed first.

Functions

Polynomials. A polynomial of degree d is a function $p(n) = \sum_{i=0}^{d} a_i n^i$ where $a_0, a_1, \ldots, a_d$ are the coefficients of the polynomial and $a_d \neq 0$.

Exponentials. $a^n, a > 0$. Properties: $(a^m)^n = a^{mn} = (a^n)^m, a^m a^n = a^{m+n}$.

Logarithms. For all real $a > 0, b > 0, c > 0$, and n,

$$\lg n = \log_2 n \qquad \ln n = \log_e n$$
$$\lg^k n = (\lg n)^k \qquad \lg\lg n = \lg(\lg n)$$
$$a = b^{\log_b a} \qquad \log_c(ab) = \log_c a + \log_c b$$
$$\log_b a^n = n \log_b a \qquad \log_b a = \frac{\log_c a}{\log_c b}$$
$$\log_b a = \frac{1}{\log_a b} \qquad a^{\log_b c} = c^{\log_b a}$$

$$\lim_{n\to\infty} \frac{\lg n}{n^a} = 0, a > 0$$

Factorials. $n! = 1 \cdot 2 \cdot 3 \dots n$
Stirling's approximation $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta(\frac{1}{n})\right)$
It implies that $\lg(n!) = \Theta(n \lg n)$.

Series

Arithmetic series:
$$\sum_{i=1}^{n} i = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$
Geometric:
$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \dots + a^n = \frac{1 - a^{n+1}}{1 - a} \text{ where } 0 < a \neq 1.$$
$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ where } 0 < a < 1.$$
Harmonic sum
$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n = \log_e n$$
Telescoping series.
$$\sum_{i=1}^{n} (a_i - a_{i-1}) = a_n - a_0.$$
$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=0}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right) = 1 - \frac{1}{n+1}.$$

Example. For which values of k line 4 of ALG1() is executed and how many times for each k? Find the asymptotic running time of ALG1() using a summation.

```
ALG1($n$)
1  $s = 0, i = n, k = 0$
2  while $i \geq 1$
3      for $j = 1$ to $i$
4          $s+ = j * j$
5      $i = i/2, k = k + 1$
```

k runs lgn times

Solution. Line 5 is executed for $k = 0, 1, 2 \dots, \lg n$. It is executed $i = n/2^k$ times for each k.

The running time is

$$T(n) = \sum_{k=0}^{\lg n} \frac{n}{2^k} = n \sum_{k=0}^{\lg n} \frac{1}{2^k} = n \cdot \frac{1 - 1/2^{\lg n + 1}}{1 - 1/2} \le n \cdot \frac{1}{1/2} = 2n = O(n).$$

Fibonacci numbers

$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}, i \ge 2.$

Golden ratio is $a = (1 + \sqrt{5})/2$, its conjugate is $b = (1 - \sqrt{5})/2$.

Example. Prove by induction

$$F_i = \frac{a^i - b^i}{\sqrt{5}}.$$

Solution. Basis of induction: $i = 0, 1$.
Step of induction: Assume that

$$F_{i-1} = \frac{a^{i-1} - b^{i-1}}{\sqrt{5}} \quad \text{and} \quad F_{i-2} = \frac{a^{i-2} - b^{i-2}}{\sqrt{5}}.$$

We want to show that

$$\frac{a^i - b^i}{\sqrt{5}} = \frac{a^{i-1} - b^{i-1}}{\sqrt{5}} + \frac{a^{i-2} - b^{i-2}}{\sqrt{5}}, \quad \text{or}$$
$$a^{i-2}(a^2 - a - 1) = b^{i-2}(b^2 - b - 1).$$

It follows since a and b are the roots of $x^2 - x - 1 = 0$. 0=0 prove

Exercises

1.1.
Sort $12, 45, 66, 7, 41, 57$ using *InsertionSort* (show the array after each iteration of the main loop).

1.2.
Express the functions $n^2/38 + 0.8n^3 - 15$ and $n \lg n + n^2/130$ in terms of Θ-notation.

1.3.
Prove that $5^n = o(n!)$.

1.4.
How many times line 4 of ALG1 is executed for each i? In total? Give an asymptotic analysis of the running time using big-Oh (or big-Theta which would be technically more precise).

```
ALG1($n$)
1   $s = 0$
2   for $i = 1$ to $n$
3       for $j = 1$ to $i$
4           $s += j * j$
```

1.5.
Let $A[..]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

a. Find all inversions of the array $\langle 3, 5, 9, 7, 1\rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

c. Run *InsertionSort* on the array $\langle 3, 5, 9, 7, 1\rangle$ and compute the number of times line 5 is executed. Express the running time of *InsertionSort* in terms of n and I, the number of inversions in the array. Justify your answer.

1.6.
Prove that $(n + 98)^{99} = O(n^{99})$.

1.7.
Prove or disprove
a) $3^{n+3} = O(3^n)$,
b) $3^{2n} = O(3^n)$.

1.8.
Sort the following functions by asymptotic growth rate. Indicate functions with the same growth rate, i.e. $f(n) = \Theta(g(n))$.

$(\sqrt{3})^{\log_3 n}$	n^2	$n!$	$\ln n$	$(\frac{4}{3})^n$	n^3
$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n \lg n$	$\lg \lg n$	$n \cdot 2^n$
$4^{\lg n}$	$(n+1)!$	n	2^n	$2^{\lg n}$	e^n

1.9.
Find a function $f(n)$ such that $f(n) = o(n^2)$ and $f(n) = \Omega(n^2)$. Or prove that such a function does not exist.

1.10.
Suppose that $f(n)$ and $g(n)$ are positive functions.

a. Does $f(n) = O(g(n))$ imply $g(n) = O(f(n))$? Justify your answer.

b. Does $f(n) = O(g(n))$ imply $3^{f(n)} = O(3^{g(n)})$? Justify your answer.

Chapter 2

Divide-and-Conquer and Recurrences

Divide and conquer is an algorithm design technique which works

- by recursively breaking down a problem into two or more sub-problems of the same type and
- by combining their solutions to solve the original problem.

Important! Stop the recursion when the size of the problem is small and solve the problem directly (or using a different algorithm).

Usually the running time of such an algorithm can be found by solving a recurrence.

2.1 Mergesort

Mergesort is a divide and conquer algorithm. It has 3 steps (assuming that $n \geq 2$):

- Divide: divide the input array of n elements into two subproblems of size $n/2$.
- Conquer: sort the two subarrays recursively.
- Combine: merge the two sorted subarrays to produce the sorted array.

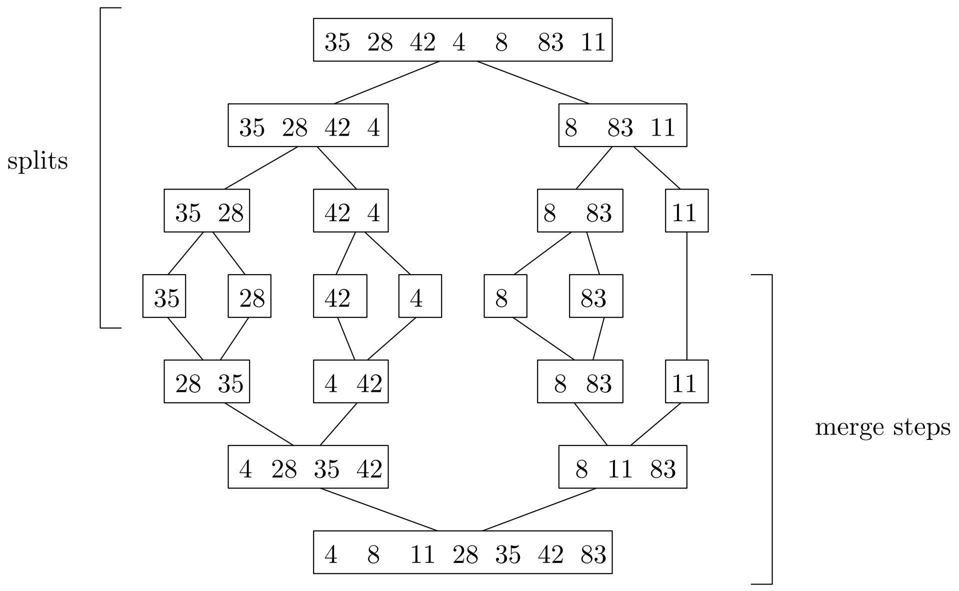
To sort $S[0..n-1]$ call $MergeSort(S, n)$.

```
$MergeSort\ (S, n)$
// Input: array $S[0..n-1]$ of $n$ numbers
// Output: sorted numbers in $S[l..r]$
1   create temp array $X[0..n-1]$
2   $RecurSort(0, n-1)$;
// assume that $S[..]$ and $X[..]$
// are available in $RecurSort$
```

```
$RecurSort\ (l, r)$
// Input: array $S[l..r]$
// Output: sorted numbers in $S[l..r]$
1   if ($l < r$)
2       $m = (l + r)/2$; // middle element
3       $RecurSort(l, m)$;
4       $RecurSort(m + 1, r)$;
5       $Merge(l, m, r)$;
```

```
$Merge(l, m, r)$
// Input: array $S[l..r]$ such that $S[l..m]$ and $S[m+1..r]$ are sorted.
// Output: sorted numbers in $S[l..r]$.
1   // create temp array $X[l..r]$ or pass it as a parameter
2   $i1 = l; i2 = m + 1$; // the first elements of the two subarrays
3   for $i = l$ to $r$
4       if ($i1 \le m$ and ($i2 > r$ or $S[i1] < S[i2]$)) // condition for copying $S[i1]$
5           $X[i] = S[i1]$; $i1{+}{+}$;
6       else
7           $X[i] = S[i2]$; $i2{+}{+}$;
8   copy $X[l..r]$ to $S[l..r]$
```

Example: Sort 35,28,42,4,8,83,11 using *MergeSort*.



Let $T(n)$ be the running time of *MergeSort* for a sequence of n numbers.
If $n = 1$ then $T(n) = O(1)$.
If $n > 1$ then $T(n) = 2T(n/2) + O(n)$.

Running time by induction or substitution

We show that $T(n) \le cn \lg n$ for some constant c.

Base case. We can pick c large enough to satisfy $T(n) \le cn \lg n$ for $n = O(1)$. Note that n should be > 1 for this recurrence since $\lg 1 = 0$.

Inductive Step. We assume that $T(n) \le 2T(n/2) + an$.

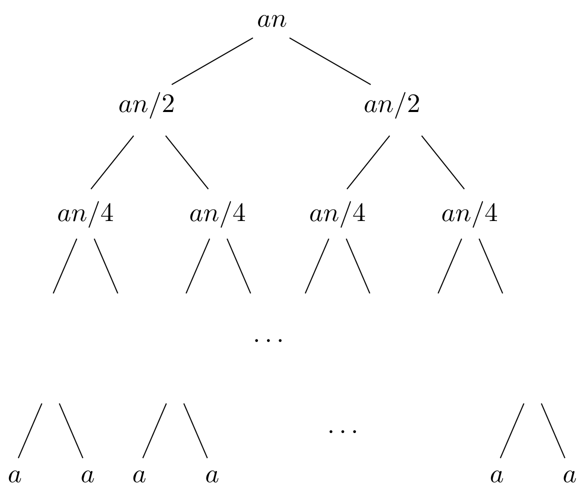
$$\begin{aligned} T(n) &\le 2T(n/2) + an \\ &\le 2c \cdot \frac{n}{2} \lg \frac{n}{2} + an \\ &\le cn(\lg n - 1) + an \\ &\le cn \lg n \qquad \text{if } cn \ge an. \end{aligned}$$

We can pick $c \ge a$ for the inductive step.

The recursion tree is a binary tree. Therefore $T(n) = O(n \lg n)$.

Running time by a recursion tree

Suppose that the running time of *Merge* is an. Then the total time of *MergeSort* is



There are $\lg n$ levels in the tree and the total time at each level is an. Therefore the total time $T(n) \le an \lg n$.

2.2 Closest pair of points

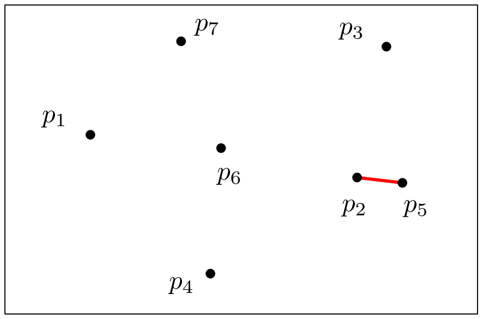
CPP is an abbreviation of closest pair of points.

CPP problem. Given a set $P = \{p_1, \dots, p_n\}$ of $n \geq 2$ points in the plane, find the closest pair $(p_i, p_j), i \neq j$ such that $|p_i p_j|$ is minimum.

We assume that the input is a list of points where each point p_i is given as (x_i, y_i). Then

$$|p_i p_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

Example: The closest pair in the figure below is (p_2, p_5).



CPP problem has applications in traffic-control systems. A system controlling air or sea traffic needs to know which are two closest vehicles in order to detect potential collisions.

A brute-force algorithm checks all $\binom{n}{2} = \Theta(n^2)$ pairs of points. We show a divide-and-conquer algorithm with $O(n \lg n)$ running time. The output is the distance between the points of the closest pair of P.

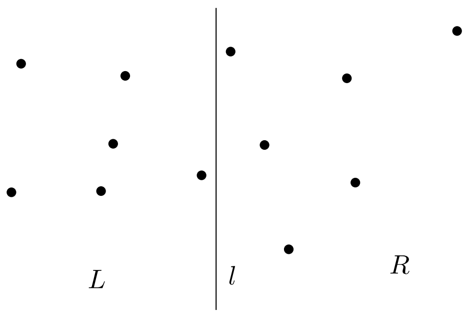
CPP algorithm

1. Divide $P = L \cup R$ by a vertical line l such that $|L| = \lceil n/2 \rceil$ and $|P_R| = \lfloor n/2 \rfloor$.

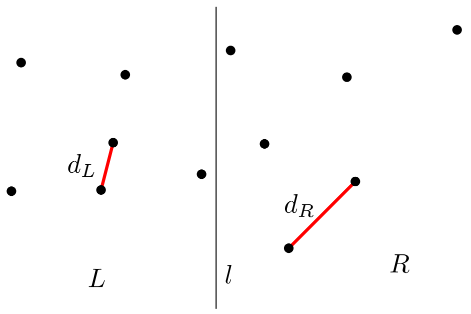
2. Find recursively d_L and d_R, the closest pair distances in L and R.
   Let $d = \min(d_L, d_R)$.

3. Let S be $2d$-wide vertical strip centered at line l.
   Let P' be the array of points $P \cap S$ sorted by y-coordinate.
   For each point p we check only 7 points that follow p in P'.
   Let d' be the shortest distance.
   Return $\min(d, d')$.

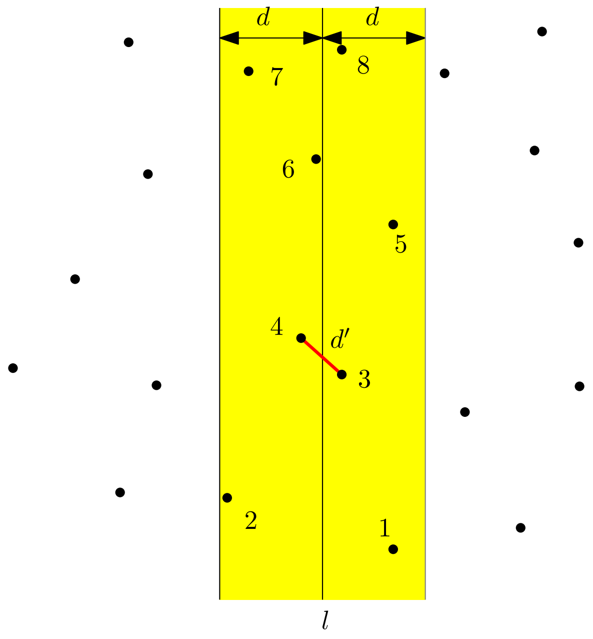
Step 1.



Step 2.



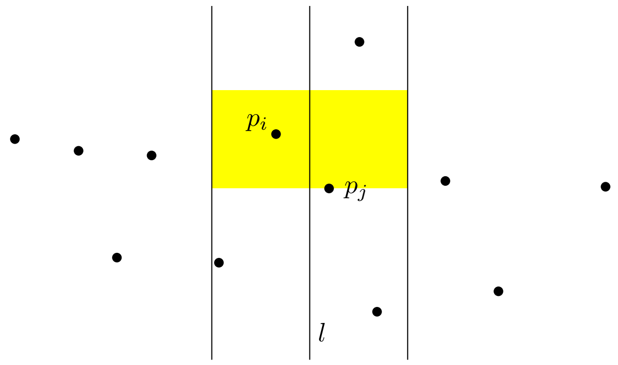
Step 3.



Correctness

If the closest pair lies in L then the algorithm finds it.
If the closest pair lies in R then the algorithm finds it.

Suppose that the closest pair is $(p_i, p_j), p_i \in L, p_j \in R$.
Then $|p_i p_j| < d$ and $p_i, p_j \in S$ and $|y_i - y_j| \leq d$.
There can be at most 8 points in a rectangle $d \times 2d$.



Simple implementation: sort points in $P \cap S$ every time. Then the recurrence is

$$T(n) = 2T(n/2) + O(n \lg n).$$

To solve it we assume that $T(n) \leq 2T(n/2) + an \lg n$. Let $R(n) = T(n)/\lg n$. Then $T(n) = R(n) \cdot \lg n$ and $T(n/2) = R(n/2) \cdot \lg \frac{n}{2} \leq R(n/2) \cdot \lg n$ Then

$$R(n) \leq 2R(n/2) + an.$$

The solution is $R(n) = O(n \lg n)$ and $T(n) = O(n \lg^2 n)$.

Faster implementation: make 3 parameters of $CPP(P, X, Y)$ where array X contains points of P sorted by x-coordinate and array Y contains points of P sorted by y-coordinate.

We need to sort P before the first call $CPP(P, X, Y)$ (presorting). To find l we use the median of X. Make X_L, Y_L and X_R, Y_R in $O(n)$ time. The set P' can be extracted from Y in $O(n)$ time. Then $T(n) = 2T(n/2) + O(n)$ and $T(n) = O(n \lg n)$ which is an improvement.

2.3 Recurrences

Recurrence defines a function $f(n)$ in terms of its values for smaller values of the argument.

Example. *MergeSort*, the worst case running time is

$$T(n) = 2T(n/2) + O(n)$$

It is a simplified form: no base case and no ceiling and floor function.

Substitution Method

It is based on mathematical induction.

1. Guess the solution of the recurrence using unknown constants.

2. Prove your conjecture by induction. Find the values of constants.

Example. $T(n) = 3T(n/3) + 5n$.

Solution. Guess: $T(n) = O(n \lg n)$. We prove $T(n) \le cn \lg n$ by substitution, i.e. using $T(n/3) \le c\frac{n}{3} \lg \frac{n}{3}$.

$$T(n) = 3T(n/3) + 5n \le 3c\frac{n}{3} \lg \frac{n}{3} + 5n = cn \lg(n/3) + 5n = cn \lg n - cn \lg 3 + 5n.$$

If $cn \lg n - cn \lg 3 + 5n \le cn \lg n$, then the proof is finished. Can we guarantee the inequality by selecting an appropriate c? Yes: the inequality $cn \lg n - cn \lg 3 + 5n \le cn \lg n$ is the same as $-cn \lg 3 + 5n \le 0$ or $c \ge 5/\lg 3 \approx 3.15$ We can pick $c = 4$ for example.

Example. $T(n) = 5T(n/5) + 4$.

The correct guess $T(n) = O(n)$. Try substitution method to prove $T(n) \le cn$:

$$T(n) \le 5\left(c \cdot \frac{n}{5}\right) + 4 = cn + 4.$$

So, it doesn't work. But $T(n) \le cn - b$ works:

$$T(n) \le 5\left(c \cdot \frac{n}{5} - b\right) + 4 = cn - 5b + 4 \le cn - b$$

if $b \ge 1$ (this is by solving $cn - 5b + 4 \le cn - b$).

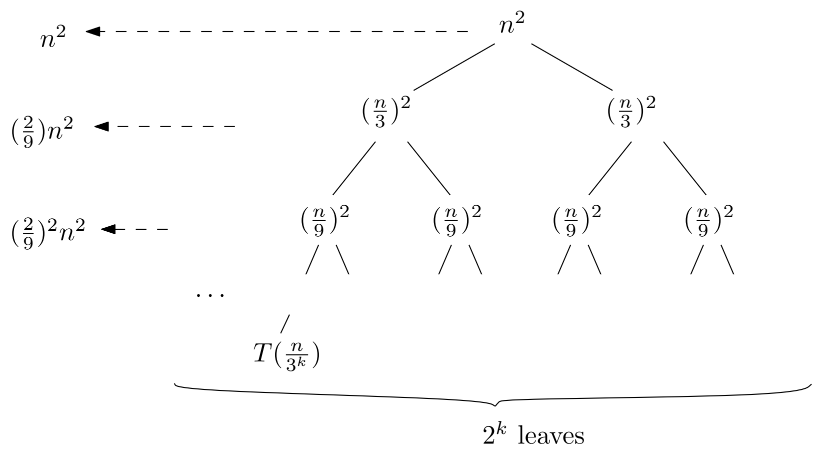
Changing variables and functions

Example. Solve $T(n) = 2T(\sqrt{n}) + \log_3 n$.
Solution. Rename $m = \log_3 n$. Then $n = 3^m$ and $T(3^m) = 2T(3^{m/2}) + m$.
Rename $S(m) = T(3^m)$. The new recurrence is $S(m) = 2S(m/2) + m$.
It is similar to the merge sort. So $S(m) = O(m \lg m)$ and $T(n) = S(\log_3 n) = O(\log_3 n \lg \log_3 n) = O(\lg n \lg \lg n)$.

Recursion-tree method

Example. Solve $T(n) = 2T(n/3) + n^2$.
Solution. The running time can be represented as a tree

Find the total time at each level.



At the bottom level $n/3^k \approx 1$. So, $k = \log_3 n$. Then $2^k = 2^{\log_3 n} = n^{\log_3 2} < n$ and $T(n/3^k) = O(1)$. Total for the bottom level is $O(n)$.

$$\begin{aligned}
T(n) &= n^2 + \frac{2}{9}n^2 + \left(\frac{2}{9}\right)^2 n^2 + \cdots + \left(\frac{2}{9}\right)^{k-1} n^2 + O(n) \\
&= n^2 \sum_{i=0}^{k-1} \left(\frac{2}{9}\right)^i + O(n) \\
&< n^2 \sum_{i=0}^{\infty} \left(\frac{2}{9}\right)^i + O(n) \\
&= \frac{1}{1-2/9} n^2 + O(n) \\
&= O(n^2).
\end{aligned}$$

Master Method

Theorem. Let $a \geq 1$ and $b > 1$ be constants, f be a non-negative function, and $T(n)$ is defined by the recurrence

$$T(n) = aT(n/b) + f(n).$$

Let $d = \log_b a$.
1. If $f(n) = O(n^{d-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^d)$.
2. If $f(n) = \Theta(n^d)$, then $T(n) = \Theta(n^d \lg n)$.
3. If $f(n) = \Omega(n^{d+\varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Example. $T(n) = 100T(n/10) + n \lg n$
Solution. It is Case 1 of Master theorem since $n \lg n = O(n^{2-\varepsilon})$ for $\varepsilon = 1/2$.
$T(n) = \Theta(n^2)$.

Example. $T(n) = 4T(n/2) + 3n^2 + 5n$
Solution. It is Case 2 of Master theorem since $3n^2 + 5n = \Theta(n^2)$.

$T(n) = \Theta(n^2 \lg n)$.

Example. $T(n) = 3T(n/3) + n^{1.5}$

Solution. It is Case 3 of Master theorem since

- $n^{1.5} = \Omega(n^{1+\varepsilon})$ for $\varepsilon = 0.5$, and
- $af(n/b) = 3(n/3)^{1.5} = \dfrac{1}{\sqrt{3}} \cdot n^{1.5} \leq cf(n)$ for $c = \dfrac{1}{\sqrt{3}} < 1$.

Then $T(n) = \Theta(n^{1.5})$.

2.4 Fast Integer Multiplication

Let $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_n$ be n-digit numbers. We assume that n can be large and we store a number as an array of digits. The usual way of computing $X \cdot Y$ takes $O(n^2)$ time (it computes all x_iy_j). We develop a divide-and-conquer algorithm. Divide both X and Y into numbers with "half" of digits. Suppose that n is a power of 2 for simplicity. Let $m = n/2$.

$$\begin{array}{lcl} X_1 = x_1x_2 \ldots x_m & \text{and} & X_2 = x_{m+1}x_{m+2} \ldots x_n \\ Y_1 = y_1y_2 \ldots y_m & \text{and} & Y_2 = y_{m+1}y_{m+2} \ldots y_n \\ X = X_1 10^m + X_2 & \text{and} & Y = Y_1 10^m + Y_2 \end{array}$$

$X \cdot Y = X_1 \cdot Y_1 10^n + (X_1 \cdot Y_2 + X_2 \cdot Y_1)10^m + X_2 \cdot Y_2$

Let $T(n)$ be the time for multiplying n-digit numbers. Multiplication by 10^n and 10^m is just a shift of digits and can be done in $O(n)$ time. Additions take at most cn time. Then

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

Using Master method, case 1, the running time is $T(n) = \Theta(n^2)$ which is not faster than the usual way. New idea: using

$$X_1 \cdot Y_2 + X_2 \cdot Y_1 = (X_1 + X_2) \cdot (Y_1 + Y_2) - X_1 \cdot Y_1 - X_2 \cdot Y_2$$

we get only three multiplications.

New recurrence is $T(n) = 3T(n/2) + cn$. Solving by Master theorem, case 1: $T(n) = \Theta(n^{\lg 3})$, $\lg 3 \approx 1.5849625 < 2$ which is an improvement.

Exercises

2.1.

Find the best-case running time of *MergeSort*? Show that *MergeSort* can be modified such that (a) the worst-case running time is the same, and (b) the best-case running time is $O(n)$.

2.2.

Sort 4,40,51,17,33,55,11,48 using *MergeSort* as in the example in Section 2.1.

2.3.

Solve the following recurrences using the master method

a) $T(n) = 2T(n/4) + 7$.

b) $T(n) = 3T(n/9) + \sqrt{n}$.

c) $T(n) = 2T(n/4) + n \lg n$.

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

2.4.

Solve the following recurrence using the recursion-tree method $T(n) = 3T(n/3) + n^2$.

2.5.

Let $T(n)$ be the running time of $\textsc{Alg1}$ called for $l = 0$ and $r = n - 1$. Write the recurrence for $T(n)$, solve it and give a big-Theta bound.

```
$\textsc{Alg1}(A, l, r)$ // input: array $A[l..r]$
1   if $r \le l$ then return 0
2   $s = 0, n = r - l + 1$
3   for $i = l$ to $r$
4       $s += A[i]$
5   $s += \textsc{Alg1}(A, l, r - n/3)$
6   $s += \textsc{Alg1}(A, l + n/3, r)$
7   return $s$
```

2.6. *CPP algorithm analysis*

Prove that it suffices to check only 5 points (instead of 7) in step 3 of the Closest-Pair Algorithm.

2.7. *CPP algorithm analysis*

The example with 8 points in the $\delta \times 2\delta$ box has points on l from both L and R. A modification of the Closest-Pair Algorithm was suggested by

(1) assigning points on line l to set L, and

(2) checking only 5 points in step 3.

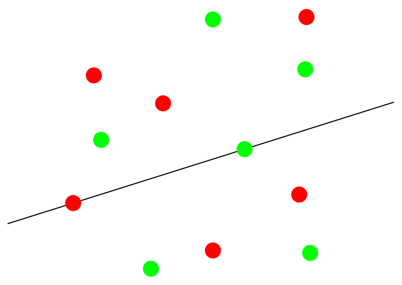
What is the flaw in the new algorithm?

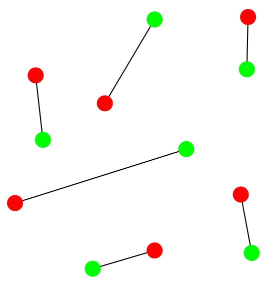
2.8. *Red-green points*

Let R be a set of n red points and G be a set n green points in the plane such that no three points in $R \cup G$ are collinear. The task is to connect every red point to a green point by a straight-line segment such that any two segments do not intersect and every green point is connected to a red point.

a) Show that there always exists a line passing through one red and one green point the number of red points on one side of the line is equal to the number of green points on the same side. Describe how to find such a line in $O(n \lg n)$ time.

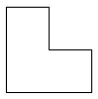
b) Show that n segments connecting red and green points can be computed in $O(n^2 \lg n)$ time.

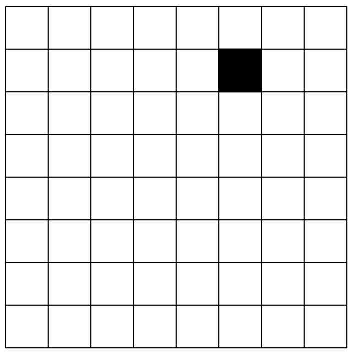

Example for (a).

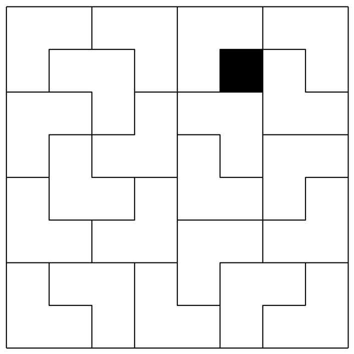

Example for (b).

2.9. *Grid jigsaw puzzle*

A jigsaw puzzle is the $2^n \times 2^n$ grid with one square missing and the jigsaw pieces are formed by 3 adjacent squares. Show that, for any $n \geq 1$ and any missing square, the puzzle can be solved by a divide-and-conquer algorithm.


Jigsaw shape


Jigsaw puzzle


A solution

Chapter 3

Selection Problem

The selection problem: Find kth smallest element in array S of n distinct elements.

Easy cases: $k = 1$ and $k = n$. min and max.

Difficult case: median $k = (n + 1)/2$ if n is odd. If n is even, then there are 2 medians: lower median for $k = n/2 - 1$ and upper median for $k = n/2 + 1$.

The selection problem can be solved by sorting in $O(n \lg n)$ time. We show
(1) a practical algorithm with $O(n)$ *expected* running time (Section 3.2), and
(2) a theoretical algorithm with $O(n)$ running time in the *worst case* (Section 3.3).

First, we consider the problem in terms of comparisons.

3.1 Counting comparisons

Suppose we want to compute the minimum of n numbers using only comparisons of given numbers. Find an algorithm that has as few comparisons as possible. If the numbers are given in array $S[0..n-1]$ then the usual way of computing the minimum is as follows.

```
$Minimum(S)$
// Input: array $S[0..n-1]$ of $n$ numbers.
// Output: the index of the minimum number in $S[0..n-1]$.
1   $ind = 0$ // current min is $S[0]$
2   for $i = 1$ to $n - 1$
3       if $S[ind] > S[i]$ then $ind = i$
4   return $ind$
```

The number of comparisons (line 3) is $n - 1$.

Example. Show that *any* algorithm for computing the minimum uses at least $n-1$ comparisons.

Two approaches discussed in a class: 1) If an algorithm uses less than $n - 1$ comparisons, then some number will be not compared to any other number. 2) Any algorithm for computing the minimum compares the minimum to all other $n - 1$ numbers.

Are these statements true or false? How many comparisons are made by some algorithm if we only know that it compares every number at least one time?

Min and max

It can be done using $2n-3$ comparisons: first compute min, then compute max among the remaining $n-1$ numbers. Then the number of comparisons is $(n-1)+(n-2) = 2n-3$. Can we use fewer comparisons?

One idea is to compare $A[0]$ and $A[1]$ and use the smallest number as a candidate for the minimum and the largest number as a candidate for the maximum. For each remaining $n-2$ numbers, we check if the candidates need to be updated. This requires ≤ 2 comparisons for each number. The total number is $1 + 2(n-2) = 2n-3$ which is actually the same as before.

Example. Show that min and max can be computed using $3\lfloor n/2 \rfloor$ comparisons.

Solution. Pair numbers and find the smallest number in each pair. Find the minimum among the smallest numbers in pairs and the maximum among the largest numbers in pairs. If n is even then we are done. If n is odd then compare the remaining number with the current minimum and maximum.

Example. Show that this algorithm has $\leq 3\lfloor n/2 \rfloor$ comparisons.

Min and second min

Example. Show that the smallest and the second smallest of n numbers can be found with at most $n + \lg n$ comparisons in the worst case.

It is known that min and second min can be computed in $n + O(\lg \lg n)$ expected time.

Medians

- The median can found using $(3-\delta)n$ comparisons.
- Any algorithm for finding the median has $\geq (2+\varepsilon)n$ comparisons in the worst case.
- **Conjecture**. The median can found using cn comparisons where $c = \log_{4/3} 2$.

3.2 Probabilistic Selection

The idea is to pick a random element, called pivot, and find its position in the sorted array. If it is the kth smallest element then return the pivot, otherwise recurse on either the low side (the numbers $\leq pivot$) or the high side (the numbers $\geq pivot$). The following code is a recursive procedure. The first call is $Select(0, n-1)$. Function $Partition(l, r)$ picks a random $pivot$ in $S[l..r]$ and rearrange $S[l..r]$: the low side, followed by $pivot$, followed by the high side. It returns the final position of the pivot.

```
$Select(l, r)$
// Input: $S[l..r]$ is the array of numbers and $k$ such that $k - 1 \in [l, r]$.
//    (both $S$ and $k$ are external)
// Output: number at position $k - 1$ of $S[l..r]$ if $S[l..r]$ would be sorted
1   if $l = r$ then return $S[l]$
2   $p = Partition(l, r)$   // the pivot is at $S[p]$ after the partition
3   if $p = k - 1$ then return $S[k - 1]$
4   if $i < p$ then return $Select(l, p - 1)$ // search on low side
5   else return $Select(p + 1, r)$ // search on high side
```

```
$Partition(l, r)$
// Input: array $S[l..r]$.
// Output: partition numbers of $S[l..r]$ according to a random pivot
// (the low side, the pivot, and the high side)
1   $j = Random(l, r)$   // $j \in \{l, l + 1, \ldots, r\}$
2   exchange $S[r]$ and $S[j]$   // place pivot at the end
3   $pivot = S[r]$   // this is the pivot
4   $last = l - 1$   // $last$ points to the last number $\leq x$
5   for $j = l$ to $r - 1$
6     if $S[j] \leq pivot$ then
7       $last = last + 1$
8       if $last < j$ then   // keep $S[j]$ in the low side
9         exchange $S[last] \leftrightarrow S[j]$
10  if $last + 1 < r$ then   // place the pivot after the low side
11    exchange $S[last + 1] \leftrightarrow S[r]$
12  return $last + 1$
```

Example. Show the performance of *Select*() on the array $S = \{7, 4, 8, 2, 5, 9, 1, 3, 6\}$, $n = 9$ and $k = 3$ (show $S[l..r]$ for every call of *Select*()). Assume that the pivot is always selected at position l.

Solution. $k = 4$, so we want to find the fourth smallest number. Then $i = k - 1 = 2$ meaning that we want to find $S[2]$ is S would be sorted. The first time $l = 0, r = 8$. The pivot is $S[0] = 7$. The array after the partitioning is $S = \{6, 4, 2, 5, 1, 3, \mathbf{7}, 9, 8\}$ and $p = 6$. We continue on the low side.
Next time (next recursive call) $l = 0, r = 5, S[l..r] = \{6, 4, 2, 5, 1, 3\}$. The pivot is $S[0] = 6$. The array after partition is $S[l..r] = \{3, 4, 2, 5, 1, \mathbf{6}\}$.
Next time (next recursive call) $l = 0, r = 4, S[l..r] = \{3, 4, 2, 5, 1\}$ The pivot is $S[0] = 3$. The array after partition is $S[l..r] = \{1, 2, \mathbf{3}, 5, 4\}$. Since $p = i$, return $S[2] = 3$.

Running time

First observe that lines 1-3 of *Select* take $O(n)$ time.

The worst case running time.

In the worst case the pivot is always the smallest (or the largest) number in $S[l..r]$. The worst case (for example, $k = n$ (for computing the max number) and the pivot is always the smallest number, i.e. $j = l$ in all recursive calls) recurrence is then $T(n) = T(n-1) + an$. The solution is $T(n) = \Theta(n^2)$ (why?).

Expected time.
The input size n' of the recursive call of *Select* depends on p and i:

pivot position p	0	1	$\dots$	$i-1$	i	$i+1$	$i+2$	$\dots$	$n-1$
next input size n'	$n-1$	$n-2$	$\dots$	$n-i$	0	$i+1$	$i+2$	$\dots$	$n-1$

Then $T(n) \le an + T(n'(p,i))$. Let $E(n)$ denotes the expected running time. Since all values of the pivot are equally likely,

$$E(n) \le an + \frac{1}{n}\sum_{p=0}^{n-1} E(n'(p,i)).$$

We prove by induction that $E(n) \le cn$ for some constant $c > 0$. Pick $c = 4a$ in the inductive step:

$$E(n) \le an + \frac{1}{n}\sum_{p=0}^{n-1} c \cdot n'(p,i) = an + \frac{c}{n}\sum_{p=0}^{n-1} n'(p,i) \le cn$$

if we show that $\sum_{p=0}^{n-1} n'(p,i) \le (1 - a/c)n^2 = \frac{3}{4}n^2$. From the table above and the summation

$$a + (a+1) + \dots b = (a+b)(b-a+1)/2$$

(for any integers $a \le b$), we conclude

$$\begin{aligned} 2\sum_{p=0}^{n-1} n'(p,i) &= (2n-i-1)i + (n+i)(n-i-1) \le (2n-i)i + (n+i)(n-i) \\ &= 2ni - i^2 + n^2 - i^2 = 2i(n-i) + n^2. \end{aligned}$$

Since $(n/2 - i)^2 = (n/2)^2 - i(n-i) \ge 0$, we have $i(n-i) \le n^2/4$. Then

$$2\sum_{p=0}^{n-1} n'(p,i) \le n^2/2 + n^2 = \frac{3}{2}n^2 \quad \text{and} \quad \sum_{p=0}^{n-1} n'(p,i) \le \frac{3}{4}n^2.$$

The end of the proof.

3.3 Deterministic Algorithm

We show that the selection problem can be solved in $O(n)$ time (in the worst case!). The idea is to pick the pivot not randomly but using a recursive call.

1. Divide n elements into $\lceil n/5 \rceil$ groups of at most 5 elements each.
2. Find median in each group by sorting it.
3. Find median-of-medians x recursively (call this program for the array of medians).
4. Partition the input array A using pivot x. Suppose that x is the ith smallest element of S.
5. If $i = k$ then return x. Otherwise recursively find kth element on the low side of the partition if $i < k$, or $(k - i)$th element on the high side.

Example

Compute 10th smallest number in an array A with 53 numbers
$A = \{33, 22, 30, 14, 45, 10, 44, 23, 9, 39, 38, 52, 6, 5, 50, 37, 11, 26, 3, 15, 2, 53, 40, 54, 25, 55, 12, 19, 31, 16, 18, 13, 1, 48, 41, 24, 43, 46, 47, 17, 34, 20, 31, 32, 33, 35, 4, 49, 51, 7, 21, 27, 8\}$

Step 1. Divide n elements into $\lceil n/5 \rceil$ groups.

33	10	38	37	2	55	18	24	34	35	21
22	44	52	11	53	12	13	43	20	4	27
30	23	6	26	40	19	1	46	31	49	8
14	9	5	3	54	31	48	47	32	51	
45	39	50	15	25	16	41	17	33	7	

2. Find the median in each group by sorting it.

14	9	5	3	2	12	1	17	20	4	
22	10	6	11	25	16	13	24	31	7	8
30	23	38	15	40	19	18	43	32	35	21
33	39	50	26	53	31	41	46	33	49	27
45	44	52	37	54	53	48	47	34	51	

3. Find median-of-medians recursively, i.e. find 6th smallest number in the array [30,23,38,15,40,19,18,43,32,35,21]. $x = 30$.

14	9	5	3	2	12	1	17	20	4	
22	10	6	11	25	16	13	24	31	7	8
30	23	38	15	40	19	18	43	32	35	21
33	39	50	26	53	31	41	46	33	49	27
45	44	52	37	54	53	48	47	34	51	

4. Partition the array A using pivot x. Let x be kth element of A.
$Low = \{22, 14, 10, 23, 9, 6, 5, 11, 26, 3, 15, 2, 25, 12, 19, 16, 18, 13, 1, 24, 17, 20, 4, 7, 21, 27, 8\}$
$High = \{33, 45, 44, 39, 38, 52, 50, 37, 53, 40, 54, 55, 31, 48, 41, 43, 46, 47, 34, 31, 32, 33, 35, 49, 51\}$

5. Recursively find 10th element in Low.

Running Time

Let $T(n)$ be the running time for computing the kth smallest element of n numbers. We analyze the running time by steps.

Step 1 and 2: $O(n)$ time. Step 3: $T(\lceil \frac{n}{5} \rceil)$ time. Step 4: $O(n)$ time.

Step 5: $T(\max(|L|, |H|))$ time where L is the low side and H is the high side. We will show later that $|L|, |H| \leq 7n/10 + 5$.

Then we write a recurrence $T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(7n/10 + 5) + an$.

We solve a simplified recurrence $T(n) \leq T(n/5) + T(7n/10) + an$.

We pick the constant $c = 10a$ and show by induction that $T(n) \leq cn$.

$$\begin{aligned} T(n) &= T(n/5) + T(7n/10) + an \\ &\leq cn/5 + c(7n/10) + an \\ &= 9cn/10 + an \\ &= 9cn/10 + cn/10 \\ &= cn. \end{aligned}$$

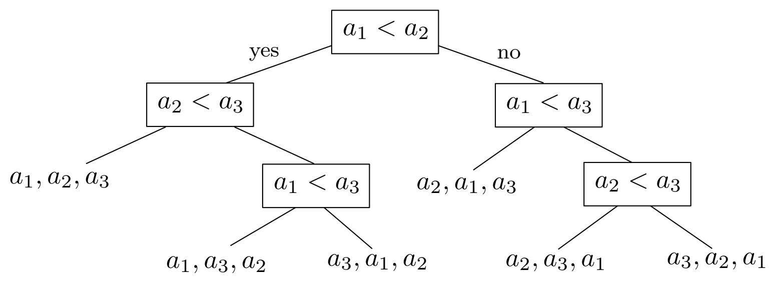
Now, we show the bounds for L and H. About half of the columns (excluding the one with x and the last one) have 3 elements $\geq x$. Then that $|H| \geq 3(\lceil \frac{m}{2} \rceil - 2)$ where m is the number of columns. Then $|H| \geq \frac{3}{2}m - 6 \geq \frac{3}{10}n - 6$ since $m = \lceil \frac{n}{5} \rceil \geq \frac{n}{5}$. Then $|L| \leq n - 1 - |H| = 7n/10 + 5$ (it's $n - 1$ since the pivot is not included). Similarly $|H| \leq 7n/10 + 5$. So, step 5 takes at most $T(7n/10 + 5)$ time.

3.4 Lower bound for sorting

Let $a_1, a_2, \ldots, a_n$ be a sequence of n numbers to sort. Most of the sorting algorithms are based on comparisons of the given numbers. For a given n, the algorithm can be viewed as a decision tree.

Example. Show a decision tree for sorting 3 numbers. What is the best case and the worst case for the number of comparisons?

Solution.



2 comparisons in the best case ($a_1 < a_2 < a_3$) and 3 comparisons in the worst case ($a_1 < a_3 < a_2$).

A sorting algorithm is called comparison-based if it can only gain information about the elements by comparing two of them.

Example. Show that any comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Solution. Let h be the height of the decision tree of a comparison-based sorting algorithm for some n. The number of comparisons in the worst case is equal to h. The decision tree is binary and the number of leaves l is at most 2^h (why?). On the other hand, the number of leaves is at least $n!$ since the algorithm is correct (why?). So, $n! \leq l \leq 2^h$ and $h \geq \lg(n!) = \Omega(n!)$.

Exercises

3.1.

Find the 2nd smallest number in $\{7, 4, 8, 2, 5, 9, 1, 3, 6\}$. Show $S[l..r]$ for every call of *Select* and after *Partition*. Assume that the pivot is always $S[l]$.

3.2.

Use random variables to solve the following problem. Each of k customers in a rainy day gives an umbrella to an umbrella-check person in a cinema. The umbrella-check person gives the umbrellas back to the customers in a random order. What is the expected number of customers who get back their own umbrella?

3.3.

Suppose that we change the deterministic selection algorithm and use groups of 3 instead of groups of 5.

(a) Describe a worst case scenario (the value of k and the values of i for pivots)?

(b) Write the recursion for the worst-case running time $T(n)$.

(c) Show that there exist some constants $a, b > 0$ such that

$$an \lg n \leq T(n) \leq bn \lg n$$

for all $n \geq 2$.

3.4.

Show that the smallest and the second smallest of n numbers can be found with at most $n + \lg n$ comparisons in the worst case.

3.5.

Given an array A of n numbers, a pivot (used in *Select*) is called good if the low side of the partition and the high side of the partition contain at least $n/4$ elements each. Show that all good pivots can be computed in $O(n)$ time.

3.6.

Prove that any comparison-based algorithm to sort 4 numbers requires 5 comparisons in the worst case.

3.7.

Show that 4 numbers can be sorted using a decision tree of height 5.

Chapter 4

Greedy Algorithms

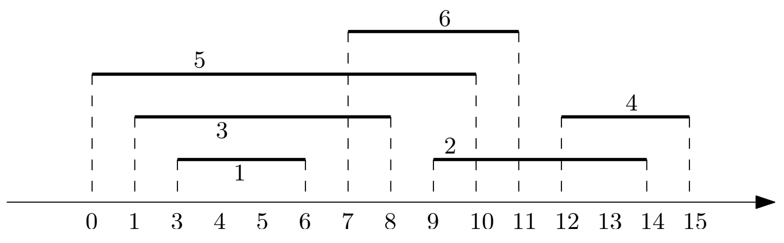
- Scheduling problem
- Knapsack Problem
- Huffman code

4.1 Scheduling problem

Consider n activities $1, 2, \ldots, n$ where k-th activity starts at time s_k and finishes at time f_k. Suppose that they require the same resource (for example, one printer). Then two activities i and j are non-conflicting if $s_i \geq f_j$ or $s_j \geq f_i$.

The scheduling problem is to find a maximum size subset of non-conflicting activities.

Example. Solve the scheduling problem for the following $n = 6$ activities.



Solution. We cannot pick 4 activities since
• only one activity can be selected from $\{1, 3, 5\}$, and
• we cannot pick all activities from the other activities.
Solution with 3 activities: $1, 4$, and 6.

Greedy Approach

Greedy choice. Select an activity with minimum f_i. Remove it from the set and the activities conflicting with it. Continue until no more activities left.

We prove that the set S_g computed by the greedy algorithm is optimal. Suppose that i is an activity in S_g with minimum f_i. Consider any optimal solution S and let k be an activity in S with minimum f_k.

Example. Suppose that $i \neq k$. Show that $i \notin S$. Let S' be the set S with k replaced by i. Show that S' is a solution of the scheduling problem.

Therefore, we can assume that $i = k$ (by considering S' instead of S).

Example. Suppose that $i \in S$. Show that $S_g - \{i\}$ and $S - \{i\}$ are the solutions of the same problem and $S_g - \{i\}$ is computed by the greedy algorithm for it.

By repeating this argument we see that $|S_g|$ must be equal to $|S|$.

Fast algorithm

To make an efficient algorithm we sort the activities by finish times first. We discard activities on-the-fly by checking them in the sorted order. To discard an activity we can just check if it conflicts with the last selected activity.

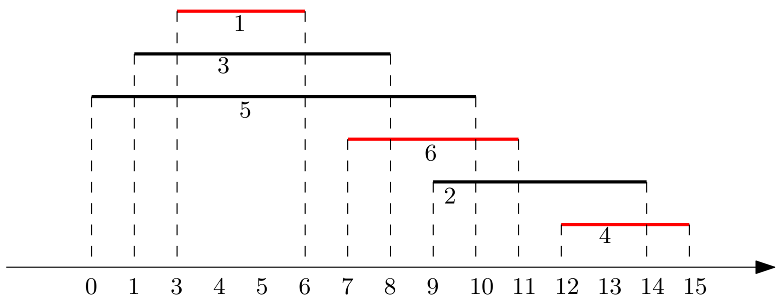
```
*Greedy-Schedule*$(s, f)$
// Input: $s[1..n]$ is the array of start times, $f[1..n]$ is the array of finish times.
// Activities are sorted by finish times.
1   $S = \{1\}$; $f = f[1]$;    // $f$ is the finish time of the last selected activity
2   for $j = 2$ to $n$
3      if $s[j] \geq f$
4         $S = S \cup \{j\}$
5         $f = f[j]$
```

It is correct (as the recursive algorithm).

The running time is $O(n)$ (assuming that the activities are sorted by finish time).

Example. Run the greedy scheduling algorithm on the set of activities from the example above.

Solution. The sorted order is $1, 3, 5, 6, 2, 4$. The greedy selection is $1, 6, 4$.



4.2 General Greedy Approach

Try to analyze the structure of an optimal solution especially find an optimal substructure. Typically it is a "cut-and-paste" idea. For example, in the scheduling problem, it was helpful to replace activity k by activity i in S. This is a "cut-and-paste" that preserves the optimality of the solution. This can be viewed as an idea for the greedy choice.

The second important part of the greedy scheduling algorithm was the property that, after the first greedy selection, an optimal solution for the remaining problem combined with the the first selection is actually an optimal solution of the entire problem. This is the optimal substructure property. It is common for many greedy algorithms.

4.3 Knapsack Problem

There are n items; ith item is worth v_i dollars and weights w_i pounds, where v_i and w_i are integers. Select items to put in knapsack with total weight $\leq W$ so that total value is maximized.

0-1 knapsack problem: each item must either be taken or left behind.

Fractional knapsack problem: fractions of items are allowed.

Greedy choice: take an item with maximum value per pound.

0-1 Knapsack Problem

i	1	2	3
w_i	2	4	6
v_i	\$10	\$16	\$18
value per pound	\$5	\$4	\$3

$W = 10$

Greedy algorithm: item 1 and then item 2. Total value \$26.

Optimal solution items 2 and 3. Total value \$34.

The greedy approach doesn't work for 0-1 knapsack problem.

Fractional Knapsack Problem

i	1	2	3
w_i	2	4	6
v_i	\$10	\$16	\$18
value per pound	\$5	\$4	\$3

$W = 10$

Greedy algorithm: item 1, item 2 and 2/3 of item 3. Total weight 10. Total value \$38.

Optimal!

Greedy choice works for the fractional knapsack problem.

4.4 Huffman Coding

Suppose we want to compress a file using binary codes where each symbol corresponds to a binary code called codeword.

Example. Suppose that a text file contains only characters a, b and c, 100 each. Compress it with codewords of fixed length. What is the size of the compressed file?

Solution. Fixed-length code example $a = 00, b = 01, c = 11$. Then $acb = 001101$. Decode 010011=? The size is 300*2=600 bits.

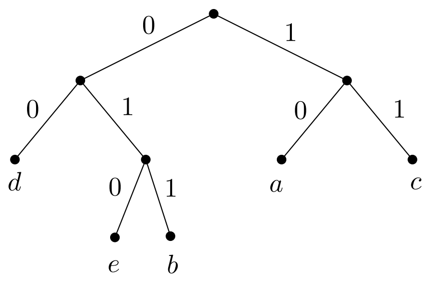
Example. Show that a variable-length code can do better.

Solution. Example $a = 0, b = 10, c = 11$. Then $acb = 01110$. Decode 010100=? The size is 100+200+200=500 bits.

A binary sequence can be decoded if no codeword is a prefix of another codeword. We call such code a prefix code. It can be represented by a binary tree.

Example. Find codewords of the letters using the following tree. Encode ace. Decode 0100010.

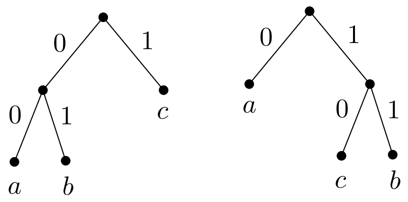
Solution. $a = 10, b = 011, c = 11, d = 00, e = 010$. Then $ace = 1011010$ and $0100010 = 010\ 00\ 10 = eda$.



Example. A text contains only letters a, b, c with frequencies $f(a) = 10, f(b) = 2, f(c) = 6$. Find the length of the encoded text using prefix codes corresponding to 2 trees.

Solution. Left tree: $a = 00, b = 01, c = 1$ and the length is $2f(a) + 2f(b) + f(c) = 30$.

Right tree: $a = 0, b = 11, c = 10$ and the length is $f(a)+2f(b)+2f(c) = 26$. The code for this tree is shorter.

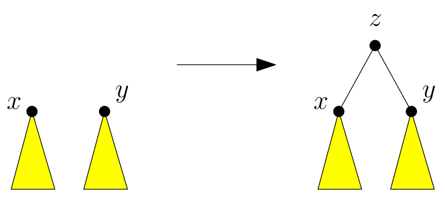


Optimal Code

Given an alphabet C of n characters and a frequency of each character in a text, find an optimal prefix code, i.e the encoded file should have minimum number of bits.

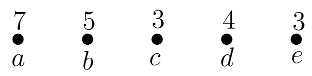
Huffman's algorithm constructs the optimal tree T. The characters are at the leaves of T.

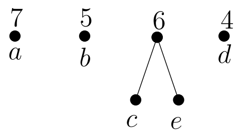
Greedy choice: select two vertices x and y of lowest frequency and replace them by a vertex z so that x and y are the children of z and the frequency of z is the sum of frequencies of x and y.

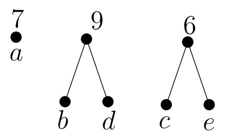


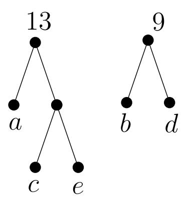
Huffman Code

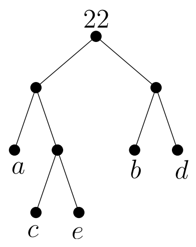
character	a	b	c	d	e
frequency	7	5	3	4	3

1)


2)


3)


4)


5)


Huffman's algorithm

```
HUFFMAN($C, f$)
// Input: array $c[1..n]$ of characters and array $f[1..n]$ of frequencies
// Output: Tree $T$
1   Initialize empty priority queue $Q$
2   for $i = 1$ to $n$
3      Create a vertex for $c[i]$ and $f[i]$
4      Insert it into $Q$ with key $f[i]$
5   while $Q.size() > 1$
6      $v1 = RemoveMin()$;
7      $v2 = RemoveMin()$;
8      Create a vertex with left child $v1$ and right child $v1$
9      Insert it into $Q$ with key $v1.freq + v2.freq$
10  return $RemoveMin()$; // the root of the tree
```

Using a binary min-heap, the initialization in line 2 takes $O(n)$ time.
Every *RemoveMin* and *Insert* takes $O(\lg n)$ time.
The total time is $O(n \lg n)$.

Huffman Code

Let $L(T)$ be the cost of a binary tree T (the length of the encoded text in bits), i.e.

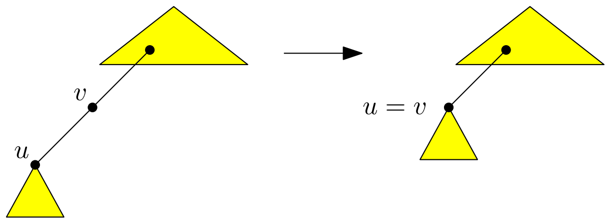
$$L(T) = \sum_{i=1}^{n} f[i] \cdot d[i]$$

where $f[i]$ is the frequency of ith character and $d[i]$ is its depth in T.

Theorem. Huffman's algorithm computes an optimal code.

Proof. Let T be an optimal tree.

Part 1. T is full, i.e. a vertex u cannot be the only child of its parent v. Indeed, we can remove such a vertex as shown in the figure.

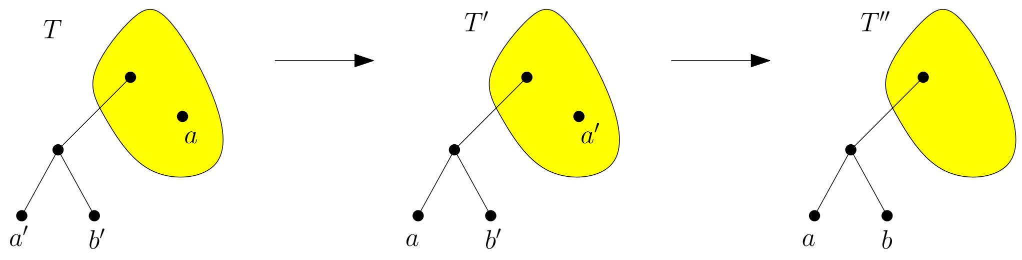


Part 2. Let a be the least frequent characters in C. Let a' and b' be siblings of maximum depth (they exist by Part 1). Then $L(T) = X + f[a'] \cdot d' + f[a] \cdot d$ where $d' = d[a'], d = d[a]$ and X is the cost of all characters except a and a' in T.

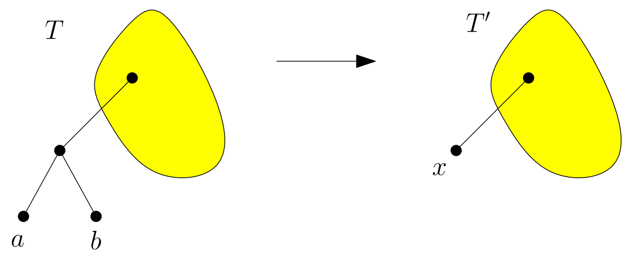
Make a tree T' by exchanging a and a'. Then $L(T') = X + f[a] \cdot d' + f[a'] \cdot d$. Then

$$L(T) - L(T') = f[a'] \cdot d' + f[a] \cdot d - f[a] \cdot d' - f[a'] \cdot d = (f[a'] - f[a])(d' - d) \geq 0.$$

So, T' is not worse than T. We also can replace b' by b. This justifies the first step of the greedy algorithm.



Part 3. By Part 2, we want to find a tree T minimizing $B(T)$ with a constraint that a and b are siblings in T. Let T' be the binary tree $T - \{a, b\}$ and let x be the new leaf and $f(x) = f(a) + f(b)$. Minimizing $L(T)$ is the same as minimizing $L(T')$ since $L(T) = L(T') + f[a] + f[b]$ and $f[a] + f[b]$ is a constant.



Exercises

4.1.

There are n customers at a bank and only one cashier. An ith customer $1 \leq i \leq n$ needs a service time t_i. Design a greedy algorithm to order the customers such that the average waiting time is minimized. The waiting time of a customer is from the opening time of the bank untill (s)he leaves the bank.

4.2.

Consider the following modifications of the greedy scheduling algorithm. Do they solve the scheduling problem? Argue if yes, show a counterexample if not.

(a) The change: select an activity with earliest start time instead of activity with min f_i.

(b) The change: select an activity with smallest duration $f_i - s_i$ instead of activity with min f_i.

4.3.

Show that 0-1 knapsack problem can be solved in $O(nW)$ time. *Hint: Make a dynamic program.*

4.4.

Consider the following input text: aaabbbbccccccdddddd. Find frequencies of each symbol in the text. Find a Huffman tree for these frequencies. Show codewords for each symbol. Encode the text. How many bits are in the encoded text?

Chapter 5

Dynamic Programming

Dynamic programming is a powerful technique for solving some optimization problems (not all). Optimization problem: each solution has a value, and we wish to find a solution with optimal value (minimum or maximum). This is an optimal solution.

Two important steps in designing a dynamic program:

1. Express an optimal solution using optimal solutions of smaller problems.
2. Describe all sub-problems if recursion is applied.

There two ways of implementing :

1. Bottom-up approach. Solve smaller sub-problems first. Solve bigger subproblems later. This is dynamic programming. .

2. Top-down approach. Direct recursive algorithm. Memoization.

We study the following problems

- Rod-cutting problem
- Longest common subsequence
- Optimal binary search tree

5.1 How to cut a rod?

The cutting problem: How to cut a rod of length n into (integer length) pieces in order to maximize the total cost of pieces? A piece of length $i, 1 \leq i \leq n$ costs $p[i]$ dollars.

Example. Solve the cutting problem for $n = 4$ using the following price table.

length i	1	2	3	4
price	3	7	8	9

Solution. There are 4 different outcomes of the cutting: $1+1+1+1, 2+1+1, 2+2, 3+1, 4$. $2+2$ is optimal (the only solution).

Suppose we try all possible cuttings to solve the problem. How many ways are there to cut a rod of length n? 2^{n-1} (why?)

But some outcomes are counted more than 1 time, for example $1 + (n - 1)$ and $(n - 1) + 1$.

Suppose that we count only different outcomes of the cutting. Still many. $\approx c^{\sqrt{n}}$ outcomes where $c > 1$ is a constant.

Solution

Idea: Let $c_k, 1 \leq k \leq n$ be the max cost of cutting a rod of length k. Consider an optimal cutting of rod of length n. If the first piece has length i, then the rest is cut optimally with cost c_{n-i}. Therefore

$$c_n = \max_{1 \leq i \leq n} (p_i + c_{n-i}).$$

Notice that, if $i = n$, then $c_{n-i} = c_0 = 0$ (no cuts).

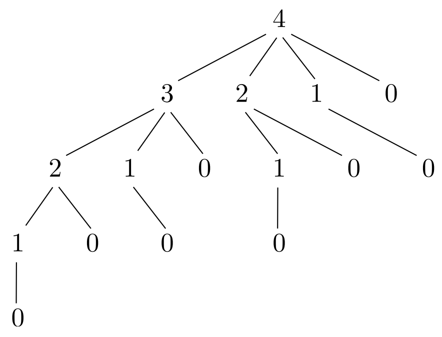
Recursive algorithm

```
RecurCut (n)
1   if $n = 0$ then return 0 // the base case
2   $cost = 0$ // initialize $cost$ to find max cost
3   for $i = 1$ to $n$
4       $cost = \max(cost, p[i] + RecurCut\ (n - i))$
5   return $cost$
```

RecurCut takes an *hour* for $n = 40$. Why is it so slow? Recalculations.

Example. Show the recursion tree of *RecurCut* when it is called for $n = 4$. How many times *RecurCut* (0) is called?

Solution. We show n at every vertex (the input of the corresponding recursive call). The children of a vertex correspond to calls for $i = 1, 2, \ldots, n$.



Example. How many calls to *RecurCut* are made if the first call is made for n and there are prices for $i = 1, 2, \ldots, n$?

Solution. Let $T(n)$ be the number of calls. The tree above shows that $T(4) = 16$. We prove that $T(n) = 2^n$ by induction.
Base case $n = 0$. $T(0) = 1$.
Inductive step. Let $j = n - i$ (in line 4 of $RecurCut$). Then $T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + (2^n - 1) = 2^n$ (geometric series).

Two ideas to avoid recomputing

- top-down with memoization, and
- bottom-up method.

Memoization

Memoization is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously-processed inputs[1]. A memoized function "remembers" the results corresponding to some set of specific inputs. For the cutting problem, we store a lookup table $c[..]$ where $c[i]$ is the max cost for a rod of length i. Initially, $c[i] = 0$ for all i.

```
$MemoizedCut(n)$
1  if $c[n] > 0$ then return $c[n]$     // already computed
2  if $n = 0$ then return $0$     // the base case
3  $cost = 0$ // to compute the max cost
4  for $i = 1$ to $n$
5     $cost = \max(cost, p[i] + MemoizedCut\,(n - i))$
6     $c[n] = cost$
7  return $cost$
```

Running time?

Bottom-up method

```
$CutRod(n)$
1  make a new array $c[0..n]$
2  $c[0] = 0$
3  for $j = 1$ to $n$
4     $cost = 0$
5     for $i = 1$ to $j$
6        $cost = \max(cost, p[i] + c[j - i])$
7     $c[j] = cost$
8  return $c[n]$
```

[1]Wikipedia. Memoization is from memo, not memory. So, no "r".

Example. Modify *CutRod* to print the best cutting of a rod of length n.

Solution. We store in $t[j]$ the cut used in computing $c[j]$ for all j.

```
PrintCutting($n$)
make new arrays $s[0..n]$ and $t[0..n]$
$c[0] = 0$
for $j = 1$ to $n$
    $cost = 0$
    for $i = 1$ to $j$
        if $cost < p[i] + c[j-i]$
            $cost = p[i] + c[j-i]$
            $t[j] = i$
    $c[j] = cost$
// print the solution
$n1 = n$ // current length
while $n1 > 0$
    print $t[n1]$
    $n1 = n1 - t[n1]$
```

5.2 Longest Common Subsequence

Suppose we have strings $X[1..m]$ and $Y[1..n]$.

A string Z is a subsequence of X if it can be derived from X by deleting some characters without changing the order of remaining characters. For instance, "CS", "S4", "39", "C44" are subsequences of "CS4349".

LCS problem. Given two strings $X[1..m]$ and $Y[1..n]$, find Z, the longest subsequence of X and Y.

It is used to compare two strings and to measure of similarity of two strings. Important in bioinformatics.

Example: Find LCS of $X = "AzBzCzD"$ and Y="xAxxBxxC".

Solution. Z="ABC".

One way to solve it: For each subsequence of X, inspect Y for the occurrence of that subsequence. This solution has exponential time (why?).

Example. Find all subsequences of "abc".

Solution. "", a, b, c, ab, ac, bc, abc.

Example. How many subsequences does a string of n distinct characters have?

Solution. 2^n subsequences since every character can be included to or excluded from a subsequence. Note that the number of subsequences could be $< 2^n$ if some characters in the string are the same.

So, any algorithm inspecting all subsequences would be slow. We show that dynamic programming can be used to solve it in $O(mn)$ time.

Optimal Substructure

Suppose that $Z[1..k]$ (of length k) is a LCS of X and Y.

Claim 1. If $X[m] = Y[n]$ then $Z[k] = X[m]$, and $Z[1..k-1]$ is a LCS of $X[1..m-1]$ and $Y[1..n-1]$.

Proof by contradiction: Suppose $Z[k] \neq X[m]$. Then $Z[k] \neq Y[n]$ and Z is the LCS of $X[1..m-1]$ and $Y[1..n-1]$. If we append $X[m]$ to Z, the new string is the LCS of X and Y. This contradicts the assumption that Z is a LCS of X and Y.

What if $X[m] \neq Y[n]$? Then $Z[k] \neq X[m]$ or $Z[k] \neq Y[n]$ (or both). In the first case, Z is a LCS of $X[1..m-1]$ and Y. In the second case, Z is a LCS of X and $Y[1..n-1]$.

Claim 2. If $X[m] \neq Y[n]$ then
(a) Z is a LCS of $X[1..m-1]$ and Y, or
(b) Z is a LCS of X and $Y[1..n-1]$.

In order to compute a LCS of X and Y in this case, we compute both a LCS in (a) and a LCS in (b) and use the longest subsequence of (a) and (b).

Naive implementation

It follows Claim 1 and 2 above. The first call is $RecurLCS(m, n)$. Assume that the strings X and Y are external.

```
$RecurLCS(i, j)$    // Compute the length LCS of $X[1..i]$ and $Y[1..j]$
1   if $i < 1$ or $j < 1$ then
2       return 0
3   if $X[i] = Y[j]$ then    // claim 1
4       return 1+$RecurLCS(i-1, j-1)$
5   else return max($RecurLCS(i, j-1)$,$RecurLCS(i-1, j)$)    // claim 2
```

There could be recomputations according to the following example.

Example. Find the minimum and maximum number of times $RecurLCS$ can be called for $i = m-1$ and $j = n-1$?

Solution. Minimum = 0, maximum = 2.
If $X[m] = Y[n]$ then $RecurLCS(m-1, n-1)$ is called only one time.
Suppose that $X[m] \neq Y[n]$. Then $RecurLCS(m, n-1)$ and $RecurLCS(m-1, n)$ are called.
(a) If $X[m] = Y[n-1]$ and $X[m-1] = Y[n]$ then these programs will call $RecurLCS(m-1, n-2)$ and $RecurLCS(m-2, n-1)$ and $RecurLCS(m-1, n-1)$ will never be called. This can happen, for example, if $X = ..ba$ and $Y = ..ab$. This is the minimum.
(b) $RecurLCS(m-1, n-1)$ will be called from both $RecurLCS(m, n-1)$ and $RecurLCS(m-1, n)$, if $X[m] \neq Y[n-1]$ and $X[m-1] \neq Y[n]$. So, the maximum is equal to 2 if, for example $X = ..aa$ and $Y = ..bb$.

To avoid overlapping computation in the program above, we store in $L[i,j]$ the length of LCS of $X[1..i]$ and $Y[1..j]$.

```
LCS(n, m)
make an array L[0..m,0..n]
1   for $i = 1$ to $m$
2       $L[i,0] = 0$
3   for $j = 1$ to $n$
4       $L[0,j] = 0$
5   for $i = 1$ to $m$
6       for $j = 1$ to $n$
7           if $X[i] = Y[j]$ then
8               $L[i,j] = 1 + L[i-1, j-1]$
9           else $L[i,j] = \max(L[i,j-1], L[i-1,j])$
```

How to read out an LCS? Call $PrintLCS(m,n)$ after line 9 in LCS.

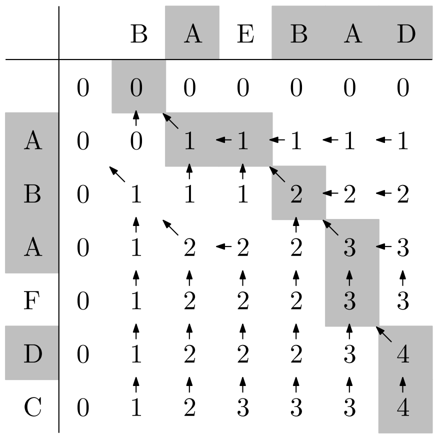
```
PrintLCS(i, j)    // print an LCS of $X[1..i]$ and $Y[1..j]$
1   if $i = 0$ or $j = 0$ return
2   if $X[i] = Y[j]$ then
3       $PrintLCS(i-1, j-1)$
4       print $X[i]$
5   elseif $L[i-1,j] \geq L[i,j-1]$
6       $PrintLCS(i-1, j)$
7   else $PrintLCS(i, j-1)$
```

The running time is $O(mn)$.

Example. Find LCS of ABAFDC and BAEBAD.

Solution. It is ABAD. The table on the left shows array $c[..]$ and the table on the right has back pointers and the path corresponding to the LCS.

		B	A	E	B	A	D
	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1
B	0	1	1	1	2	2	2
A	0	1	2	2	2	3	3
F	0	1	2	2	2	3	3
D	0	1	2	2	2	3	4
C	0	1	2	3	3	3	4



5.3 Optimal Binary Search Tree

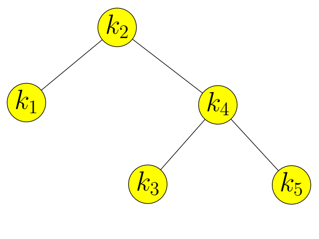
Problem. Given a sorted sequence of n keys $k_1 < k_2 < \cdots < k_n$, and probabilities p_i of searching k_i for all i, build an optimal binary search tree such that the expected search time is minimum.

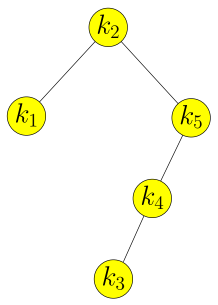
The cost of searching k_i is equal to the number of nodes on the path from the root to k_i, i.e. $cost(k_i) = depth(k_i) + 1$. Then the expected search time is

$$E[T] = \sum_{i=1}^{n} cost(k_i) \cdot p_i.$$

Example. Given the probabilities in the table, compute the expected search time of the two trees below.

i	1	2	3	4	5
p_i	0.2	0.3	0.1	0.1	0.3





Solution. The expected costs of the two trees are:
$E[T_1] = 2p_1 + p_2 + 3p_3 + 2p_4 + 3p_5 = 2.1$
$E[T_2] = 2p_1 + p_2 + 4p_3 + 3p_4 + 2p_5 = 2$

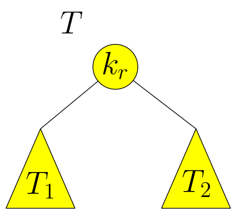
To design a dynamic program for the problem we need the following

Property: for any node k_x of T, the subtree rooted at k_x is optimal.

The proof is by contradiction. If a subtree is not optimal then we replace it by an optimal tree covering the same nodes. The expected cost of the tree will be smaller. Contradiction.

Computation. Consider the root k_r. Let T_1 and T_2 be the subtrees of the root. The depth of every node in $T_i, i = 1, 2$ is smaller by one than its depth in T. Therefore (note that p_r is added too)

$$E[T] = E[T_1] + E[T_2] + \sum_{i=1}^{n} p_i$$



This can be extended to any subtree. The keys in the subtree form a contiguous range $k_i, k_{i+1}, \ldots, k_j$. Let $c[i,j]$ be the optimal cost of such a tree. Since the root k_r of this tree is unknown, we try all values of $r \in \{i, i+1, \ldots, j\}$.

$$c[i,j] = \begin{cases} 0 & \text{if } j = i-1, \\ \min\limits_{i \le r \le j} \{c[i,r-1] + c[r+1,j] + \sum\limits_{t=i}^{j} p_t\} & \text{if } i \le j. \end{cases}$$

We store the value of r minimizing $c[i,j]$ in $root[i,j]$. Note that $j-i$ is larger than $(r-1)-i$ and $j-(r+1)$. So, we compute $c[i,j]$ values in increasing order of (the difference) $d = j - i$.

```
$BST(p, n)$
// Input: array $p$ of $n$ probabilities
// Output: array $c$ and $root$
1    for $i = 1$ to $n+1$
2      $c[i, i-1] = 0$ // empty tree
3    for $d = 0$ to $n-1$ // the difference $d$
4      sum=0
5      for $i = 1$ to $n-d$
6        $j = i + d$
7        $c[i,j] = \infty$
8        $sum += p_j$
9        for $r = i$ to $j$
10         $x = c[i, r-1] + c[r+1, j] + sum$
11         if $c[i,j] > x$ then $c[i,j] = x$ and $root[i,j] = r$
11 return $c[..]$ and $root[..]$
```

BST takes $O(n^3)$ time since there three nested loops and each runs $\le n$ times.

Exercises

5.1. *Binomial Coefficient*
A binomial coefficient $\binom{n}{k}$ can be computed as $\frac{n!}{k!(n-k)!}$ but it involves large numbers even if $\binom{n}{k}$ is not so large. Give a dynamic programming algorithm for computing $\binom{n}{k}$ using formula

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

5.2.
Show that, after $c[1], c[2], \ldots, c[n]$ are computed (using *CutRod* or *MemoizedCut*), the best cutting of a rod of length n can be printed in $O(n)$ time using only $O(1)$ additional memory. Array $p[..]$ can be used too. [2]

5.3.
Determine an LCS of (a, a, b, a, b, a, b) and (b, b, a, a, b, b, a) by running the program *LCS*. Show the table L and the path corresponding to LCS.

5.4.
Modify program *RecurLCS* using memoization to run it in $O(mn)$ time.

5.5.
Write a nonrecursive version of *PrintLCS*.

5.6.
Show how to compute the length of LCS using only $O(n)$ space in addition to X and Y. *Hint*: How many rows of the L table are needed to compute $L[i, j]$?

5.7. *Maximizing Grade*
A student takes a class with n projects and has N hours to work on the projects (maybe some projects). The grading depends on the number of hours spent for each project. Let $S_i[0..N]$ be the score function for ith project, i.e. a student spending j hours on ith project receives score $S_i[j]$. The final grade is the sum of project scores $s_1 + s_2 + \cdots + s_n$. Design a dynamic programming algorithm that computes the maximum final grade a student can receive in the class.

[2]Try this example $n = 9$ and $p = \{0, 1, 5, 8, 9, 10, 15, 17, 20, 23\}$ (so $p[1] = 1$, ...). *CutRod* computes array $c = \{0, 1, 5, 8, 10, 13, 16, 18, 21, 24\}$.

Dynamic programming problems

5.8. *Max subarray*

Let $A[0..n-1]$ be an integer array with positive numbers and negative numbers. A subarray is defined as $A[i..j]$ for any $0 \le i \le j < n$. Find a subarray with maximum sum in
(a) $O(n^2)$ time by computing the sum for each subarray,
(b) $O(n)$ time using dynamic programming.
Example: the maximum sum subarray of $A = \{1, -2, 3, 5, -4, 6, 1, -2\}$ is $\{3, 5, -4, 6, 1\}$ (the sum is 11).

5.9. *Incremental subsequence*

Let $A[0..n-1]$ be an integer array of unsorted numbers. Given an unsorted array, find the max length of subsequence in which the numbers are in incremental order.

For example: If the input array is {7, 2, 3, 1, 5, 8, 9, 6}, a subsequence with the most numbers in incremental order is {2, 3, 5, 8, 9} and the expected output is 5.

5.10. *Maximum steal*

There are n houses built in a line and the value of goods that can be stolen from ith house is c_i where $i = 1, 2, \ldots, n$. A thief wants to steal the maximal value in these houses, but he cannot steal in two adjacent houses (the owner of a robbed house will tell his two adjacent neighbors). What is the maximal stolen value if the thief does not care how many houses will be robbed?

5.11. *Job scheduling*

n jobs are given and it job is given as (s_i, f_i, p_i) where s_i is the start time, f_i is the finish time and p_i is the profit. Find a subset of nonoverlapping jobs such that their total profit is maximized.

Example. job 1: (1,2,40), job 2: (3,4,30), job 3: (5,20,100), job 4: (2,25,150).
Jobs 1,2,3 are nonoverlapping and give profit 40+30+100=170. Better solution (optimal!): jobs 1 and 4 with total profit 40+150=190.

5.12. *Polygon cutting*

Given a convex polygon P with vertices $p_0, p_1, \ldots, p_{n-1}$ in clockwise order (each point is given as $p_i = (x_i, y_i)$), cut it into triangles using diagonals such that the total length of the cuts is minimized.

5.13. *Counting binary strings*

Given an integer $n > 0$, count binary strings of length n such that there are no consecutive 1s.

5.14. *Counting decodings*

Let 1 represent A, 2 represents B, .. , 26 represents Z. Given a digit sequence S, count the number of possible decodings of S.
Example: input: $S =$ "121", output: 3 // The possible decodings are "ABA", "AU", "LA"

Solution. *Max subarray*

a) There $O(n^2)$ subarrays (why?). Computing the sum for each subarray directly takes $O(n^3)$ time in total (why? write a program). Better idea: compute the sums of subarrays $A[i..j]$ for a fixed i. There $n-i$ such subarrays and the sums can be computed in a loop with $\leq n-i$ iterations. The total time is $O(n^2)$ (fo all i).

b) One idea: compute a_i, the max sum in array $A[0..i]$ for all $i = 0, 1, \ldots, n-1$. It is hard to compute dynamically. Better idea: compute b_i, the max sum of a subarray of $A[0..i]$ that ends with $A[i]$. Then b_i can be computed dynamically: $b_i = \max(A[i], A[i] + b_{i-1})$. Then the solution is $\max(0, b_0, b_1, \ldots, b_{n-1})$ (why 0?).

Solution. *Incremental subsequence*

Idea 1: compute L_i, the longest incremental subsequence ended with $A[i]$. Brute-force: $O(n)$ time to compute one L_i (how?). [3] Show L-values if the input array is $\{7, 2, 3, 1, 5, 8, 9, 6\}$. What is the subsequence with the most numbers in incremental order?

Better idea: compute each L_i in $O(\log n)$ time. A new array $M[1..t]$ can help where t is the max length of an incremental subsequence found after processing $A[i]$. We store in $M[1..t]$ the following values: $M[j] = k$ if $A[k]$ is the smallest element in $A[0..i]$ such that there is an incremental subsequence of length j ending at $A[k]$. Show how t and array $M[..]$ are updated for every i if the input array is $\{7, 2, 3, 1, 5, 8, 9, 6\}$. How many elements of $M[..]$ may change when $A[i]$ is processed? How fast can it be done? *Hint: Show that $M[..]$ is incremental sequence.*

Show how to compute the longest incremental subsequence (not just its length) in $O(n \log n)$ time. *Hint: Use an additional array (in addition to $L[..], M[..]$)* [4]

Solution. *Maximum steal*

Let s_i be the max stolen value in the first i houses (the other houses are not robbed). If ith house is not robbed then $s_i = s_{i-1}$, otherwise $s_i = c_i + s_{i-2}$ (why?). Array s can be computed in $O(n)$ time and s_n is the answer.

Solution. *Job scheduling*

We sort jobs by finish time, so $f_0 \leq f_1 \leq \cdots \leq f_{n-1}$.
Subproblems: If job $n-1$ is not in an optimal solution then we solve the problem for the first $n-1$ jobs. Otherwise find jobs with $f_i \leq s_{n-1}$, solve the same problem for them and add it to p_{n-1}.

```
$MaxP(k)$ // find max profit for jobs $0, 1, \ldots, k$
if $j - i <= 2$ return 0// nothing to cut
$max1 = MaxP(k-1)$ // case 1
for $i = k-1$ downto 0
    if $f_i \leq s_k$ // case 2
        $max2 = MaxP(i)$
        return max($max1, max2$)
return $max1$
```

[3] Use formula $L_i = 1 + \max\{L_j\}$ over all $j < i$ such that $A[j] < A[i]$. If such an L_j doesn't exist, then $L_i = 1$.
[4] For each i compute $P[i]$, the index of the predecessor of $A[i]$ in the longest incremental subsequence ending at $A[i]$.

Questions. 1) Are there recalcutalions? 2) Modify the program to avoid recalcutalions.

Solution. *Polygon cutting*

Input: array p storing vertices $p_0, p_1, \dots, p_{n-1}$ in clockwise order where each point is given as $p_i = (x_i, y_i)$. We find subproblems as follows. Consider an optimal cutting. Then $p_0 p_{n-1}$ is the side of a triangle, say $p_0 p_{n-1} p_k$. The polygons $P_1 = \{p_0, p_1, \dots, p_k\}$ and $P_2 = \{p_{k+1}, p_{k+2}, \dots, p_{n-1}\}$ (one polygon can be empty!) are cut optimally. In general we solve problems of cutting polygons $\{p_i, p_{i+1}, \dots, p_j\}$. The same idea can be used to find a triangle with side $p_i p_j$.

```
$Cut(i,j)$ // find an optimal cutting of polygon $\{p_i, p_{i+1}, \dots, p_j\}$
if $j - i <= 2$ return 0// nothing to cut
$mincut = \infty$
for $k = i+1$ to $j-1$
    $cost = Cut(i,k) + Cut(k,j) + dist(p_i, p_k) + dist(p_j, p_k)$
    if $mincut > cost$
        $mincut = cost$
return $mincut$
```

Questions. 1) Are there recalcutalions? 2) Modify the program to avoid recalcutalions.

Solution. *Counting binary strings*

Let a_i be the number of binary strings of length i that have no consecutive 1's and end with 0. Let b_i be the number of binary strings of length i that have no consecutive 1's and end with 1. Then $a_i = a_{i-1} + b_{i-1}$ and $b_i = a_{i-1}$. Initial values are $a_1 = b_1 = 1$.

Solution. *Counting decodings*

We assume that S is the valid encoding of a string. Let $d_i, i = 1, 2, \dots, n$ be n input digits. Let v_i be the number of possible decodings of the string first i digits of the input. Then

```
$v_1 = 1$
for $i = 2$ to $n$
    $v_i = 0$ // initial count
    if $d_i > 0$
        $v_i += v_{i-1}$ // last digit could be a character
    if $d_{i-1} = 1$ or ($d_{i-1} = 2$ and $d_i < 7$)
        $v_i += v_{i-2}$ // last 2 digits could be a character
```

Questions. 1) Write a program testing whether S is valid. 2) Modify the program such that it uses only d and $O(1)$ space. Your program should run in $O(n)$ time as well.

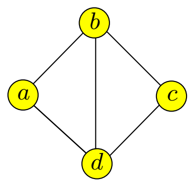
Chapter 6

Minimum Spanning Trees

6.1 Minimum Spanning Tree Problem

A graph $G = (V, E)$ is connected if, for any two vertices $u, v \in V$, there is a path from u to v in G. A graph is a tree if it is connected and has no cycles. Every tree with n vertices has exactly $n - 1$ edges. A tree $T = (V_T, E_T)$ is a spanning tree of a graph $G = (V, E)$ if $V_T = V$ (tree spans G) and $E_T \subseteq E$.

Example. Find all spanning trees of the following graph.

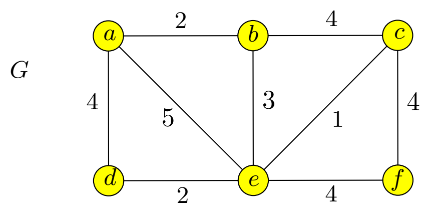


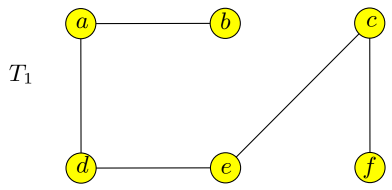
Solution. We specify each tree by a list of its edges. $T_1 = \{ab, ad, cd\}, T_2 = \{ad, bc, cd\}, T_3 = \{ab, ad, bc\}, T_4 = \{ab, bc, cd\}, T_5 = \{bd, ab, bc\}, T_6 = \{bd, ab, cd\}, T_7 = \{bd, ad, bc\}, T_8 = \{bd, ad, cd\}$.

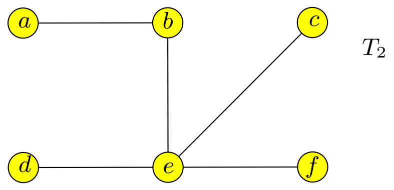
Example. Show that every connected graph has a spanning tree. Show how a spanning tree can be computed.

Let $G = (V, E)$ be a a connected undirected graph $G = (V, E)$ such that every edge $(u, v) \in E$ is assigned a weight $w(u, v) \in \mathbb{R}$.
The weight of a spanning tree T of G is defined as the sum the weights of its edges.
A spanning tree that has minimum weight is called minimum spanning tree (MST).
MST problem is to compute a minimum spanning tree of G.

Example. Find the weights of two spanning trees T_1 and T_2 of a graph $G = (V, E)$.







Solution. $w(T_1) = 2 + 4 + 2 + 1 + 4 = 13$. $w(T_2) = 2 + 3 + 2 + 1 + 4 = 12$. Therefore T_1 is not a minimum spanning tree.

General greedy approach

Maintain a set of edges F such that
1) (V, F) is a *spanning forest* of G, and
2) there is a MST T such that $F \subseteq T$.

```
GreedyMST(G, w)
1   F = ∅
2   while |F| < n − 1
3      find an edge e ∈ E − F // greedy choice
4      F = F ∪ e
5   return F
```

3 algorithms follows this approach: Prim's algorithm, Kruskal's algorithm and Borůvka's algorithm.

Borůvka's algorithm discovered in 1926 (before computers!). Rediscovered many times.

Jarník developed an MST algorithm in 1929. Rediscovered by Prim in 1957. Called Prim-Jarník algorithm.

Kriskal found his algorithm in 1956.

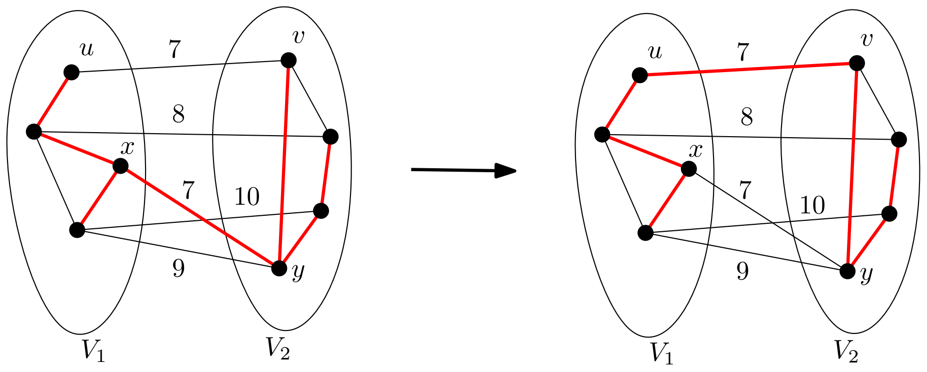
Cut theorem

A cut (V_1, V_2) of G is a partition of V into two sets V_1 and $V_2 = V - V_1$.
An edge e crosses the cut (V_1, V_2) if one of the endpoints of e is in V_1 and the other is in V_2.
A cut respects a set $F \subseteq E$ if no edge of F crosses the cut.

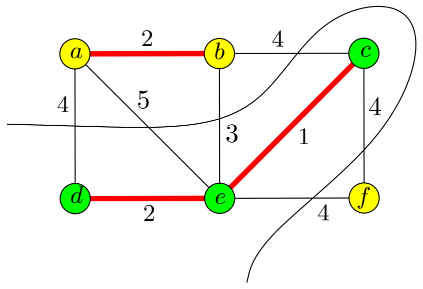
The following theorem allows us to add a new edge to compute a MST.

Theorem. Let F be a subset of some MST of G and let (V_1, V_2) be a cut respecting F. Let e be a minimum weight edge crossing the cut. Then $F \cup \{e\}$ a subset of some MST of G.

Proof. See Figure below where $e = (u, v)$. Let T (red edges) be a MST that includes F (not shown in the figure). If $(u, v) \in T$ then we are done. Suppose that $(u, v) \notin T$. There is a path in T from u to v. There is a cross edge (x, y) in the path. Thus $(x, y) \notin F$. We replace (x, y) in T by (u, v) and obtain a tree T' of weight no larger than T since $w(u, v) \leq w(x, y)$. Then T' is a MST containing the edges of $F \cup \{e\}$.



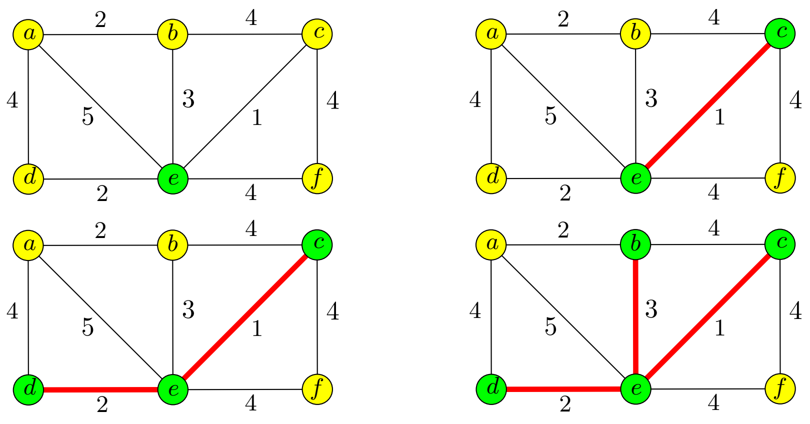
Example. In the following figure, set F consists of red edges and the cut is $(\{a, b, f\}, \{c, d, e\})$. Find all edges that can be added to set F using the cut theorem.

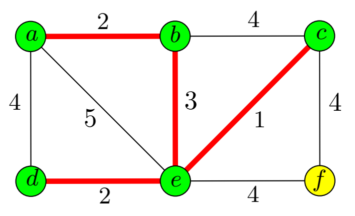


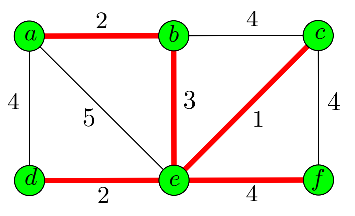
Solution. Only one edge (b, e).

6.2 Prim's algorithm

Grow a single tree T starting with any node called root of MST. Greedy choice: select a minimum weight edge that connects a vertex in T and a vertex not in T.







Prim's algorithm starting at vertex e. In the last step, there a choice of adding ef or cf to T.

Implementation of Prim's algorithm

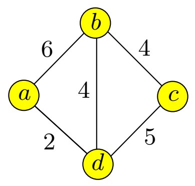
We store the graph using the adjacency lists, i.e. for each vertex, we store the list of vertices adjacent to it.

Main question is how to find new edge efficiently. Use a priority queue Q to store non-tree vertices with a key. The key $v.key$ is the minimum weight of an edge (v, u) connecting v to a tree vertex u. Store u in $v.pred$.

Priority queue operations:
Insert (v) - insert a vertex v with priority $v.key$,
RemoveMin - remove the vertex from Q with min key,
DecreaseKey (v, t) - decrease the key of v to t.

```
PrimMST $(G, w, s)$
// Input: Graph $G = (V, E)$, weight function $w$ and a start vertex $s \in V$.
// Output: MST with edges $(v, pred[v]), v \in V - \{r\}$.
1   make an empty $Q$ and an empty $T$
2   insert all $v \in V - \{s\}$ into $Q$ with $v.key = \infty$
3   for each neighbor $v$ of $s$
4       DecreaseKey $(v, w(v, s))$
5   while $Q \neq \emptyset$    // main loop
6       $v =$ RemoveMin ()
7       add $(v, v.prev)$ to $T$
8       for each neighbor $u$ of $v$
9           if $u \in Q$ and $w(u, v) < u.key$
10              $u.pred = v$
11              DecreaseKey $(u, w(u, v))$
```

Example. Run *PrimMST* on the following graph starting with a.



Solution. We show the steps using the following table.

Q: vertex	key	pred	vertex v	new MST edge
b	6	a	d	(d, a)
c	∞	NIL		
d	2	a		
b	4	d	b	(b, d)
c	5	d		
c	4	b	c	(b, c)

Running time

If we use a binary heap for the priority queue, then each operation *Insert*, *RemoveMin*, and *DecreaseKey* takes $O(\lg V)$ time. Then lines 1-4 take $O(V \lg V)$ time. Line 6 takes $O(V \lg V)$ time in total.

Every edge is tested at most 2 times in line 9 (every edge has 2 endpoints). Lines 9-11 take $O(E \lg V)$ time in total. Total time of the algorithm is $O(V \lg V + E \lg V)$. This is $O(E \lg V)$ since $E \geq V - 1$ for connected graph.

demo

Running time using Fibonacci Heap

Facts about Fibonacci heap:

- *Insert*: amortized cost is $O(1)$.
- *RemoveMin*: amortized cost is $O(\lg V)$.
- *DecreaseKey*: amortized cost is $O(1)$.

Total time is $O(E + V \lg V)$ if Fibonacci heap is used for the priority queue.

6.3 Kruskal's Algorithm

Grow a forest with set of edges F. Initially $F = \emptyset$ and each vertex of G is a separate tree.

Greedy choice: try to add minimum weight edge. Add the edge only if it connects different trees.

demo

Implementation of Kruskal's Algorithm

We need to maintain connected components. Disjoint-set data structure.

Two operations:

- *findSet* (u) returns a representative element from the set that contains u.
- *union* (u, v) joins two sets containing u and v.

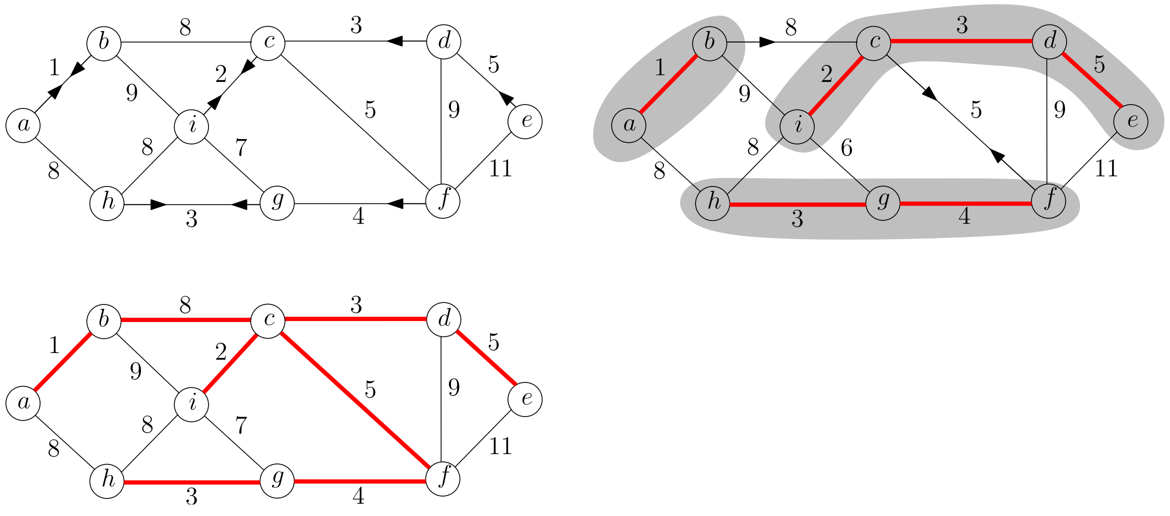
```
MST-Kruskal $(G, w)$
1  sort the edges of $E$ into non-decreasing order by weight $w$
2  $F = \emptyset$
3  for each vertex $v \in V$
4      makeSet $(v)$ // make a set for every vertex
5  for each edge $(u, v) \in E$ (use the sorted order)
6      if findSet $(u) \neq$ findSet $(v)$ // $u$ and $v$ are in different sets
7          union $(u, v)$ and add $(u, v)$ to $F$
8  return $F$
```

Running time. Line 1 takes $O(E \lg V)$ time. Lines 2-4 take $O(V)$ time. Line 7 takes $O(\lg V)$ time. Total time is $O(E \lg V)$.

6.4 Borůvka's Algorithm

Grow many trees like in Kruskal's algorithm. Unlike Prim's and Kruskal's algorithm, which add edges one at the time, Borůvka's algorithm adds many edges in one step. We assume that the edge weights are distinct (otherwise number the edges and break ties using min edge number).

```
MST-Borůvka $(G, w)$
1  $A = \emptyset$
2  while $|A| < n - 1$
3      for each component $C$
4          find the min-weight edge $(u, v)$ with $u \in C, v \notin C$
5          add $(u, v)$ to $A$ (unless it is already there)
6  find new components
7  return $A$
```



Borůvka's algorithm. At the beginning every vertex is a single tree. Three trees are made after the first iteration. MST is computed after the second iteration.

Exercises

6.1.
Let G be a connected, undirected and weighted graph. Let (a, b) be a smallest weight edge of G. Show that there is a MST of G such that it contains (a, b).

6.2.
Show that if G has at least two different minimum spanning trees, then some edges in G have equal weight. *Hint: Take an edge (a, b) of T_1 that is not present in T_2. Consider the path in T_2 connecting a and b and its heaviest edge e.*

6.3.
Prove or disprove that if G has a unique MST then the edge weights in G are distinct.

6.4.
Show how to modify Prim's algorithm to run in $O(V^2)$ time if $G = (V, E)$ is represented as an adjacency matrix.

6.5.
Suppose that the edges of $G = (V, E)$ have weights in $\{1, 2\}$. Modify Kruskal algorithm such that it runs in $O(E)$ time.

6.6.
Show that Borůvka's algorithm has $O(\lg V)$ iterations.

6.7.
Show that Prim's algorithm is correct using the Cut Theorem.

6.8.

Suppose that a program `FastTree` computes an MST of a graph with positive weights in $O(E)$ time. Show that it can be used to compute an MST of *any* graph (some edges may have negative weights) in $O(E)$ time.

Chapter 7

Shortest Paths

7.1 Shortest Paths

Let $G = (V, E)$ be a weighted, directed graph. Every edge (v_i, v_j) has a cost or weight $w(v_i, v_j)$. The weight of a path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is defined as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

A path p from u to v with minimum weight $w(p)$ is called a shortest path. Notation: $\text{depth}(u, v)$ is the cost of the shortest path from u to v. If there is no path from u to v, then $\text{depth}(u, v) = \infty$.

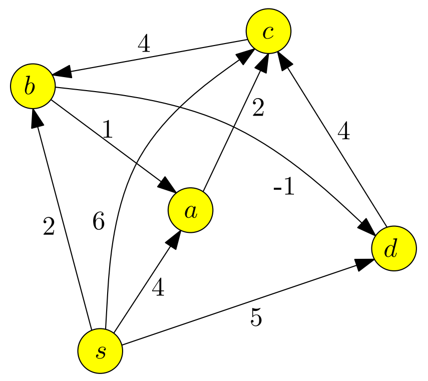
Single-source shortest-path problem. Given a weighted, directed graph $G = (V, E)$ and a source vertex s, find the shortest weighted path from s to every other vertex in G.

Variations

The shortest path problem can also be defined for

- *unweighted graphs.* Assign the cost 1 for every edge.
- *undirected graphs.* Every edge can be used in both directions.

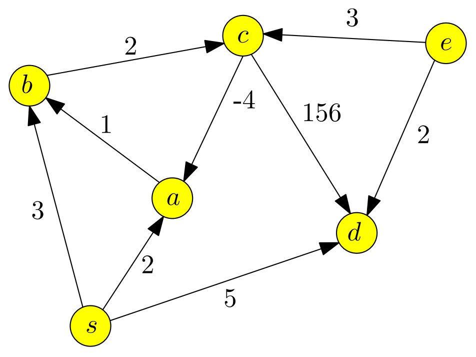
Example. Find a shortest path from s to c and its length (note that negative edge weights are allowed).



Solution. There are two shortest paths s, b, a, c and s, b, d, c, each of length 5.

Negative edge weights

Example. Find shortest paths from s to d and e. Find $\delta(s, d)$ and $\delta(s, e)$.



Solution. There is no shortest path from s to d. Actually, for any cost $c \in \mathbb{R}$ there is a path from s to d of weight smaller that c (by traversing the negative-weight cycle a, b, c many times). So $\delta(s, d) = -\infty$.
There is no path in G from s to e. Vertex e is not reachable from s and so $\delta(s, e) = \infty$.

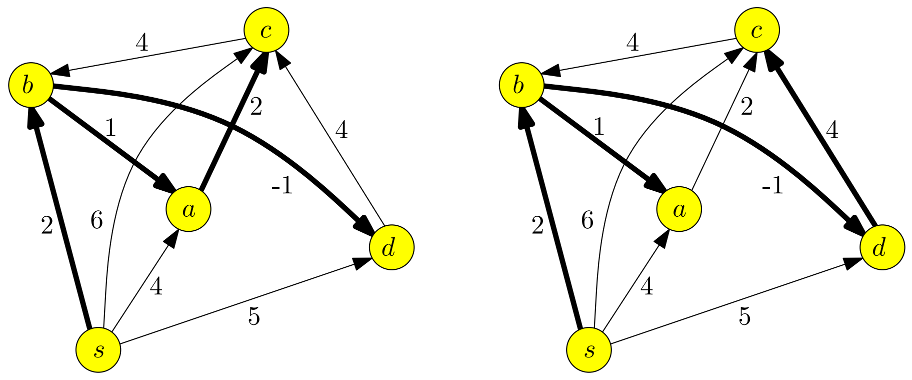
Optimal substructure

Shortest paths have a nice property that their subpaths are shortest paths as well (between their endpoints).

Lemma. Let $v_1, v_2, \ldots, v_k$ be a shortest path from vertex v_1 to vertex v_k in a graph G. Then $v_i, v_{i+1}, \ldots, v_j$ is the shortest path from vertex v_i to vertex v_j in graph G, for any i and j such that $1 \leq i \leq j \leq k$.

Proof: Let $p = v_1, v_2, \ldots, v_k$. Suppose to the contrary that the path $p_{i,j} = v_i, v_{i+1}, \ldots, v_j$ is not shortest and there is a shorter path $p'_{i,j}$ from v_i to v_j. Replace $p_{i,j}$ by $p'_{i,j}$ in p. The new path is shorter than p. It contradicts the fact that p is the shortest path from v_0 to v_k. ■

This property suggests a simple way of storing shortest paths: for each vertex $v \in V$, store its predecessor in a shortest path. Shortest paths from s to the vertices reachable from s form a shortest-paths tree rooted at s. The shortest-paths tree maybe not unique, see two shortest-paths trees for the same graph in the next figure.



Idea: store a tentative distance and tentative predecessor of a vertex v at $v.dist$ and $v.prev$ (previous vertex). Initialization:

```
Initialization (G, s)
1   for each vertex $v \in V$
2       $v.dist = \infty$
3       $v.prev$ =nil
4   $s.dist = 0$
```

Relaxation

The basic tool to improve the tentative distances is relaxation of an edge (u, v): try to use the path $s \rightsquigarrow u \to v$ of cost $u.dist + w(u, v)$.

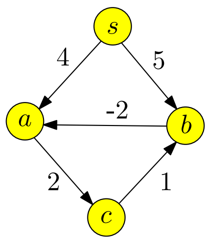
```
Relax (u, v)
// relax edge $(u, v)$ using weight $w(u, v)$
1   if $v.dist > u.dist + w(u, v)$ then
2       $v.dist = u.dist + w(u, v)$
3       $v.prev = u$
```

7.2 Bellman-Ford algorithm

```
Bellman-Ford $(G, w, s)$
1  Initialization $(G, s)$
2  for $i = 1$ to $n - 1$
3      for each edge $(u, v) \in E$
4          Relax $(u, v)$
5  for every edge $(u, v)$
6      if $v.dist > u.dist + w(u, v)$
7          return false
8  return true
```

The algorithm returns false if G contains a negative-weight cycle reachable from s. Otherwise it returns true and all distances are correct.

Example. Run Bellman-Ford algorithm on the following graph. Use the lexicographic order of the edges.



Solution. The order of the edges: ac, ba, cb, sa, sb. There are 3 iterations of the first loop (line 2).

Initialization

	s	a	b	c
dist	0	∞	∞	∞
prev	nil	nil	nil	nil

Iteration 1

	s	a	b	c
dist	0	4	5	∞
prev	nil	s	s	nil

Iteration 2

	s	a	b	c
dist	0	3	5	6
prev	nil	b	s	a

Iteration 3

	s	a	b	c
dist	0	3	5	5
prev	nil	b	s	a

The algorithm returns true since the condition in line 6 doesn't hold for any edge of G .

Example. Suppose that G contains no negative-weight cycles that are reachable from s. Show that the Bellman-Ford algorithm returns true and the computed distances are correct.

Solution. We show that after the loop at line 2 finished $v.dist = \delta(s, v)$ for any v. If v is not reachable then $\delta(s, v) = \infty$. If it is reachable and $v_1 = s, v_2, \ldots, v_k = v$ is a shortest path, then $k \leq n$ (otherwise there is a cycle in the path). We show that at the beginning of i-th iteration $v_i.dist = \delta(s, v_i)$. It is true for $i = 0$ (why?). Iteration i relaxes edge (v_i, v_{i+1}) and sets $v_{i+1}.dist = \delta(s, v_{i+1})$ (since $v_1, \ldots, v_{i+1}$ is the shortest path from v_1 to v_{i+1}). So, the claim holds for the next iteration. Then $v.dist = \delta(s, v)$ after the last iteration.
The loop at line 5 does not return false since $v.dist$ has the smallest possible value. So, the output is true.

Example. Show that, if G contains a negative-weight cycle C reachable from s, the algorithm returns false.

Solution. Since C is reachable from s, $v.dist \neq \infty$ for all $v \in C$ after the loop at line 2 is finished. We show that at least one edge of C satisfy the condition in line 6, no matter what values of $v.dist, v \in C$ are assigned. The condition can be written as $v_{i-1}.dist + w(v_{i-1}, v_i) - v_i.dist < 0$. It suffices to prove that the sum of $v_{i-1}.dist + w(v_{i-1}, v_i) - v_i.dist$ is negative. Indeed,

$$\sum_{(v_{i-1},v_i)\in C} (v_{i-1}.dist + w(v_{i-1}, v_i) - v_i.dist) = \sum_{(v_{i-1},v_i)\in C} w(v_{i-1}, v_i) < 0.$$

Bellman-Ford algorithm runs in $O(VE)$ time.

7.3 Dijkstra algorithm

We assume that all the edge weights in G are non-negative.

```
Dijkstra $(G, w, s)$
1  Initialization $(G, s)$
2  $S = \emptyset$
3  for $i = 1$ to $n$
4     find $v \in V - S$ with minimum $v.dist$
5     $S = S \cup \{v\}$
6     for each vertex $u \in v.Adj$    // $u$ is a neighbor of $v$
7        Relax $(v, u)$
```

Implementation and running time

Use priority queue to store vertices from $V - S$ with keys $v.dist$. Then *RemoveMin* is executed $|V|$ times (line 4) and *DecreaseKey* is executed $|E|$ times (line 7). Running time depends on implementation of the priority queue (time for *RemoveMin* and time for*DecreaseKey*):

priority queue	*RemoveMin*	*DecreaseKey*	Total time
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O((V + E)\lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$ (amortized)	$O(V \lg V + E)$

Correctness of Dijkstra Algorithm

We prove that $v.dist = \delta(s, v)$ for all $v \in S$ at any time. Consider new vertex v added to S in line 5. We prove that $v.dist = \delta(s, v)$. It is true if $v = s$. Suppose that $v \neq s$. Then $s \in S$ and $v \notin S$ (at the time of execution of line 4). Let π be the shortest path from s to v. Then $\pi = s, \ldots, x, y, \ldots, v$ where y is the first vertex not in S. Note that x may be equal to s and y may be equal to v.

Since $x \in S$ and edge xy is relaxed when x is added to S, we have $y.dist = \delta(s,y)$. By the optimal substructure property, $\delta(s,y) \le \delta(s,v)$. Then $\delta(s,v) \le v.dist$ (tentative distance). By the choice of v, $v.dist \le y.dist$. So, $y.dist = \delta(s,y) \le \delta(s,v) \le v.dist \le y.dist$. Then the inequalities must be equalities and $v.dist = \delta(s,v)$.

Shortest Paths in DAGs

Can we speed up Dijkstra algorithm for special graphs?

DAG, directed acyclic graph is a directed graph without cycles (cycle with negative weight may be a problem).

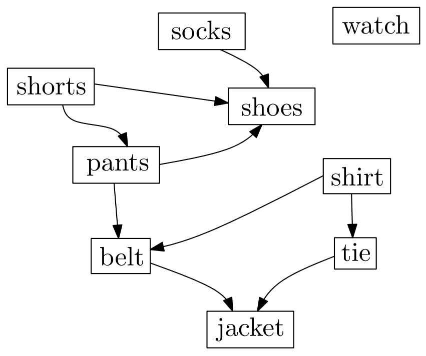
We show that the shortest path problem can be solved faster for DAGs using topological sort.

7.4 Topological sort

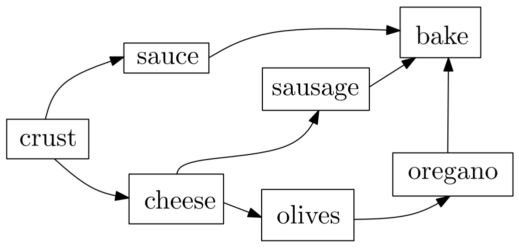
A topological sort or topological ordering of a directed acyclic graph is a linear ordering of its vertices such that u comes before v in the ordering for any edge (u,v) in the graph.

For example, suppose we want to perform a set of tasks but some tasks have to be done before other tasks. In what order should we perform the tasks? This problem can be solved by representing the tasks as node in a graph, where there is an edge from task A to task B if task A must be done before task B. Then a topological sort of the graph will give an ordering to perform the tasks. The examples below show a task graph for getting dressed and a task graph for cooking a pizza. Possible topological orders for these graphs:

- socks, undershorts,pants,shoes,watch,shirt,belt,tie,jacket
- crust,sauce,cheese,sausage,olives,oregano,bake.



Dress graph

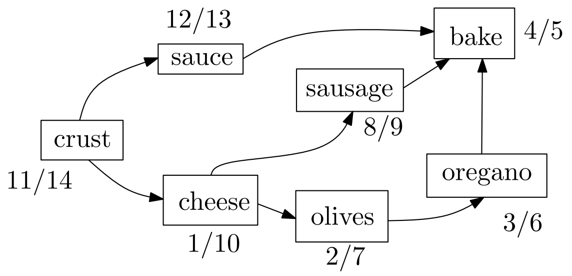


Pizza graph

Topological sort using DFS

Topological ordering of a graph G can be computed using depth-first search (DFS) as follows. Run DFS of G and arrange the vertices in order of decreasing finish time. For example, a run of DFS on the pizza graph is shown below. The numbers next to each vertex indicate its discovery and finish time. The decreasing order of the finis imes is

crust (14), sauce (13), cheese (10), sausage (9), olives (7), oregano (6), bake (5)
which is the topological sort (why?).



A linked list can be used to maintain the order of vertices finished so far by DFS.

```
TopologicalSort (G)
1   make an empty list L
2   call DFS with the modification:
        when v is finished, add it as the head of list L
3   return T
```

The running time $O(V+E)$ since DFS takes time $O(V+E)$. Insertion of one vertex takes $O(1)$ time, so the total time for the insertions is $O(V)$.

Not every directed graph has a topological ordering. What graphs have a topological ordering? Is there an algorithm to test whether a graph has a topological ordering?

Example. Suppose that a directed graph G has a topological order. Prove that G is acyclic (has no cycles).

Solution. Proof by contradiction. Suppose to the contrary that G has a cycle, say $v_1 v_2 \dots v_k$. Consider a topological ordering of G. Let v_i be the first vertex of the cycle in the order. Consider edge (v_{i-1}, v_i). Vertex v_{i-1} comes after v_i in the topological ordering. Contradiction.

Example. Let G be a directed acyclic graph. Prove that a DFS of G yields no back edges.

Solution. Suppose to the contrary that an edge (u, v) is labeled "back edge" by a DFS of G. There is a path from v to u consisting of tree edges. They form a cycle together with edge (u, v). Contradiction.

Example. Suppose that a DFS of G yields no back edges. Prove that G has a topological order.

Solution. We run the above algorithm *Topological-Sort*. The DFS based algorithm finds an order of the vertices. Consider an edge (u, v). It is either a tree edge or a forward edge or a cross edge. In each case the finishing time of v is less than the the finishing time of u. Therefore the ordering computed by *Topological-Sort* is topological.

These exercises imply the following theorem.

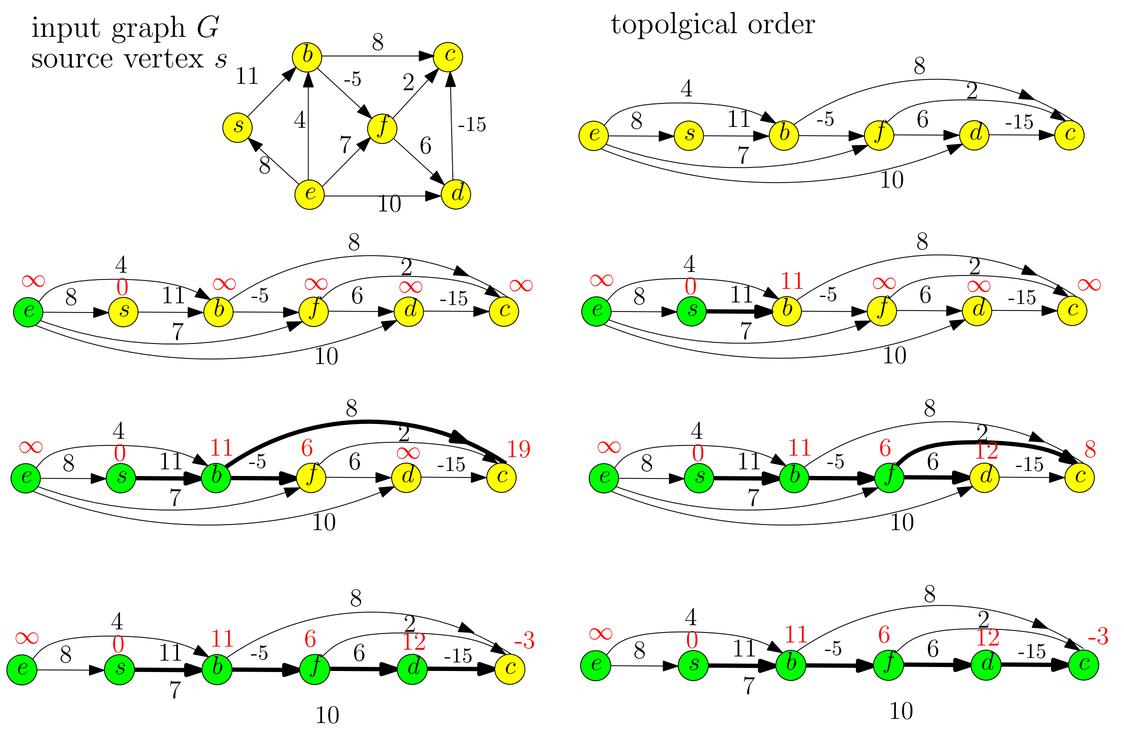
Theorem. Let G be a directed graph. The following conditions are equivalent:

- G has a topological ordering.
- G is acyclic (i.e. G has no cycles).
- A DFS of G yields no back edges.

7.5 Shortest paths in DAGs

```
DAG-Paths $(G, w, s)$
1    $L$=TopologicalSort $(G)$
2    Initialization $(G, s)$
3    for each vertex $v$ in $L$
4       for each vertex $u \in v.Adj$    // $u$ is a neighbor of $v$
5          Relax$(v, u)$
```

Example.



Analysis

Theorem. *DAG-Paths* is correct and computes shortest-paths tree in $O(V + E)$ time.

Proof. We show that the distances computed by *DAG-Paths* are correct, i.e. $v.dist = \delta(s, v)$ for all $v \in V$. Clearly $v.dist = \delta(s, v) = \infty$ if v is not reachable from s.

Suppose that v is reachable from s and consider a shortest path π from s to v. The vertices of π are in topological order and the edges of π are relaxed in the order of π. Then $v.dist = \delta(s, v)$.

Time for topological sort is $O(V + E)$, time for line 2 is $O(V)$, time for line 5 is $O(1)$ each time and $O(E)$ in total. ■

7.6 Difference constraints and shortest paths

Consider a special case of linear programming where constraints are

$$x_j - x_i \le b_k$$

for $1 \le k \le m$ and some pairs $1 \le i, j \le n$. They are called difference constraints.

Example.

$$\begin{cases} 1 \le x_2 - x_1 \le 2 \\ \quad x_1 - x_3 \le -1 \\ \quad x_3 - x_2 \le 0 \end{cases} \quad \text{or} \quad \begin{cases} x_2 - x_1 \le 2 \\ x_1 - x_2 \le -1 \\ x_1 - x_3 \le -1 \\ x_3 - x_2 \le 0 \end{cases}$$

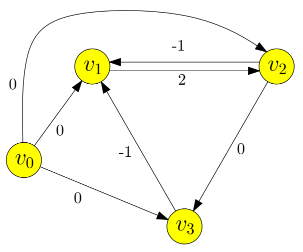
Is it feasible?

Constraint graph

Vertices and edges:

- vertex v_i for each unknown $x_i, 1 \le i \le n$
- vertex v_0
- edge (v_i, v_j) of weight b_k for each constraint $x_j - x_i \le b_k$
- edge (v_0, v_i) of weight 0 for each $1 \le i \le n$

$$\begin{cases} x_2 - x_1 \le 2 \\ x_1 - x_2 \le -1 \\ x_1 - x_3 \le -1 \\ x_3 - x_2 \le 0 \end{cases} \quad \longrightarrow$$



Theorem

1. If G contains a negative-weight cycle, then there is no feasible solution for the system.
2. If G contains no negative-weight cycles, then

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$$

   is a feasible solution for the system.

Proof.

1. Sum constraints of the cycle. $0 \le w(C) < 0$. No solution.
2. Shortest distances $\delta(v_0, v_i)$ can be computed using Bellman-Ford algorithm. By the triangle inequality

$$\delta(v_0, v_j) \le \delta(v_0, v_i) + b_k.$$

   Thus $x_j - x_i \le b_k$.

7.7 All-Pairs Shortest Paths

Assumptions

- weights can be negative
- no negative-weight cycle
- vertices are numbered from 1 to n
- the weights are given in an $n \times n$ matrix $W = (w_{ij})$ where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of edge } (i,j) & \text{if } i \ne j \text{ and } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$$

Compute $\delta(i,j)$, shortest-path distance from i to j.
Bellman-Ford uses relaxation $d_j = \min(d_j, d_k + w_{kj})$.
Let $l_{ij}^{(m)}$ be the minimum weight of any path from i to j using $\le m$ edges.
When $m = 0$, $l_{ii}^{(0)} = 0$ and $l_{ij}^{(0)} = \infty$ for $i \ne j$.

$$\begin{aligned} l_{ij}^{(m)} &= \min(l_{ij}^{(m-1)}, \min_{k \ne j}\{l_{ik}^{(m-1)} + w_{kj}\}) \\ &= \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{kj}\}, \quad \text{since } w_{jj} = 0. \end{aligned}$$

Then $\delta(i,j) = l_{ij}^{(n-1)}$. Time for one m is $O(n^3)$. Total time $O(n^4)$.

Floyd-Warshall algorithm

Let $d_{ij}^{(k)}$ be the minimum weight of any path from i to j with intermediate vertices from $1, 2, \ldots, k$.

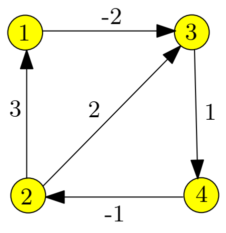
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } \ k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } \ k \geq 1 \end{cases}$$

Note that d-values can be computed using the same array, say D, since $d_{ik}^{(k)} = d_{ik}^{(k-1)}$, for all i, (why?). Also $d_{kj}^{(k)} = d_{kj}^{(k-1)}$, for all j, (why?).

```
FLOYD-WARSHALL $(W, n)$
// Input: $n \times n$ matrix $W$
// Output: matrix $D$ of shortest-path weights
1   $D = W$
2   for $k = 1$ to $n$
3       for $i = 1$ to $n$
4           for $j = 1$ to $n$
5               if $D[i][j] > D[i][k] + D[k][j]$
6                   $D[i][j] = D[i][k] + D[k][j]$
7   return $D$
```

Running time $\Theta(n^3)$.

Example. Use the Floyd-Warshall algorithm to find all shortest distances in the following graph.



Solution.

initilly	1	2	3	4
1	0	∞	-2	∞
2	3	0	2	∞
3	∞	∞	0	1
4	∞	-1	∞	0

$k=1$	1	2	3	4
1	0	∞	-2	∞
2	3	0	1	∞
3	∞	∞	0	1
4	∞	-1	∞	0

$k=2$	1	2	3	4
1	0	∞	-2	∞
2	3	0	1	∞
3	∞	∞	0	1
4	2	-1	0	0

$k=3$	1	2	3	4
1	0	∞	-2	-1
2	3	0	1	2
3	∞	∞	0	1
4	2	-1	0	0

$k=4$	1	2	3	4
1	0	-2	-2	-1
2	3	0	1	2
3	3	0	0	1
4	2	-1	0	0

Exercises

7.1. *Shortest paths in Floyd-Warshall algorithm*

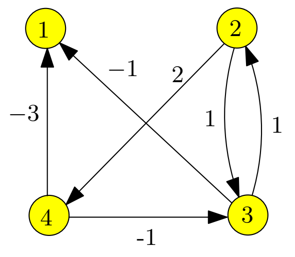
Modify the Floyd-Warshall algorithm to compute an array $S[n][n]$ such that $S[i][j]$ is the second vertex in the shortest path from i to j. Show that, for given pair i, j, the shortest path π from i to j can be printed in $O(|\pi|)$ time using array S.

7.2. *Bad cycles in Floyd-Warshall algorithm*

Modify the Floyd-Warshall algorithm to detect whether graph G has a cycle of weight $w < 0$. Explain the correctness of your algorithm.

7.3.

Run the Floyd-Warshall algorithm on the following graph and show matrices D after each iteration.



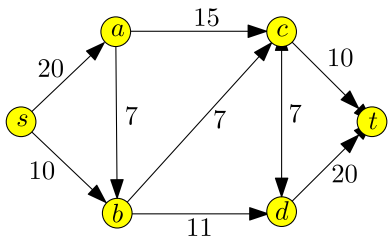
Chapter 8

Maximum Flow

8.1 Maximum-flow problem

Input: A a directed, weighted graph $G = (V, E)$ such that

- a capacity $c(u, v) \geq 0$ is assigned to every edge of G,
- if $(a, b) \in E$ then $(b, a) \notin E$ (explained later why),
- two special vertices s and t called source and sink.



A flow in G is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies:

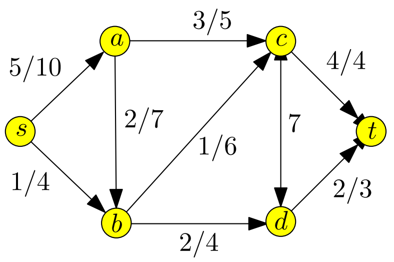
- Capacity constraint. $f(u, v) \leq c(u, v)$ for all $(u, v) \in E$.
- Flow conservation. $\forall v \in V - \{s, t\}$, $\sum_u f(u, v) = \sum_u f(v, u)$.

For simplicity, assume $f(u, v) = 0$ if $(u, v) \notin E$.
The value of a flow f is defined as $|f| = \sum_v f(s, v) - \sum_v f(v, s)$.

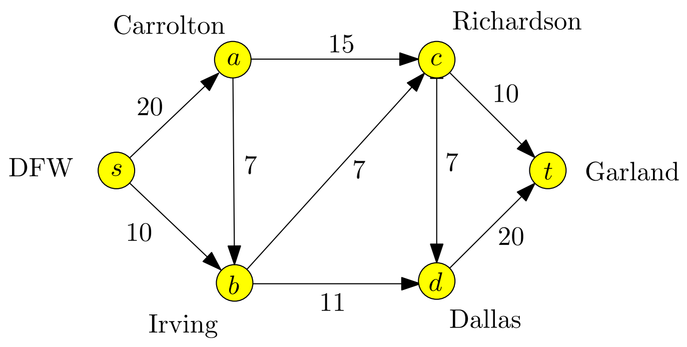
Maximum-flow problem. Given G, s, and t, find a flow of maximum value.

Example. Check the flow properties and find the flow value in the following graph.

Labels:
$f(u, v)/c(u, v)$ if $f(u, v) > 0$, and
$c(u, v)$ if $f(u, v) = 0$.



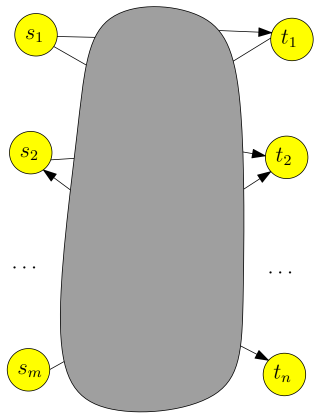
Newspaper delivery



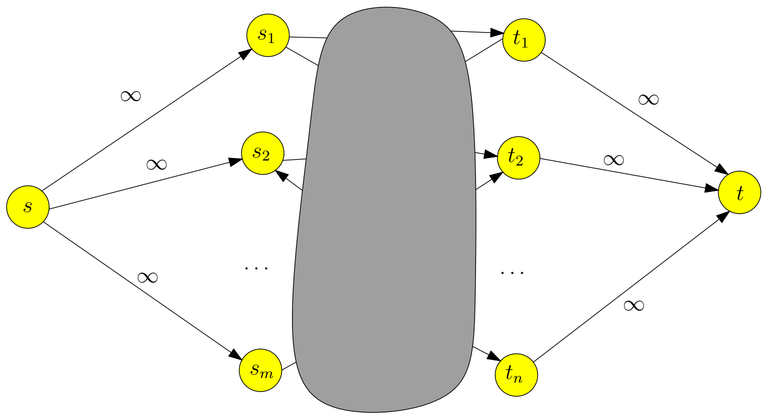
DGN (Dallas Good News) company has a publishing factory near DFW airport and a warehouse in Garland. DGN leases space on trucks from FF (Fast-and-Furious) truck company to ship the newspapers from the factory to the warehouse. FF can ship $c(u, v)$ newspapers per day from u to v. How many newspapers can be delivered in one day?

Multiple sources and sinks

What if we have m factories $s_1, s_2, \ldots, s_m$ and n warehouses $t_1, t_2, \ldots, t_n$?

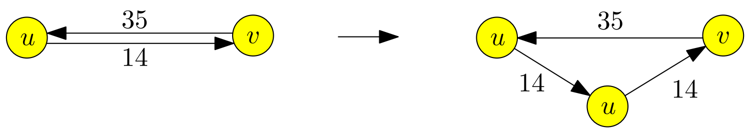


Solution: add supersource s and supersink t.



Antiparallel edges

What if the input graph has antiparallel edges? Can be fixed by adding a new vertex for two antiparallel edges (u, v) and (v, u).



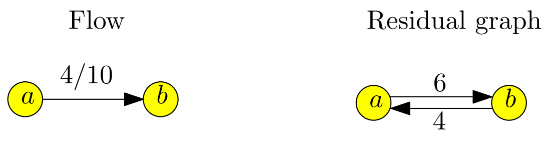
Example. Let G' be the graph obtained from G by fixing all antiparallel edges. Explain how a flow f' for G' can be transformed to a flow f for G.

Residual graph

Residual capacity is defined as

$$r(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The residual graph $G' = (V, E')$ where E' is the set of edges (u, v) such that $r(u, v) > 0$.
! Note that E_f may contain edges not present in E.



To increase the flow, we find a directed path from s to t in G' called augmented path and saturate it. To saturate a path p, we add a flow ϕ along the path p where

$$\phi = \min_{(u,v) \in p} \{r(u, v)\}.$$

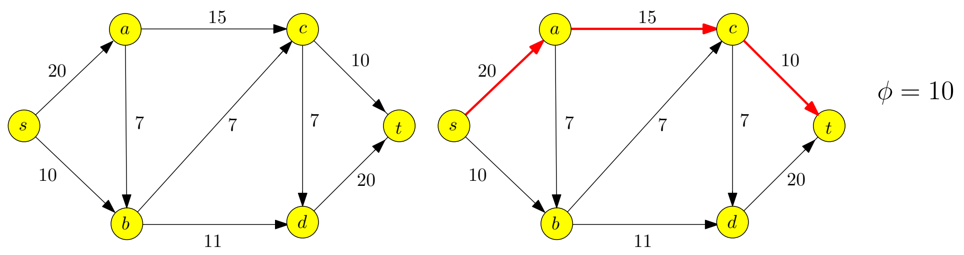
An edge (u, v) of p is called a bottleneck edge if $r(u, v) = \phi$.

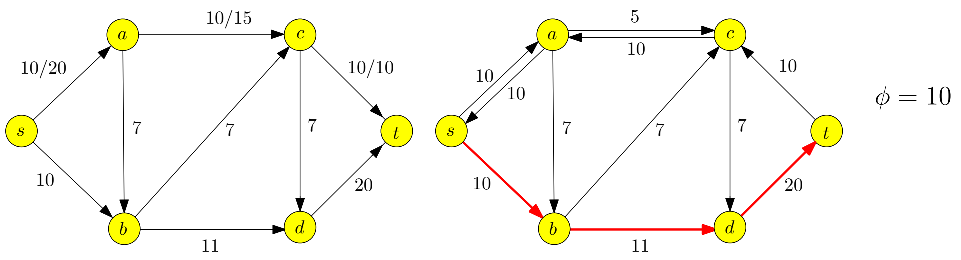
8.2 Ford-Fulkerson algorithm

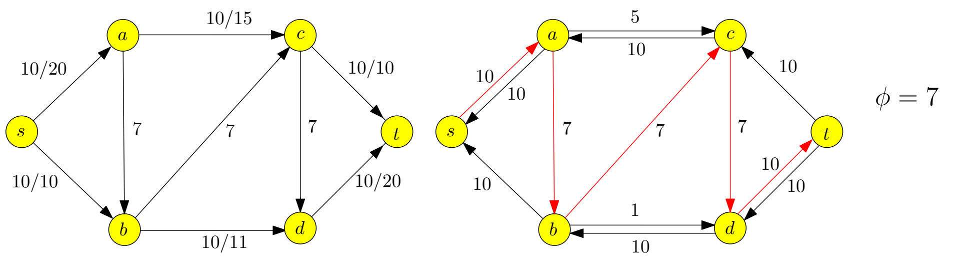
```
Ford-Fulkerson (G, s, t)
1   initialize flow $f$ to 0
2   while true
3       compute residual graph $G'$
4       if there exists an augmenting path $p$ in $G'$
5           saturate $p$
6       else return $f$
```

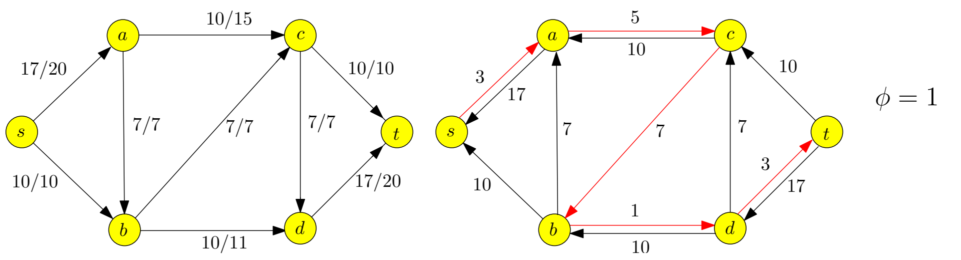
Running time. If the weights are integers, then $T = O(E \max |f|)$.
If the weights are real numbers, then F-F may not even terminate.

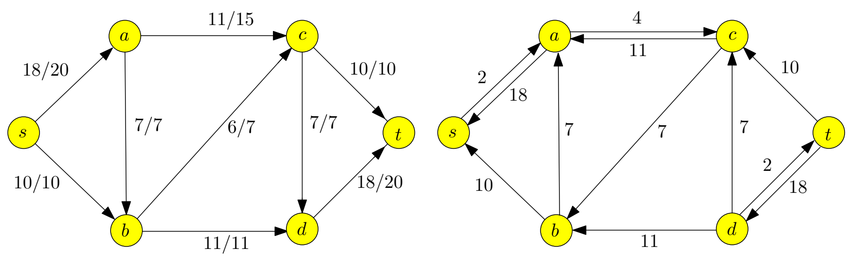
Example


a
c
s
t
b
d
15
20
10
7
7
7
10
20
11
$\phi = 10$


10/15
10/20
10/10
7
7
7
10
20
11
5
10
10
10
10
20
11
$\phi = 10$


10/15
10/20
10/10
7
7
7
10/10
10/20
10/11
5
10
10
10
10
10
10
1
10
$\phi = 7$


10/15
17/20
10/10
7/7
7/7
7/7
10/10
17/20
10/11
5
10
3
17
10
10
3
17
1
10
$\phi = 1$


11/15
18/20
10/10
7/7
6/7
7/7
10/10
18/20
11/11
4
11
2
18
10
10
2
18
11

Cuts

A cut $(S, T), T = V - S$ separates s and t if $s \in S$ and $t \in T$.
The capacity of the cut (S, T) is defined as the sum of $c(u, v)$ for all $(u, v) \in E, u \in S, v \in T$.

Lemma. For any cut (S, T) and any flow f, $|f| \leq c(S, T)$.

Proof.

$$\begin{aligned} |f| &= \sum_{u \in V} (f(s, u) - f(u, s)) \\ &= \sum_{v \in S, u \in V} (f(v, u) - f(u, v)) \\ &= \sum_{v \in S, u \in T} (f(v, u) - f(u, v)) \\ &\leq \sum_{v \in S, u \in T} f(v, u) \\ &\leq \sum_{v \in S, u \in T} c(v, u) \\ &= c(S, T). \end{aligned}$$

Max-flow Min-cut Theorem. The value of the maximum flow is equal to the capacity of the minimum cut.

Proof. Suppose that f is the maximum flow, then there is no augmenting path in G'. Let S be the set of vertices reachable from s in G'. Let $T = V - S$. Flow f saturates every edge from S to T and $f(u, v) = 0$ if $u \in T$ and $v \in S$. Then $|f| = c(S, T)$. By the above lemma, (S, T) is a minimum cut.

8.3 Edmonds-Karp algorithm

The idea is to find an augmented path with minimum number of edges. This is done by a breadth-first search.

Suppose that flow f is augmented to f' by one iteration of E-K algorithm.
Let R be the residual graph corresponding to f and let R' be the residual graph corresponding to f'.
Let $d(v)$ denote the shortest-path distance (the number of edges) from s to v in R.
Let $d'(v)$ denote the shortest-path distance (the number of edges) from s to v in R'.

Lemma. $d'(v) \geq d(v)$ for any $v \in V$.

Proof by contradiction. Suppose that $d'(v) < d(v)$ for some $v \in V$. Since $d'(s) = d(s) = 0$, vertex $v \neq s$. Since $d'(u) = \infty$ for any vertex u not reachable from s in R', we assume that v is reachable from s in R'. Let $s, \ldots, u, v$ be the shortest path from s to v in R'. We can assume that v is the first vertex in this path satisfying $d'(v) < d(v)$. Therefore $d'(u) \geq d(u)$.

We prove that $d(v) \leq d(u) + 1$. It follows if (u, v) is an edge in R. If (u, v) is not an edge in R (but it is an edge of R') then (v, u) is in the augmenting path of R. Then $d(v) = d(u) - 1 < d(u) + 1$.

Therefore $d'(v) = d'(u) + 1 \geq d(u) + 1 \geq d(v)$. Contradiction. ∎

Lemma. For any $u, v \in V$, (u, v) is the bottleneck edge of at most $V/2$ residual graphs.

Proof. Suppose that (u, v) is the bottleneck edge of the augmenting path in R. Then $d(v) = d(u) + 1$ (since the augmenting path is the shortest one). It disappears from R' (why?). It can appear later only of (v, u) appears in an augmenting path. Wlog (v, u) appears in an augmenting path of R'. Then $d'(u) = d'(v) + 1 \geq d(v) + 1 = d(u) + 2$. So, the distance from s to v increases by ≥ 2 when (u, v) is the bottleneck edge next time. ∎

Theorem. Edmonds-Karp algorithm performs $O(VE)$ augmentations. The total running time is $O(VE^2)$.

Proof. At least one edge is the bottleneck edge in each E-K iteration. Then there are $O(VE)$ iterations. The running time follows since every iteration requires $O(E)$ time. ∎

8.4 Maximum bipartite matching

Consider an undirected graph $G = (V, E)$.

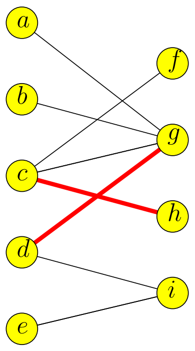
A matching in G is defined as a subset of edges $M \subseteq E$ such that any two edges of M do not share a common vertex.

A vertex is matched if it is an endpoint of some edge in M; otherwise it is unmatched.

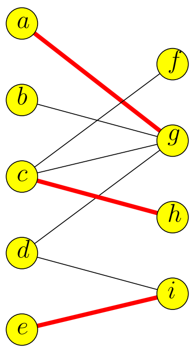
A maximum matching is a matching with maximum number of edges.

A graph is bipartite if V can be partitioned into V_1 and $V_2 = V - V_1$ such that every edge of G has one endpoint in V_1 and one endpoint in V_2.

Example. A matching of cardinality 2 is show below. Is it a maximum matching?

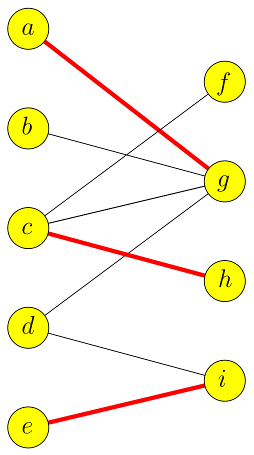


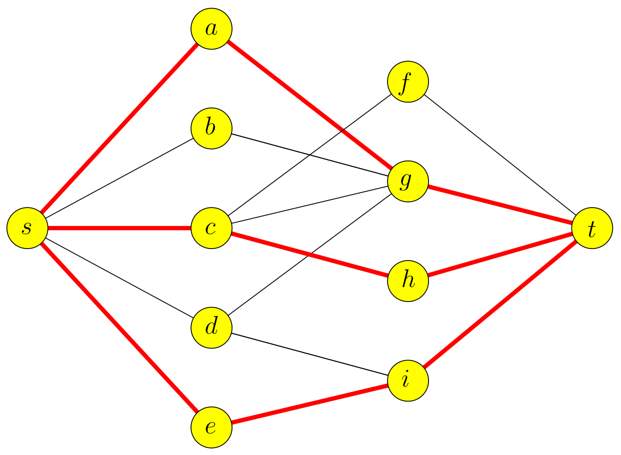
Solution.



Finding matching

Assign unit capacity to every edge.





Theorem.

A) If M is a maximum matching in G then the corresponding flow has value $|M|$.

B) If f is a maximum flow in G' then it corresponds to a matching M with $|M| = |f|$.

Conclusion: F-F algorithm or E-K algorithm can be used to compute a maximum matching in a bipartite graph.

Index