ECE 406 COURSE NOTES
ALGORITHM DESIGN AND ANALYSIS

Paolo Torres

University of Waterloo
Winter 2021

Table of Contents

1 INTRODUCTION AND BASIC ARITHMETIC

1.1 Algorithms, Correctness, Termination, Efficiency

1.1.1 Algorithms

Given the specification for a function, an algorithm is the procedure to compute it.

Example:

$$F: Z_o^+ \to Z_o^+, \text{ where } F(n) = \begin{cases} 0, if\ n = 0 \\ 1, if\ n = 1 \\ F(n-1) + F(n-2), otherwise \end{cases}$$

Commonly used sets: $N, Z(all\ ints), Z^+(positive\ ints), Z_o^+(non-negative\ ints), R, \ldots$

Fibonacci Sequence:

$$FIB_1(n)$$
$$1\ if\ n = 0\ then\ return\ 0$$
$$2\ if\ n = 1\ then\ return\ 1$$
$$3\ return\ FIB_1(n-1) + FIB_1(n-2)$$

Important aspects:

- Function has been specified as a recurrence, so a recursive algorithm seems natural
- Imperative (procedural) specification of an algorithm has consequences:
    - Intuiting correctness can be a challenge
    - Intuiting time and space efficiency may be easier
- No mundane error checking, can focus on core logic
- Input value n is unbounded but finite

1.1.2 Correctness

Correctness refers to an algorithm's ability to guarantee expected termination. In the case of FIB_1, it is a direct encoding of the recurrence.

1.1.3 Termination

The end of an algorithm. It can be proven that FIB_1 terminates on every input $n \in Z_o^+$ by induction on n.

1.1.4 Time Efficiency

Can be calculated by counting the number of: (i) comparisons – these happen on Lines (1) and (2), and (ii) number of additions – this happens on Line (3).

Suppose $T(n)$ represents the time efficiency of FIB_1:

$$T(n) = \begin{cases} 1, if\ n = 0 \\ 2, if\ n = 1 \\ 3 + T(n-1) + T(n-2), otherwise \end{cases}$$

How bad is $T(n)$? Is it exponential in n?

For all n, $T(n) \geq F(n)$.

1.1.5 Claim 1

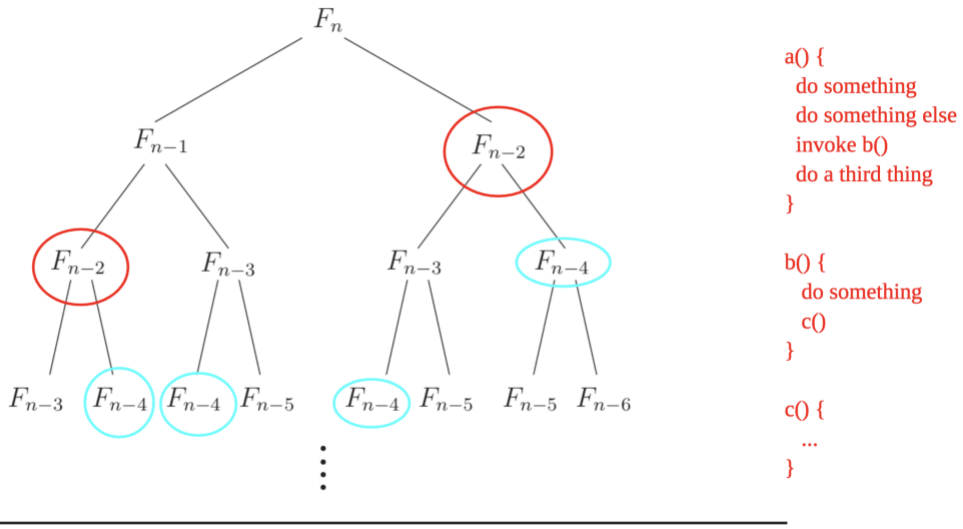
For all $n \in Z_o^+, F(n) \geq \left(\sqrt{2}\right)^n$.

If this claim is true, then $T(n) \geq \left(\sqrt{2}\right)^n$, and because $\sqrt{2} > 1$, $T(n)$ is exponential in n.

Proof for the claim: by induction on n.

Does a better algorithm exist from the standpoint of time efficiency?

Figure 0.1 The proliferation of recursive calls in `fib1`.



Recall how subroutine (recursive, in this case) invocation works:

- Every node in the tree corresponds to an invocation of the algorithm
- Sequence of invocations corresponds to a pre-order traversal
- Maximum depth of the call stack at any moment: n

Main point in this case: Redundancy, F_i, appears more than once.

1.1.6 More Efficient Algorithm

$FIB_2(n)$

1 $if\ n = 0\ then\ return\ 0$

2 $create\ an\ array\ f[0, \dots, n]$

3 $f[0] \leftarrow 0, f[1] \leftarrow 1$

4 $foreach\ i\ from\ 2\ to\ n\ do$

5 $f[i] \leftarrow f[i-1] + f[i-2]$

6 $return\ f[n]$

Let $U(n)$ be the # of comparisons plus additions on input n:

$$U(n) = \begin{cases} 1, if\ n = 0 \\ n, otherwise \end{cases}$$

Linear in n for $n \geq 1$, more efficient than FIB_1.

1.1.7 Note on Measuring Time Efficiency

Need to pick the right level of abstraction, meaning picking some kind of "hot spot" or "hot operation," then count. For example, number of additions, comparisons, recursive calls, etc.

1.2 Big-O Notation

1.2.1 Definition 1 (O)

Let $f: N \rightarrow R^+$, and $g: N \rightarrow R^+$ be functions. Define $f = O(g)$ if there exists a constant $c\ \epsilon\ R^+$ such that $f(n) \leq c \cdot g(n)$.

- $N = Z^+: \{1,2,3, \dots\}, R^+: set\ of\ positive\ real\ numbers$
- Typically consider non-decreasing functions only

1.2.2 Definition 2 (Ω)

Define $f = \Omega(g)$ if $g = O(f)$

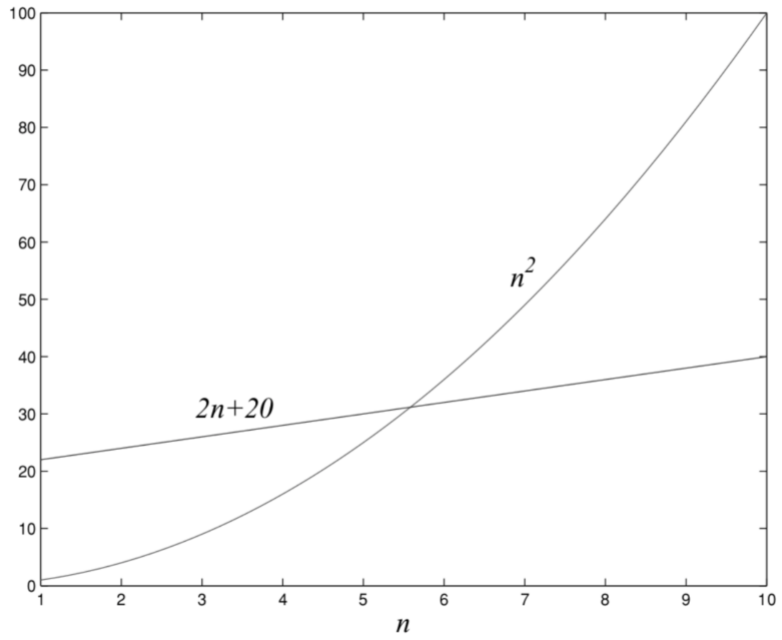
1.2.3 Definition 3 (Θ)

Define $f = \Theta$(g) if $f = O(g)$ and $f = \Omega(g)$

- $f = O(g)$ analogous to $f \leq g$
- $f = \Omega(g)$ analogous to $f \geq g$
- $f = \Theta(g)$ analogous to $f = g$

1.2.4 Example

Figure 0.2 Which running time is better?



Precise answer to this question: depends on n.

But in big-O notation:

- $2n + 20 = O(n^2)$. Proof: Adopt as the constant $c \epsilon R$ for any $c > 22$
- $2n + 2 \neq \Omega(n^2) \therefore 2n + 2 \neq \Theta(n^2)$

1.2.5 Big-O Explanation

Suppose algorithm A runs in $2n + 20$ time, B in n^2, and C in 2^n. Now suppose the speed of the computer doubles, which algorithm gives the best payoff?

For a given time period t, what is the largest input n each algorithm can handle? Set $runtime = t$ and $runtime = 2t$, solve for n:

Algorithm	Old Computer	New Computer
A	$t/2 - 10$	$t - 10$
B	$\sqrt{t}$	$\sqrt{2} \cdot \sqrt{t}$
C	$\log_2 t$	$1 + \log_2 t$

So, payoff with algorithm A is approximately $2x$, B is $1.4x$, and C is $1 +$.

1.2.6 Big-O Simplifications

- Multiplicative and additive constants can be omitted
- n^a dominates n^b for $a > b \geq 0$
- Any exponential dominates any polynomial, any polynomial dominates any logarithm
- Big-O simplifications should be used prudently, not applicable in all settings

1.3 Arithmetic

1.3.1 Addition

Hypothesize access to a function $T : \{0,1\} \times \{0,\dots,9\} \times \{0,\dots,9\} \to \{0,1\} \times \{0,\dots,9\}$:

Carry	One Digit	Other Digit	Result Carry	Result Sum
0	0	0	0	0
1	0	0	0	1
0	0	1	0	1
…				
1	8	9	1	8
0	9	9	1	8
1	9	9	1	9

To add 7,814 and 93,404:

$$C \to 1\ 1\ 1\ 0\ 0\ 0$$
$$1 \to 0\ 0\ 7\ 8\ 1\ 4$$
$$2 \to 0\ 9\ 3\ 4\ 0\ 4$$
$$T \to 1\ 0\ 1\ 2\ 1\ 8$$

digits needed to encode $x \in Z^+ = \lfloor \log_{10} x \rfloor + 1$

bits needed to encode $x \in Z^+ = \lfloor \log_2 x \rfloor + 1$

So, time efficiency of an algorithm to add $x, y \in Z^+$ as measured by number of lookups to T:

- $1 + \lfloor \log_{10}(\max\{x, y\}) \rfloor$ in the best case
- $2 + \lfloor \log_{10}(\max\{x, y\}) \rfloor$ in the worst case
- So, either way, $\Theta(n)$, or linear time, where n is the size of the input

1.3.2 Multiplication

For $x, y \in Z_o^+$, encoded in binary:

$$x \times y = \begin{cases} 0, if\ y = 0 \\ 2(x \times {}^{y}/_{2}), if\ y\ even, y > 0 \\ x + 2(x \times \lfloor {}^{y}/_{2} \rfloor), otherwise \end{cases}$$

Straightforward encoding as recursive algorithm $MULTIPLY(x, y)$.

Figure 1.1 Multiplication à la Français.

```
function multiply(x, y)
Input: Two n–bit integers x and y, where y ≥ 0
Output: Their product

if y = 0: return 0
z = multiply(x, ⌊y/2⌋)
if y is even:
    return 2z
else:
    return x + 2z
```

Worst case running time:

- *Let # bits to encode each of* x *and* $y = n$
- # *recursive calls* $= \Theta(n)$
- *In each call*:
    - One comparison to 0, one division by 2 (right bit shift), one assignment to z, one check for evenness (check LSB), one multiplication by 2 (left bit shift), one addition of $O(n)$-bit numbers
- So, $O(n^2)$ in the worst case

1.3.3 Division

Definition 1: Given $x \in Z_o^+, y \in Z^+$, the pair $\langle q, r \rangle$ where $q \in Z_o^+$, $r \in \{0, 1, \dots, y-1\}$ of x divided by y are those that satisfy:

$$x = q \cdot y + r$$

Claim 1: For every $x \in Z_o^+$, $y \in Z^+$, $\langle q, r \rangle$ as defined above (i) exists, and (ii) is unique.

To specify a recurrence for $\langle q, r \rangle$, denote as $\langle q', r' \rangle$, the result of $\lfloor {}^{x}/_{2} \rfloor$ divided by y. Now:

$$\langle q, r\rangle = \begin{cases} \langle 0,0\rangle, if\ x = 0 \\ \langle 2q', 2r'\rangle, if\ x\ even\ and\ 2r' < y \\ \langle 2q', 2r'+1\rangle, if\ x\ odd\ and\ 2r'+1 < y \\ \langle 2q'+1, 2r'-y\rangle, if\ x\ even\ and\ 2r' \geq y \\ \langle 2q'+1, 2r'+1-y\rangle, otherwise \end{cases}$$

Claim 2: The above recurrence is correct.

Proof: Cases are exhaustive. Proof by case-analysis and induction on # bits to encode x.

By induction assumption: $0 \leq r' \leq y-1 \therefore 0 \leq 2r' \leq 2y-2$.

Figure 1.2 Division.

```
function divide(x, y)
Input: Two n–bit integers x and y, where y ≥ 1
Output: The quotient and remainder of x divided by y

if x = 0: return (q, r) = (0, 0)
(q, r) = divide(⌊x/2⌋, y)
q = 2 · q,  r = 2 · r
if x is odd: r = r + 1
if r ≥ y: r = r − y,  q = q + 1
return (q, r)
```

Running time: $O(n^2)$.

2 ALGORITHMS WITH NUMBERS

2.1 Modular Arithmetic

- **Definition 1:** $For\ x \epsilon Z, N \epsilon Z^{+}, x\ modulo\ N\ is\ the\ remainder\ of\ x\ divided\ by\ N$
- **Definition 2:** $x \equiv y\ (mod\ N)\ if\ N\ divides\ x - y$."≡" read as "$congruent\ to$"
    - Example: $373 \equiv 13\ (mod\ 60), 59 \equiv -1\ (mod\ 60)$

2.1.1 Example Application: Two's Complement Arithmetic

Suppose we want to represent, using n bits, positive and negative integers, and 0. Could reserve 1 bit for sign. This would allow us to represent integers in the interval $[-(2^{n-1}-1), 2^{n-1}-1]$, with a "positive zero" and a "negative zero."

In two's complement arithmetic, we have exactly one bit-string for 0, and represent integers in the interval $[-2^{n-1}, 2^{n-1}-1]$. How? Represent any $x \epsilon [-2^{n-1}, 2^{n-1}-1] \cap Z$ as the non-negative integer modulo 2^n. So:

$$0 \leq x \leq 2^{n-1}-1 \rightarrow x\ is\ represented\ as\ x$$

- Ex: $For\ n = 5, (9)_{10}\ written\ as\ (01001)_2$

$$-2^{n-1} \leq x < 0 \rightarrow x\ is\ represented\ as\ 2^n + x$$

- Ex: $For\ n = 5, (-9)_{10} \equiv 32 + (-9) \equiv 23\ (mod\ 32), written\ as\ (10111)_2$

All arithmetic performed modulo 2^n:

- Ex: $For\ n = 5, 13 + (-7) \equiv 01101 + 11001 \equiv 100110 \equiv 00110 \equiv 6\ (mod\ 32)$

Claim 1: $x \equiv x', y \equiv y' (mod\ N)\ implies: x + y \equiv x' + y', xy \equiv x'y' (mod\ N)$

Claim 2:

$$x + (y + z) \equiv (x + y) + z \quad \text{(mod N)}$$
$$xy \equiv yx \quad \text{(mod N)}$$
$$x(y + z) \equiv xy + xz \quad \text{(mod N)}$$

Example: $2^{3045} \equiv (2^5)^{609} \equiv (1)^{609} \equiv 1\ (mod\ 31)$

2.1.2 Modular Addition, Subtraction

$$x + y \equiv \begin{cases} x + y\ (mod\ N), if\ 0 \leq x + y < N \\ x + y - N\ (mod\ N), otherwise \end{cases}$$

$$x - y \equiv \begin{cases} x - y\ (mod\ N), if\ 0 \leq x - y < N \\ x - y + N\ (mod\ N), otherwise \end{cases}$$

- Any intermediate result is between $-(N-1)$ and $2(N-1)$
- So, time efficiency is $O(n)$, where $n = \lceil \log N \rceil$

2.1.3 Modular Multiplication

Let $MULT$ and DIV be our algorithms for non-modular multiplication and division:

$$x \times y \equiv r \ (mod\ N), where\ \langle q, r \rangle = DIV(MULT(x, y), N)$$

- Any intermediate result (specifically, result of $MULT$) is between 0 and $(N-1)^2$
- So, time efficiency is $O(n^2)$, where $n = \lceil \log N \rceil$

2.1.4 Modular Exponentiation

- Recall: We used "repeated doubling" for non-modular multiplication and "repeated halving" for non-modular division
- Similarly, here, use "repeated squaring":

$$x^y = \begin{cases} 1, if\ y = 0 \\ (x^2)^{\lfloor y/2 \rfloor}, if\ y\ is\ even \\ x \cdot (x^2)^{\lfloor y/2 \rfloor}, otherwise \end{cases}$$

Figure 1.4 Modular exponentiation.

```
function modexp(x, y, N)
Input: Two n–bit integers x and N, an integer exponent y
Output: x^y mod N

if y = 0: return 1
z = modexp(x, ⌊y/2⌋, N)
if y is even:
   return z^2 mod N
else:
   return x · z^2 mod N
```

$Time\ Efficiency: O(n^3)$

2.1.5 Towards Modular Division: GCD Using Euclid

- In a non-modular world, $a/b = a \times b^{-1}$. Only case for which b^{-1} doesn't exist: $b = 0$
- In a modular world, $b^{-1}\ (mod\ N)$ may not exist even if $b \not\equiv 0\ (mod\ N)$

Crucial building block: GCD. Example: What is $\gcd(1035, 759)$?

$1035 = 3^2 \cdot 5 \cdot 23 \rightarrow 759 = 3 \cdot 11 \cdot 23 \rightarrow \gcd(1035, 759) = 3 \cdot 23$

But factoring into prime factors conjectured to be computationally hard in the worse case

Claim 3: $x, y \in \mathbb{Z}^+, x \geq y \rightarrow \gcd(x, y) = \gcd(x \bmod y, y)$

Proof: Suffices to prove: $\gcd(x, y) = \gcd(x - y, y)$. Now prove $\leq$ and $\geq$.

Figure 1.5 Euclid's algorithm for finding the greatest common divisor of two numbers.

```
function Euclid(a, b)
Input: Two integers a and b with a ≥ b ≥ 0
Output: gcd(a, b)

if b = 0: return a
return Euclid(b, a mod b)
```

How fast does *Euclid* converge?

Claim 4: $a \geq b \geq 0 \rightarrow a \bmod b < a/2$

So: Guaranteed to lose at least 1 bit for every recursive call → time efficiency is $O(n^3)$

2.1.6 Towards Modular Division: Extended Euclid

Claim 5: d *divides* a *and* b, *and* $d = ax + by$ *for some* $x, y \in \mathbb{Z} \rightarrow$ d = gcd (a, b)

Claim 6: *Let* $d = \gcd(a, b)$, $d = ax + by$ *and* $d = bx' + (a \bmod b)y'$ *for some* $x, y, x', y' \in \mathbb{Z}$. *Then*:

$$\langle x, y \rangle = \begin{cases} \langle 1, 0 \rangle, if\ b = 0 \\ \langle y', x' - \lfloor a/b \rfloor y' \rangle, otherwise \end{cases}$$

Figure 1.6 A simple extension of Euclid's algorithm.

```
function extended−Euclid(a, b)
Input: Two positive integers a and b with a ≥ b ≥ 0
Output: Integers x, y, d such that d = gcd(a, b) and ax + by = d

if b = 0: return (1, 0, a)
(x', y', d) = extended−Euclid(b, a mod b)
return (y', x' − ⌊a/b⌋y', d)
```

Example run on input $\langle 359, 82 \rangle$:

Arguments	Return Value
$\langle 359, 82 \rangle$	$\langle -37, 162, 1 \rangle$
$\langle 82, 31 \rangle$	$\langle 14, -37, 1 \rangle$
$\langle 31, 20 \rangle$	$\langle -9, 14, 1 \rangle$
$\langle 20, 11 \rangle$	$\langle 5, -9, 1 \rangle$
$\langle 11, 9 \rangle$	$\langle -4, 5, 1 \rangle$
$\langle 9, 2 \rangle$	$\langle 1, -4, 1 \rangle$
$\langle 2, 1 \rangle$	$\langle 0, 1, 1 \rangle$
$\langle 1, 0 \rangle$	$\langle 1, 0, 1 \rangle$

A good example to segue to our final step in modular division.

We figured out: $\gcd(359,82) = 1$, which implies:

- $82 \times 162 \equiv 1\ (mod\ 359)$. So: 162 is multiplicative inverse of $82\ modulo\ 359$
- So, for example: $116\ divided\ by\ 82\ modulo\ 359\ \equiv 116 \times 162 = 124\ (mod\ 359)$

2.1.7 Modular Division

Definition 3: $x\ is\ the\ multiplicative\ inverse\ of\ a\ modulo\ N\ if\ ax \equiv 1\ (mod\ N)$

Claim 7: $For\ every\ a\ \epsilon\ \{0, \dots, N-1\}, there\ exists\ at\ most\ a^{-1}\ (mod\ N)$

Claim 8: $Given\ \langle a, N \rangle, where\ a\ \epsilon\ \{0, \dots, N-1\}, a^{-1}\ (mod\ N)\ may\ not\ exist$

Definition 4: $If\ \gcd(x, y) = 1, then\ we\ say\ that\ x\ is\ relatively\ prime\ to\ y$

Claim 9: $a^{-1}\ (mod\ N)\ exists\ if\ and\ only\ if\ a\ and\ N\ are\ relatively\ prime$

So, to compute $a/b\ (mod\ N)$:

(i) Determine whether $\gcd(b, N) = 1$

(ii) If yes to (i) determine $b^{-1}\ (mod\ N)$, and

(iii) If yes to (i) compute $a \times b^{-1}\ (mod\ N)$

(i) and (ii) are done simultaneously by $extended - Euclid$

Running Time: $(i) + (ii) = O(n^3), (iii) = O(n^2). So\ (i) + (ii) + (iii) = O(n^3)$

2.2 Primality Testing

Given $n\ \epsilon\ \mathbb{Z}^+$, is n prime?

For a decision problem, i.e., co-domain of function to be computed as $\{true, false\}$, a randomized algorithm:

- Has access to an unbiased coin
- Is deemed to be correct if:
    - $\Pr\{Algorithm\ outputs\ false \mid input\ instance\ is\ false\} = 1$
    - $\Pr\{Algorithm\ outputs\ true \mid input\ instance\ is\ true\} \geq {}^{1}/_{2}$

Suppose:

- We run such an algorithm k times, pairwise independently
- We return $true$ if and only if every run returns $true$
- Then, $\Pr\{we\ return\ true\ incorrectly\} \leq 2^{-k}$

2.2.1 Fermat's Little Theorem

Claim 1: $p\ prime \rightarrow for\ all\ a \in [1, p) \cap \mathbb{Z}, a^{p-1} \equiv 1\ (mod\ p)$

To prove Fermat's little theorem, leverage the following:

Claim 2: $p\ prime, a, i, j \in \{1, 2, \dots, p-1\}\ and\ i \neq j \rightarrow a \cdot i \not\equiv a \cdot j\ (mod\ p)$

Proof for Claim 2: We know that a, p are relatively prime, so $a^{-1}\ (mod\ p)$ exists.

$$a \cdot i \equiv a \cdot j\ (mod\ p) \rightarrow a \cdot i \cdot a^{-1} \equiv a \cdot j \cdot a^{-1}\ (mod\ p) \rightarrow i \equiv j\ (mod\ p) \rightarrow i = j$$

Proof for Claim 1: From Claim 2:

$$\{1, 2, \dots, p-1\} = \{a \cdot 1\ mod\ p, a \cdot 2\ mod\ p, \dots, a \cdot (p-1)\ mod\ p\}$$

Also, $(p-1)!^{-1}\ (mod\ p)$ exists.

So, $(p-1)! \equiv a^{p-1} \cdot (p-1)!\ (mod\ p) \rightarrow a^{p-1} \equiv 1\ (mod\ p)$

Figure 1.7 An algorithm for testing primality.

```
function primality(N)
Input: Positive integer N
Output: yes/no

Pick a positive integer a < N at random
if a^(N-1) ≡ 1 (mod N):
   return yes
else:
   return no
```

Issues with the algorithm:

1. Fermat's little theorem is not an "if and only if": Carmichael numbers.

2. Suppose N is not prime/Carmichael. For a chosen, say, uniformly from $\{1, \dots, N-1\}$, what is Pr $\{a^{N-1} \not\equiv 1 \ (mod\ N)\}$?
    - We know such an a exists, but how likely is it that we will pick it?

We do not deal with (1) – cop out: Carmichael numbers are rare.

Claim 3: If $a^{N-1} \not\equiv 1 \ (mod\ N)$ for a, N relatively prime, then it must hold for at least half the choices $a \in \{1, \dots, N-1\}$.

Proof: If there exists no $b \in \{1, \dots, N-1\}$ with $b^{N-1} \equiv 1 \ (mod\ N)$, then we are done.

If such a b exists, then $(b \cdot a)^{N-1} \not\equiv 1 \ (mod\ N)$.

Also, if b, c exist with $b \neq c$, $b^{N-1} \equiv c^{N-1} \equiv 1 \ (mod\ N)$, then:

$$b \cdot a \not\equiv c \cdot a \ (mod\ N)$$

So, at least as many $\not\equiv 1 \ (mod\ N)$ as there are $\equiv 1 \ (mod\ N)$.

So:

$$\Pr\{Algorithm\ 1.7\ returns\ yes\ when\ N\ is\ prime/Carmichael\} = 1$$

$$\Pr\{Algorithm\ 1.7\ returns\ yes\ when\ N\ is\ not\ prime/Carmichael\}, < 1/2$$

For k runs of $Algorithm$ 1.7 on uniform, independent choices of a:

$$\Pr\{Algorithm\ 1.7\ returns\ yes\ on\ all\ k\ runs\ when\ N\ is\ not\ prime/Carmichael\} \leq 2^{-k}$$

2.3 Generating an n-Bit Prime

Claim 4: $\Pr\{uniformly\ chosen\ n-bit\ number\ is\ prime\} \approx 1/n.$

So, algorithm for generating a prime:

1. Randomly generate n-bit number, r.
2. Check whether r is prime.
3. If not, go to Step (1).

Guaranteed return in Step (2) if r is indeed prime.

Each trial in the above algorithm is a Bernoulli trial:

- Only one of two outcomes: success or failure

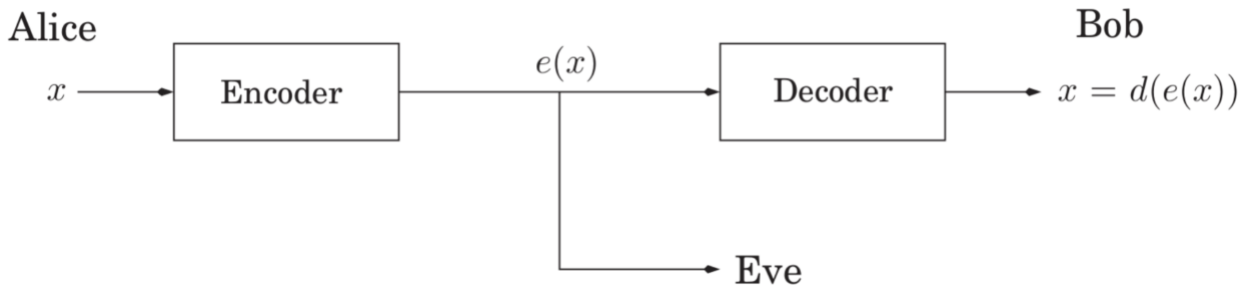
In a Bernoulli trial if $\Pr\{success\} = p$, then $expected\ \#\ trials\ to\ see\ a\ success = 1/p$

- Example: $Expected\ \#\ tosses\ of\ fair\ coin\ to\ see\ a\ heads = 2$
- Example: $Expected\ \#\ tosses\ of\ fair\ die\ to\ see, say, a\ 3 = 6$

So, we expect the above algorithm to halt in n iterations.

2.4 Cryptography

RSA: Exploits presumed computational hardness of factoring vs. computational ease of GCD, primality testing and modular exponentiation.



Symmetric Key Cryptography:

- $e(\cdot) = d(\cdot) \rightarrow Alice\ and\ Bob\ both\ know\ e(\cdot)\ and\ d(\cdot)$
    - Example: $e(x) = x \oplus r, d(y) = y \oplus r$
- They keep these secrets from everyone else
- Bootstrapping Problem: How do Alice and Bob share $e(\cdot)$, $d(\cdot)$?

Public Key Cryptography:

- Bob publishes $e(\cdot)$ to the whole world
- Bob keeps $d(\cdot)$ to himself
- RSA is an example of a public key cryptography scheme

2.4.1 Claim 1

Let p, q be primes and $N = pq$. For any e relatively prime to $(p-1)(q-1)$:

1. The function $f: \{0, 1, \dots, N-1\} \rightarrow \{0, 1, \dots, N-1\}$ where $f(x) = x^e \ mod\ N$ is a bijection
2. Let $d = e^{-1}\ (mod\ \ (p-1)(q-1))$. Then, $(x^e)^d \equiv x\ (mod\ N)$

Bob publishes the pair $\langle N, e \rangle$. Alice encodes message x as $f(x)$ from the claim.

Bob keeps the d in the claim secret to themselves.

2.4.2 Example

$p = 11, q = 17$. Then, $N = pq = 187$. The only messages that can be sent: $\{0, 1, \dots, 186\}$.

We could pick $e = 7$: $\gcd(10 \times 16, 7) = 1$

Then, $d = 23$: $7^{-1} \equiv 23\ (mod\ 160)$.

To send the message 98, Alice would send $98^7 \ mod \ 187 = 21$.

Bob would decode the message as $21^{23} \ mod \ 187 = 98$.

2.4.3 Attacks

Attacker knows: (i) $\langle N, e \rangle$, (ii) $x^e \ mod \ N$.

- Attack 1: Attacker determines x given (i) and (ii).
- Attack 2: Even more devastating – attacker factors $N = p \cdot q$.
    - They can then compute $d = e^{-1} \ (mod \ (p-1)(q-1))$.

2.4.4 Proof for Claim 1

Property #2 implies Property #1.

To prove Property #2:

Because $ed \equiv 1 \ \big(mod \ (p-1)(q-1)\big), ed = 1 + k(p-1)(q-1)$ for some $k \ \epsilon \ \mathbb{Z}$.

We seek to show: $x^{ed} - x = x^{1+k(p-1)(q-1)} - x$ is divisible by N, and is therefore $\equiv 0 \ mod \ N$.

Now, by Fermat's little theorem: $x \cdot (x^{p-1})^{k(q-1)} - x \equiv 0 \ mod \ p$.

And again, by Fermat's little theorem: $x \cdot (x^{q-1})^{k(p-1)} - x \equiv 0 \ mod \ q$.

So: $x^{ed} - x$ is divisible by the product of the two primes $pq = N$.

Figure 1.9 RSA.

Bob chooses his public and secret keys.

- He starts by picking two large (n-bit) random primes p and q.
- His public key is (N, e) where $N = pq$ and e is a $2n$-bit number relatively prime to $(p-1)(q-1)$. A common choice is $e = 3$ because it permits fast encoding.
- His secret key is d, the inverse of e modulo $(p-1)(q-1)$, computed using the extended Euclid algorithm.

Alice wishes to send message x to Bob.

- She looks up his public key (N, e) and sends him $y = (x^e \bmod N)$, computed using an efficient modular exponentiation algorithm.
- He decodes the message by computing $y^d \bmod N$.

Example:

```
$ ssh-keygen -t rsa
...
$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDJAs5HIayjHG
LdvEeiaRI2R3TG8+chfGYrjEWc82bV3ndC87+dYAFVXyVDDc2C
0vDHY6cNcN4vpjc0fZbieeJWCOwjFV8qt5VZDTdvtLJSBilH1j
lJI6FoGBjwMjqoDsXR0n3e7rundqr0xLsk6RoIVunhluloj2Ss
L2fwU7/pbhrvWBBx1jP6aaCkW5sAEu143xM71C2bAMqzoS47WY
+xH91sgm8hwji/KUEoHeeVMrc54bTsozPQp4t+3QwjDfqEMeTy
BvJ93ZTZFHJiQVORTnw3x8HyzNgYTDZVnFJi6kWx8suggcokgz
ffAM+7xNJlzDmJlbY3N+WHuRDsvFpf me@localhost
```

3 UNIVERSAL HASHING, DIVIDE AND CONQUER

3.1 Universal Hashing

3.1.1 Example

University of Waterloo has data about students that looks something like:

Student ID	Last Name	First Name	Degree Objective	Date of Matriculation	Term 1 + Marks	…

The Student ID is used to uniquely identify each student record. We call it a *key*. The remainder of the data for each student is called *satellite data*. The school seeks to store this data and retrieve it efficiently. Assume that $\mathcal{D}$ is the data structure that stores all the data. For any given student record, r, there is an efficient function $key(r)$, which returns the key, i.e., Student ID, associated with r.

- $CreateNew(\dots)$: Creates a new, empty instance of $\mathcal{D}$ and returns it. May take arguments, e.g., the space allocated to $\mathcal{D}$.
- $Insert(\mathcal{D}, r)$: Insert the record r into $\mathcal{D}$. If a record with $key(r)$ already exists in $\mathcal{D}$, adopt some convention, e.g., replace it with this new one.
- $Delete(\mathcal{D}, k)$: Remove the record associated with key k in $\mathcal{D}$. If such a record does not exist, adopt some convention, e.g., leave $\mathcal{D}$ unchanged.
- $Search(\mathcal{D}, k)$: Return the record associated with key k from $\mathcal{D}$. If such a record does not exist, return a special string, e.g., ϵ.

What data structure, and associated algorithms, will you choose for $\mathcal{D}$?

Answer: It depends on: (i) how frequently $Search$ is invoked compared to $Insert$ and/or $Delete$, (ii) how much space $\mathcal{D}$ is allowed to take, (iii) how efficiently $Search$ needs to run.

3.1.2 Two Possible Candidates for $\mathcal{D}$

Array, indexed by Student ID:

- Good: $Search$ runs in time $\Theta(1)$.
- Bad: Space inefficient. $10^8 = 100\ million\ slots\ allocated\ to\ store \approx 500\ K\ records.$

Linked List:

- Good: Highly space efficient.
- Bad: $Search$ runs slowly.

3.1.3 Hash Table Objectives

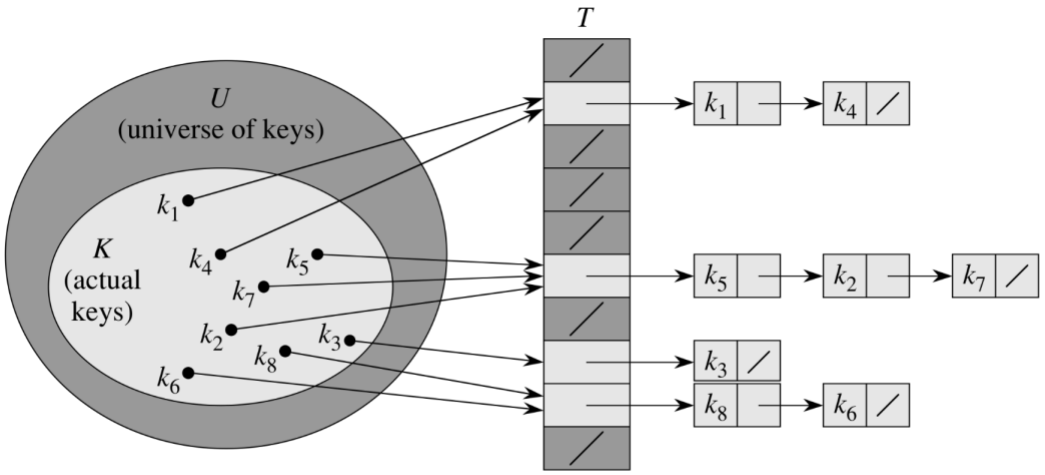
- Time Efficiency: $Search$ to run in time $\approx \Theta(1)$.
- Space Efficiency: If we seek to store n records, space we should have to allocate $\approx \Theta(n)$.

3.1.4 Problem Setup

- We have a universe of keys, U.
- We seek to store n records each with a unique key. Typically, $n \ll |U|$.
- Designer of hash table knows: (i) U, and (ii) n, to some approximation.
- Designer of hash table does not know how the n keys are chosen from U.

3.1.5 How a Hash Table Works

- Designer picks some space allocation, m. Typically, $m = \Theta(n)$. Each $\{0, 1, \dots, m-1\}$ is called a bucket.
- Designer picks a *hash function*, h: $U \to \{0, \dots, m-1\}$.
- Designer picks a strategy for *resolving collisions*.
    - Two distinct keys, $k, l \in U$ are said to collide under h if $h(k) = h(l)$.
        - Collision guaranteed if $|U| > m$.
        - Can be much worse: if $|U| \geq nm$, there exists a set of keys $K \subseteq U$ with $|K| = n$ that all hash to the same bucket, immaterial of choice of h.
    - We consider collision resolution by *chaining* only.
    - If two records with keys k_i, k_j need to exist in the hash table, and $h(k_i) = h(k_j)$, then we maintain a linked list of the records in the bucket $h(k_i)$.



Credit: Cormen et al., "Introduction to Algorithms."

CREATENEW(m)
    Pick a hash function, $h: U \to [0, m)$
    Create a table $T[0, \ldots, m-1]$
    Each $T[i] \leftarrow$ NEWLIST( )
    Store h as part of T
    return T

INSERT(T, r)
    Linked list $L \leftarrow T[h(key(r))]$
    INSERTINTOLIST(L, r)

SEARCH(T, k)
    Linked list $L \leftarrow T[h(k)]$
    return SEARCHINLIST(L, k)

DELETE(T, k)
    Linked list $L \leftarrow T[h(k)]$
    DELETEFROMLIST(L, k)

Choice of h is critical to performance. What is an "ideal" h?

- Can be computed in time constant in n.
- Minimizes collisions. More precisely, acts like a "random function."
    - Suppose for all distinct $k, l \in U$, $\Pr\{h(k) = h(l)\} = 1/m$.
    - Then <u>in expectation</u>, $length\ of\ each\ chain = n/m$.
    - Then, if $m = \Theta(n)$, expected time for $Search$ is $\Theta(1)$.
    - Note: Worst case time for $Search$ remains $\Theta(n)$.

3.1.6 Universal Hashing

Given U and m, construct a set $\mathcal{H}$ of hash functions with the following property:

Given any two distinct $k, l \in U$, the number of hash functions in $\mathcal{H}$ for which k and l hash to the same bucket is $\leq |\mathcal{H}|/m$.

Then, in $CreateNew()$, pick an h from $\mathcal{H}$ uniformly at random.

Claim 1: *Given a universal set of hash functions, $\mathcal{H}$, if we pick $h \in \mathcal{H}$ uniformly at random, then for all distinct $k, l \in U$,* $\Pr\{h(k) = h(l)\} \leq 1/m$.

Proof.

$$\Pr\{h(k) = h(l)\} = \frac{\#\ hash\ functions\ in\ \mathcal{H}\ for\ which\ k\ and\ l\ collide}{total\ \#\ hash\ functions\ in\ \mathcal{H}}$$

$$\Pr\{h(k) = h(l)\} \leq \frac{|\mathcal{H}|/m}{|\mathcal{H}|} = 1/m$$

Challenge: How to construct such an $\mathcal{H}$ given U, m.

3.1.7 How to Construct $\mathcal{H}$ Given U, m

We use Student ID as keys: $U = set\ of\ 8\ digit\ numbers$.

Now construct $\mathcal{H}$ as follows:

- Generate some prime p such that $U \subseteq \{0, 1, \dots, p-1\}$.
- For $a \in \{1, 2, \dots, p-1\}, b \in \{0, 1, \dots, p-1\}$, let $h_{a,b}(k) = \big((ak + b)\ mod\ p\big)\ mod\ m$.
- Adopt $\mathcal{H} = \Big\{h_{a,b} \mid a \in \{1, 2, \dots, p-1\}, b \in \{0, 1, \dots, p-1\}\Big\}$.

Observation right off the bat: $|\mathcal{H}| = p(p-1)$.

For the following claims, adopt:

- $k, l \in \{0, 1, \dots, p-1\}, k \neq l$.
- a, b chosen uniformly (at random) from their respective sets.
- $r = (ak + b)\ mod\ p, s = (al + b)\ mod\ p$.

Claim 2: $k \neq l \rightarrow r \neq s$.

Proof. $r - s \equiv a(k - l)\ (mod\ p)$, and neither a nor $k - l \equiv 0\ (mod\ p)$. And so, $r - s \not\equiv 0\ mod\ p \neq 0$.

Claim 3: *Let:*

$$a, c \in \{1, \dots, p-1\}$$
$$b, d \in \{0, \dots, p-1\}$$
$$r_{a,b} = (ak + b)\ mod\ p$$
$$s_{a,b} = (al + b)\ mod\ p$$

$$r_{c,d} = (ck + d)\ mod\ p$$

$$s_{c,d} = (cl + d)\ mod\ p$$

Then: $\langle a, b\rangle \neq \langle c, d\rangle \Rightarrow \langle r_{a,b}, s_{a,b}\rangle \neq \langle r_{c,d}, s_{c,d}\rangle$

Proof. *

$$a = \Big(\big(r_{a,b} - s_{a,b}\big)((k-l)^{-1}\ mod\ p)\Big)\ mod\ p$$

$$b = \big(r_{a,b} - ak\big)\ mod\ p$$

$$and\ similarly\ for\ c\ and\ d$$

Therefore, $\langle r_{a,b}, s_{a,b}\rangle = \langle r_{c,d}, s_{c,d}\rangle \Rightarrow \langle a, b\rangle = \langle c, d\rangle$, a contradiction.

**Example:* $8a + b = 5$ and $3a + b = 13$

These are both linear equations with unknowns a, b. Now, k, l are between 1 and $p - 1$, and since p is prime, $k - l$ is relatively prime to p, and so its multiplicative inverse exists.

Thus, for distinct $k, l \in U$:

- Each distinct $\langle a, b\rangle$ maps to distinct $\langle r, s\rangle$
- In each such $\langle r, s\rangle, r \neq s$.

Now:

- $\#\ distinct\ \langle a, b\rangle: p(p-1)$.
- $\#\ distinct\ r, s\ with\ r \neq s: p(p-1)$.
- So $\langle a, b\rangle$ to $\langle r, s\rangle$ mapping is one-to-one.
- So, picking $\langle a, b\rangle$ uniformly is equivalent to picking $\langle r, s\rangle$ uniformly.

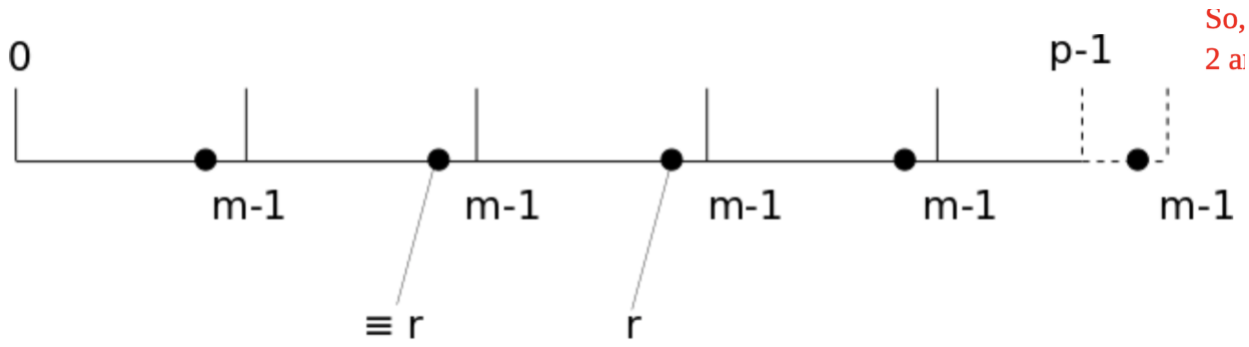
Finally, the $mod\ m$ part:

- Suppose each of a, b is chosen uniformly at random.
- Then, for $k \neq l, \Pr\{h_{a,b}(k) = h_{a,b}(l)\} = \Pr\{r \equiv s\ (mod\ m)\}, r, s$ chosen uniformly.

$$\Pr\{r \equiv s\ (mod\ m)\} = \frac{\#\ items \equiv r\ (mod\ m)\ but \neq r\ in\ \{0, \dots, p-1\}}{\#\ items \neq r\ in\ \{0, \dots, p-1\}}$$

$$\Pr\{r \equiv s\ (mod\ m)\} \leq \frac{\lceil p/m\rceil - 1}{p-1}$$

To intuit the $\lceil p/m\rceil$ term, pictorially:



*In the figure, $\lceil p/m \rceil = 5$. The value r, and every member of $\{0, 1, \dots, p-1\} \neq r$, but $\equiv r \ (mod\ m)$ is shown as a bold dot.

And we know that $\lceil x/y \rceil \leq (x-1)/y + 1$. So:

$$\Pr\{r \equiv s \ (mod\ m)\} \leq \frac{\left((p-1)/m + 1\right) - 1}{p-1} = 1/m$$

*Example: $p = 7, m = 3, r = 5$.

Then, $r \ mod\ m = 5 \ mod\ 3 = 2$.

So, in the set $\{0, \dots, 6\}$, the two numbers, 2 and 5, are congruent modulo 3: $2 \equiv 5 \ mod\ 3$.

3.2 Divide and Conquer

3.2.1 Definition

Divide and Conquer: An algorithm design strategy to which some problems lend themselves. The *divide and conquer* strategy solves a problem by:

1. Breaking it into *subproblems* that are themselves smaller instances of the same type of problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

Example: Compute the product xy given $x, y \in \mathbb{Z}^+$.

Assume our encoding is binary. First split the bits of each x, y into the leading half the bits and trailing half the bits. E.g., $100011 = 100 \circ 011$.*

$$x = x_L \circ x_R = 2^{n/2} x_L + x_R$$

$$y = y_L \circ y_R = 2^{n/2} y_L + y_R$$

Then:

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

$$xy = 2^n x_L y_L + 2^{n/2}\big((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R\big) + x_R y_R$$

**Example:* In base 10, e.g., $153803 = 153 \times 10^{6/2} + 803$

So, to compute xy, where each x, y is n bits:

- Three multiplications of numbers $n/2$ bits long
- Two left bit shifts, one of n bits and the other of $n/2$ bits
- Six additions/subtractions each of n bits

Recurrence for running time, $T(n)$:

$$T(n) = \begin{cases} O(1), if\ n = 1 \\ 3T(n/2) + O(n), otherwise \end{cases}$$

To solve the recurrence, draw a recurrence tree, which is equivalent to inductive "string rewriting" as follows.

First, for the $O(n)$ term, we adopt a canonical function, n, and for $O(1)$, we adopt 1.

$$T(n) = 3T\left(\frac{n}{2}\right) + n$$

$$= 3\left(3T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n = 3^2 \cdot T\left(\frac{n}{2^2}\right) + n + \left(\frac{3}{2}\right)n$$

$$= 3^3 \cdot T\left(\frac{n}{2^3}\right) + n \cdot \left(\left(\frac{3}{2}\right)^0 + \left(\frac{3}{2}\right)^1 + \left(\frac{3}{2}\right)^2\right)$$

$$= 3^4 \cdot T\left(\frac{n}{2^4}\right) + n \cdot \left(\left(\frac{3}{2}\right)^0 + \left(\frac{3}{2}\right)^1 + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3\right)$$

...

$$= 3^{\log_2 n} \cdot 1 + n \cdot \left(\left(\frac{3}{2}\right)^0 + \left(\frac{3}{2}\right)^1 + \cdots + \left(\frac{3}{2}\right)^{\log_2 n - 1}\right) \quad (1, 2, 3)$$

$$= n^{\log_2 3} + 2n\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right) = 3 \cdot n^{\log_2 3} - \frac{2}{n}$$

$$= O(n^{1.59})$$

(1): For what x is $\frac{n}{2^x} = 1$? Answer: $\frac{n}{2^x} = 1 \; iff \; n = 2^x \; iff \log_2 n = x$

(2): $3^{\log_2 n} = z$, $\log_2 z = \log_2 n \times \log_2 3$, then exponentiate by 2: $z = \left(2^{\log_2 n}\right)^{\log_2 3} = n^{\log_2 3}$

(3): $S = a^0 + a^1 + a^2 + \cdots + a^y \rightarrow aS = a^1 + \cdots + a^y + a^{y+1}$

Subtract: $(1 - a)S = a^0 - a^{y+1}$

Implies: $S = \frac{a^0 - a^{y+1}}{1-a}$

3.2.2 Solving Recurrences

Suppose we have the following recurrence for a function $f: \mathbb{N} \rightarrow \mathbb{R}^+$, with constants $a, b, d \in \mathbb{R}, a > 0, b > 1, d \geq 0$:

$$f(n) = \begin{cases} constant, if \; n \; is \; small \\ af(\lceil n/b \rceil) + O(n^d), otherwise \end{cases}$$

Claim 1 (Master Theorem): *Given $f(n)$ as in the recurrence above,*

$$f(n) = \begin{cases} O(n^d), if \; d > \log_b a \\ O(n^d \log n), if \; d = \log_b a \\ O\left(n^{\log_b a}\right), otherwise, i.e., d < \log_b a \end{cases}$$

Proof. Draw a tree and count.

Example 1: Binary search $BinSearch(A[1, \ldots, n], i)$ on array of n items. Running time, $T(n)$, as measured by # comparisons of items in the (sorted) input array and the item we're searching for:

$$T(n) = \begin{cases} 1, if \; n = 1 \\ T(n/2) + 1, otherwise \end{cases}$$

We have $a = 1, b = 2, d = 0$. So, $d = \log_b a$, and $T(n) = O(\log n)$.

Example 2: Recurrence for our algorithm for multiplication.

$$T(n) = \begin{cases} O(1), if \; n = 1 \\ 3T(n/2) + O(n), otherwise \end{cases}$$

We have $a = 3, b = 2, d = 1$. So, $d < \log_b a$, and $T(n) = O\left(n^{\log_2 3}\right)$.

3.2.3 Sorting Problem

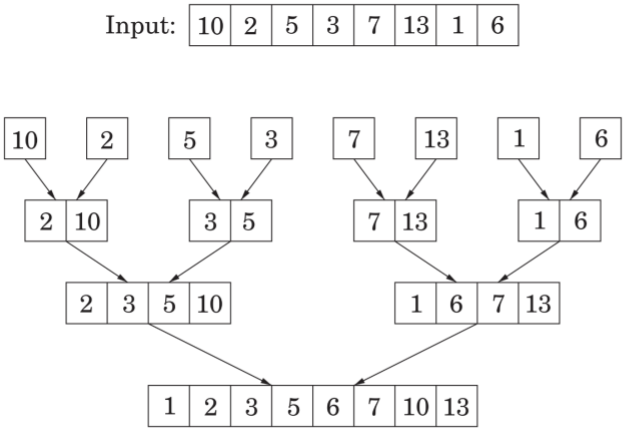
Given as input an array $A[1, \ldots, n]$ whose items are drawn from a totally ordered set, return the sorted permutation of A.

```
function mergesort(a[1...n])
Input: An array of numbers a[1...n]
Output: A sorted version of this array

if n > 1:
   return merge(mergesort(a[1...⌊n/2⌋]),
mergesort(a[⌊n/2⌋ + 1...n]))
else:
   return a
```

```
function merge(x[1...k], y[1...l])
if k = 0: return y[1...l]
if l = 0: return x[1...k]
if x[1] ≤ y[1]:
   return x[1] ∘ merge(x[2...k], y[1...l])
else:
   return y[1] ∘ merge(x[1...k], y[2...l])
```

We claim that a count of this comparison is a meaningful measure of running-time of mergesort

Figure 2.4 The sequence of merge operations in `mergesort`.



Claim 2: *merge is correct.*

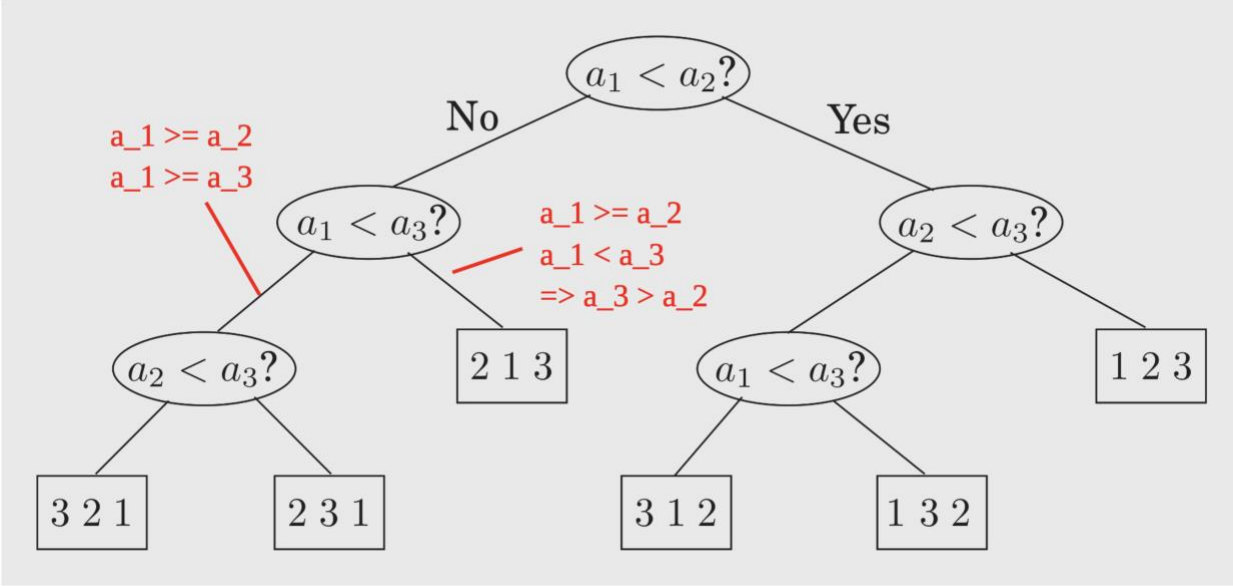
Proof. Induction on # *recursive calls*.

Claim 3: *mergesort is correct.*

Proof. Induction on # *recursive calls*. Appeal to Claim 2.

comparisons of array items: $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n \log n)$.

Claim 4: *mergesort is optimal in the # comparisons: any comparison-based sorting algorithm performs* $\Omega(n \log n)$ *comparisons.*



- What is the most compact way possible to consider every possible permutation? Where "way possible" refers to pairwise comparisons.
- Answer: Binary tree as shown above, with every permutation showing up as a leaf.
- Minimum # comparisons (in worst case) = longest path from root to a leaf = $\log(n!) = \Omega(n \log n)$.
- One of the rare instances in which we have a lower bound for an algorithmic problem.

3.3 Divide and Conquer Applications

3.3.1 Selection Problem

Given as input: (i) an array $S[1, \ldots, n]$ of items drawn from a totally ordered set, and (ii) an integer $k \in \{1, \ldots, n\}$, identify the k^{th} smallest item in S.

E.g., if $k = 1$, we seek the smallest. If $k = n$, we seek the largest. If $k = \lfloor (n+1)/2 \rfloor$, we seek the (left) median.

A divide and conquer strategy:

- Pick some member of S as the *pivot*, v.
    - E.g., randomly from amongst all items in S.
- *Partition* or *split* around that pivot v.
    - Rearrange items in S as follows:

- Move every item $< v$ to the left of v.
- Move every item $> v$ to the right of v.
- Thus, every item whose value is v ends up where it would in a sorted permutation of S.

- Check and recurse.

S :

2	36	5	21	8	13	11	20	5	4	1

is split on $v = 5$, the three subarrays generated are

S_L :

2	4	1

S_v :

5	5

S_R :

36	21	8	13	11	20

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

If v chosen to be, e.g., median, then $T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$.

Challenge is now: How to guarantee a good choice for v?

Suppose "good" v is some value in the middle 50% of S.

- E.g., if $n = 30$, $8 \le rank[v] \le 22$.

Then, recurrence for running time of $selection$:

$$T(n) \le T\left(\frac{3}{4} \cdot n\right) + O(n) = O(n)$$

To pick such a "good" v:

- Pick random v.
- Check if it is good: $time = O(n)$.
- If not good, repeat.

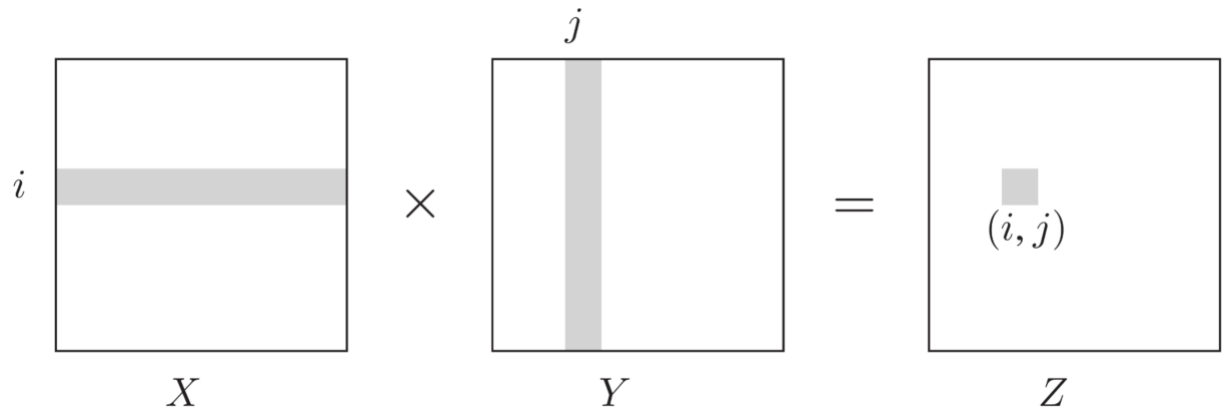
$$\Pr\{we\ pick \text{ "good" v in a trial}\} = \frac{1}{2}$$

So # trials in expectation to get a "good" $v = 2$,

So, we have an algorithm for $selection$, which runs in time $O(n)$ in the expected case.

3.3.2 Matrix Multiplication

Given two $n \times n$ matrices, X, Y, compute the $n \times n$ product $Z = X \cdot Y$.



$$Z = [z_{i,j}] \; where \; z_{i,j} = \sum_{k=1}^{n} x_{i,k} \cdot y_{k,j}$$

Naïve algorithm to compute Z: $\Theta(n^3)$ scalar multiplications and additions.

Smarter algorithm based on divide-n-conquer:

each of A, B, C, D is n/2 x n/2

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

$$= \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$P_1 = A(F - H) \qquad P_5 = (A + D)(E + H)$$
$$P_2 = (A + B)H \qquad P_6 = (B - D)(G + H)$$
$$P_3 = (C + D)E \qquad P_7 = (A - C)(E + F)$$
$$P_4 = D(G - E)$$

Now, recurrence for # scalar multiplications and additions:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$
$$= O\left(n^{\log_2 7}\right) \approx O(n^{2.81})$$

4 DECOMPOSITIONS OF GRAPHS, PATHS

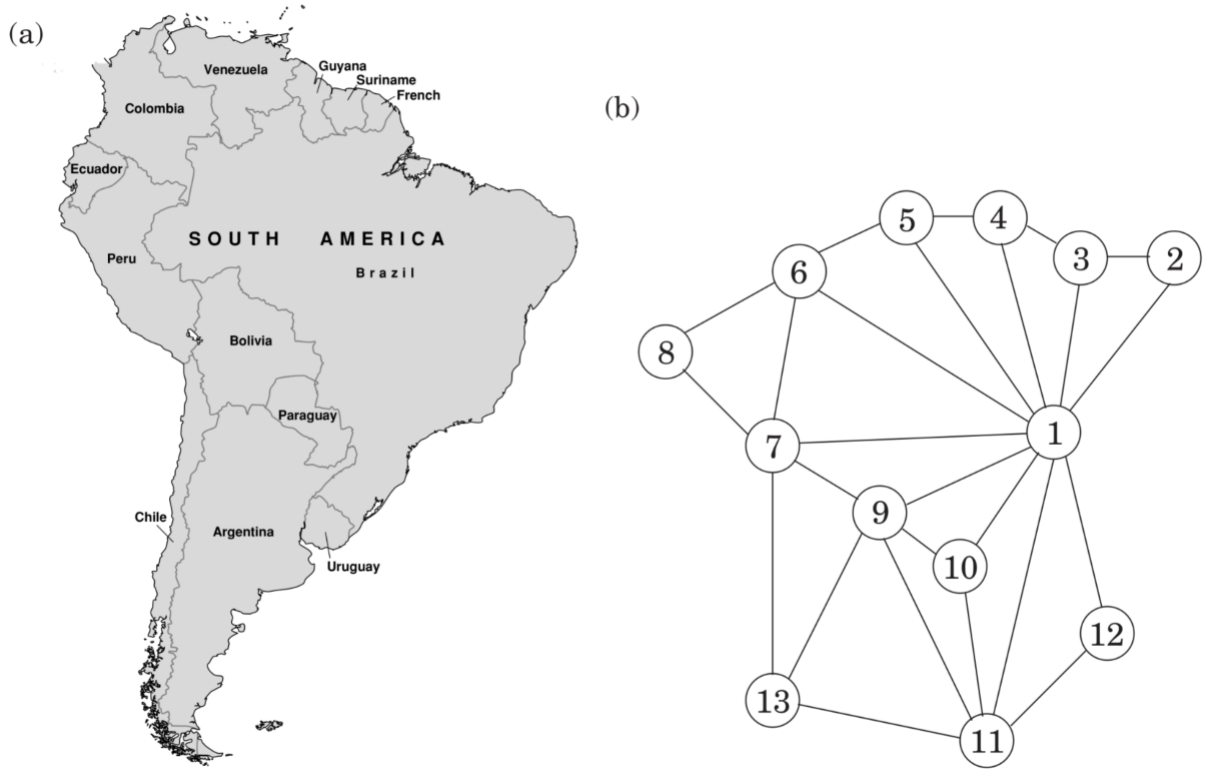
4.1 Graphs

Graph $G = \langle V, E \rangle$. $V = set\ of\ vertices$. $E = set\ of\ edges$.

- Directed: $E \subseteq V \times V$. Written $\langle u, v \rangle\ \epsilon\ E$.
- Undirected: $E \subseteq \{\{u, v\} \mid u, v\ \epsilon\ V, u \neq v\}$. Abuse notation and write $\langle u, v \rangle\ \epsilon\ E$.

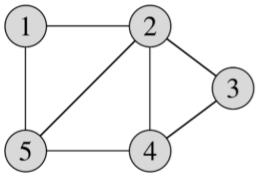
Lots of things are modelled using graphs. E.g., maps, digital circuits, social networks.

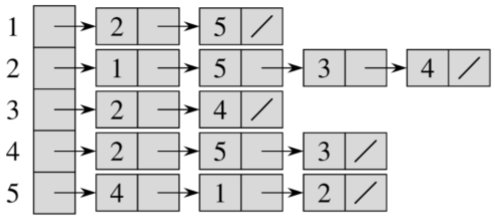
Figure 3.1 (a) A map and (b) its graph.



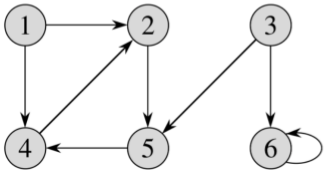
4.1.1 Representing a Graph

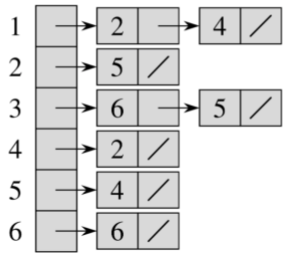
Adjacency list and adjacency matrix.





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

4.1.2 Graph Exploration

An encoding of a graph can be seen as local information, i.e., just information about adjacency.

Algorithmic problems with graphs ask about some global property of the graph.

Given a graph $G = \langle V, E \rangle$ and two distinct vertices $u, v \in V$, is v reachable from u?

An algorithm for this: Depth-First Search (DFS).

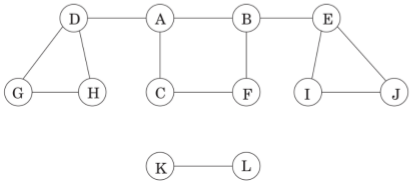
Figure 3.5 Depth-first search.

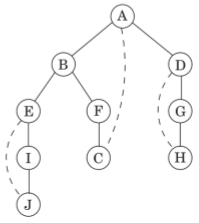
```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v): explore(v)
```

Figure 3.3 Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input: G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to true for all nodes u reachable
   from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u): explore(u)
postvisit(v)
```

Figure 3.4 The result of explore(A) on the graph of Figure 3.2.

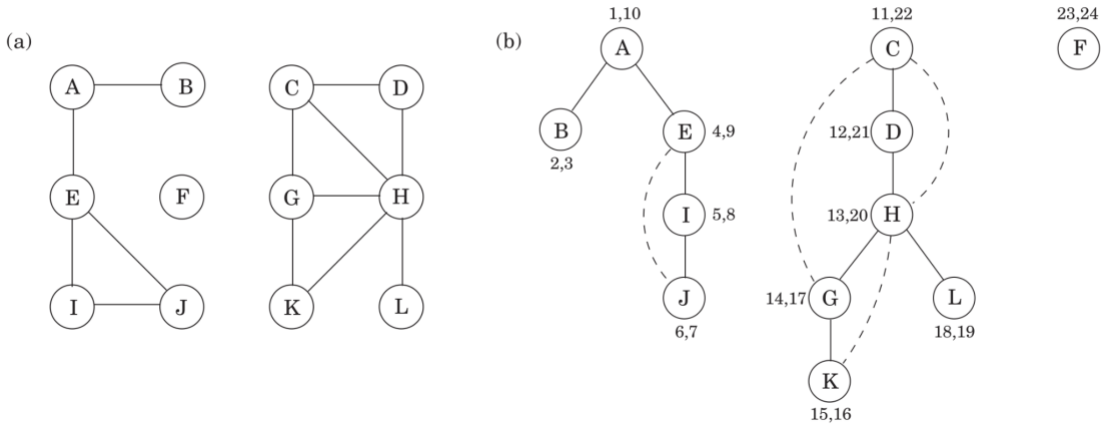




Claim 1: *v is reachable from y if and only if* ***explore***(G, u) *causes v to be* ***visited***.

Time efficiency: $O(|V| + |E|)$.

Figure 3.6 (a) A 12-node graph. (b) DFS search forest.

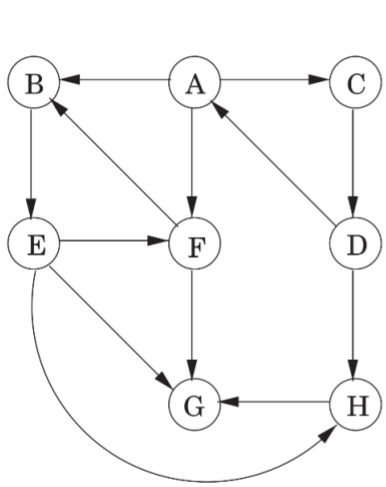


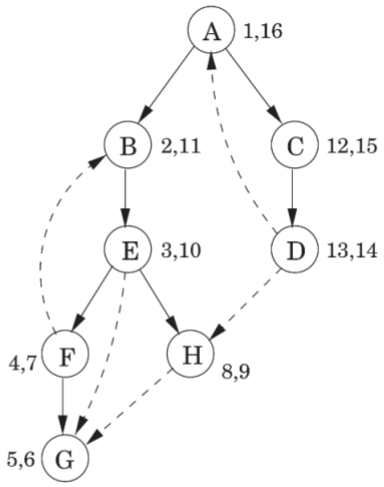

```
procedure previsit(v)
ccnum[v] = cc
```

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

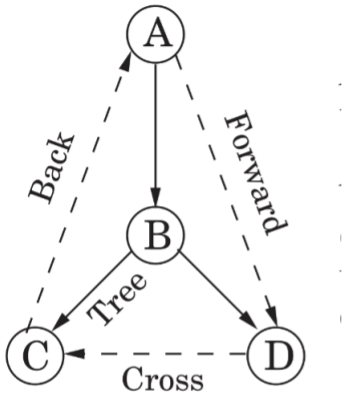
4.2 DFS, Continued

Figure 3.7 DFS on a directed graph.





DFS tree



pre/post ordering for (u, v)	Edge type
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

Specifically: if <u,v> is an edge, then [u [v u] v] cannot happen.

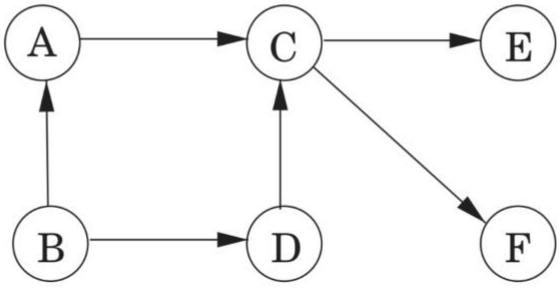
(a) (b) (c) (d)

(a): explore(u) pushed onto call stack
(b): explore(v) pushed, note: explore(u) is lower in the stack, unpopped
(c): explore(u) popped off call stack
(d): explore(v) popped off call stack

4.2.1 Directed Acyclic Graph (DAG)

Linearization or topological sort: of a DAG is a left-to-right ordering of all the vertices, $\langle u_0, u_1, \dots, u_{|V|-1} \rangle$ such that $\langle u_i, u_j \rangle \epsilon E \Longrightarrow j > i$, i.e., all edges go left to right.

Figure 3.8 A directed acyclic graph with one source, two sinks, and four possible linearizations.



A linearization: B, D, A, C, F, E.

Claim 1: *If G is a DAG, then in a run of $dfs(G)$, every edge leads to a vertex with a lower (i.e., smaller-valued)* **post** *number.*

Equivalent to: if G is a DAG, then in any run of $dfs(G)$, no edge is a back edge.

An algorithm for linearization:

Run $\boldsymbol{dfs}(G)$

When we are done with a vertex, insert it at the beginning.

Claim 2: *Directed G is acyclic if and only if it is linearizable if and only if $\boldsymbol{dfs}(G)$ results in no back edges.*

Another algorithm:

Find a source, output it, and delete it from the graph.

Repeat until the graph is empty.

Claim: This algorithm runs in time $O(|V| + |E|)$ as well.

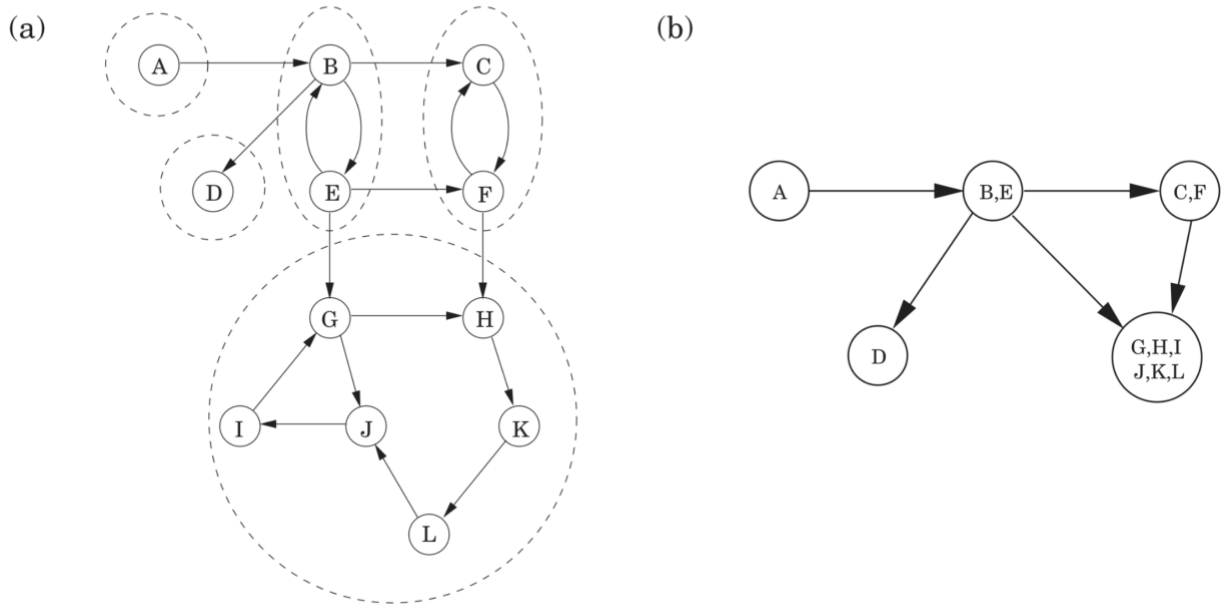
4.2.2 Strongly Connected Components

Given directed $G, u, v \epsilon G$ with $u \neq v$ are said to be strongly connected if $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Given directed $G = \langle V, E \rangle$, the set of Strongly Connected Components $\{V_0, V_1, \dots, V_{k-1}\}$ is a partition of V with the property: for all $i = 0, \dots, k-1$, distinct $u, v \epsilon V_i \Leftrightarrow u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Recall: A partition of a set $S, \{S_0, \ldots, S_{n-1}\}$ has the properties: (i) for all $i = 0, \ldots, n-1, S_i \subseteq S$, and (ii) for all distinct $i, j = 0, \ldots, n-1, S_i \cap S_j = \emptyset$, and (iii) $S_0 \cup S_1 \cup \ldots \cup S_{n-1} = S$.

Figure 3.9 (a) A directed graph and its strongly connected components. (b) The meta-graph.

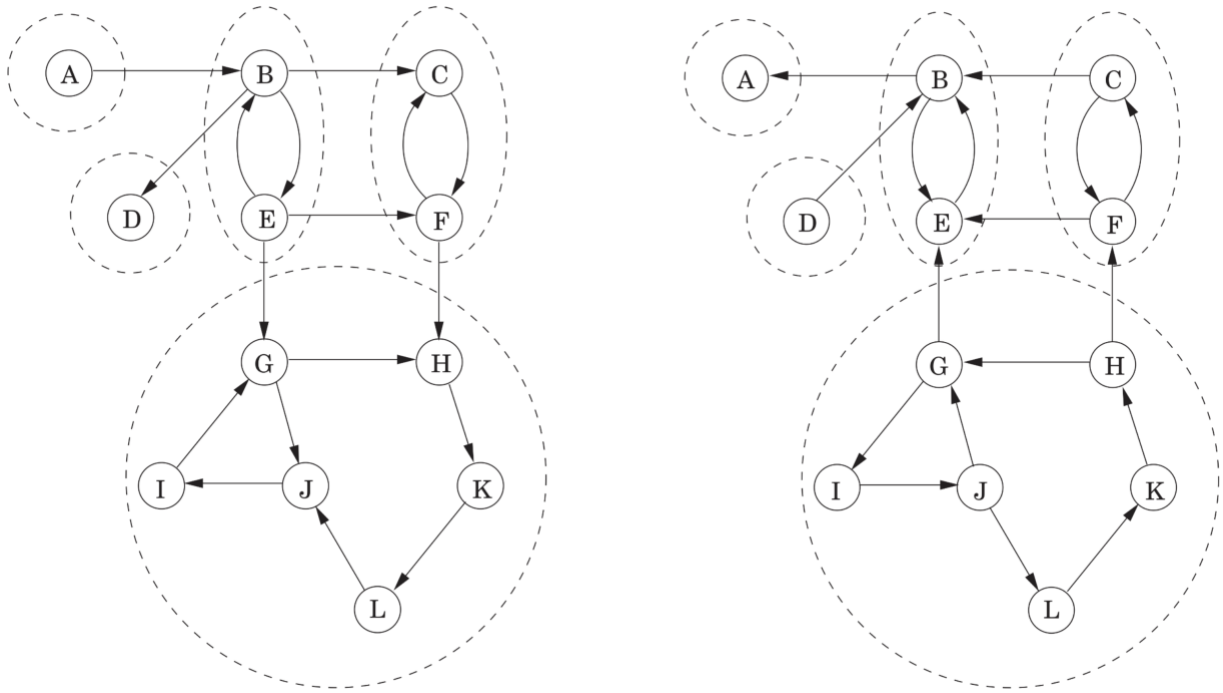


Claim 3: *The meta-graph is* (i) *unique, and* (ii) *acyclic.*

The algorithm for computing the meta-graph:

1. *Run* ***dfs*** *on* G^R.
2. *Run the undirected connected components algorithms on* G*, and during the depth-first search, process the vertices in decreasing order of their* ***post*** *numbers from Step (1).*

Where G^R is the *transpose* or *reverse* graph of G.



Claim 4: *G^R can be computed from $G = \langle V, E \rangle$ in time $O(|V| + |E|)$.*

Claim 5: *C is a SCC in G if and only if C is a SCC in G^R.*

Definition 1: *In a directed graph $G = \langle V, E \rangle, u \epsilon V$ is called a source if there is no edge that is incident on u. A node $v \epsilon V$ is called a sink if there is no edge that leaves v.*

Claim 6: *A node is a source in G if and only if it is a sink in G^R. A node is a sink in G if and only if it is a source in G^R.*

Proof. Exploit the sub-claim: $(G^R)^R = G$.

Claim 7: *A node that receives the highest **post** number in a run of **dfs** on a directed graph F lies in a source in SCC of the meta-graph.*

- And therefore, a sink SCC of F^R.

Claim 8: *If C, C' are distinct SCCs in F, and there's an edge $u \to v$ where $u \epsilon C, v \epsilon C'$, then the highest **post** number in C is bigger than the highest **post** number in C' in any run of **dfs** on F.*

- A generalization of Claim (7) above.

Step (1) of the algorithm on G^R example above can result in: $G, I, J, L, K, H, D, C, F, B, E, A$.

Then, Step (2) may group, in sequence: $(G, H, I, J, K, L), (D), (C, F), (B, E), (A)$.

Sanity-check question: if that was the sequence of **post** numbers in Step (1), which vertex happened to have been chosen first in $\boldsymbol{dfs}$? More generally, what was the order of vertices chosen in $\boldsymbol{dfs}$?

- Solution: Work backwards from the last one, i.e., A.
- First vertex chosen was A. Then B. Then C. Then D. Finally, G.

4.3 Shortest Paths

- Another "global" property in the context of a graph.
- A path (of length k) in $G = \langle V, E \rangle$ is a sequence $\langle u_0, u_1, \dots, u_k \rangle$ where $\langle u_i, u_{i+1} \rangle \epsilon E$ for all $i = 0, \dots, k-1$. A path is simple if all vertices in it are distinct.
- A shortest path from vertex u to v is one of minimum length across all paths $u \rightsquigarrow v$. We call the length of the shortest path $u \rightsquigarrow v$ the distance from u to v.
- A shortest path $u \rightsquigarrow v$ is not necessarily unique. The length of a shortest path $u \rightsquigarrow v$ is.

Claim 1: *If a path $u \rightsquigarrow v$ exists, then a shortest path $u \rightsquigarrow v$ exists. If a shortest path $u \rightsquigarrow v$ exists, it must be simple*.

4.3.1 Breadth First Search (BFS)

- An algorithm to compute distances and shortest paths from a source vertex.
- Mindset: Think of edges as cloth strings of equal length. A BFS tree results from picking the graph up by the source vertex.

Figure 4.2 A physical model of a graph.

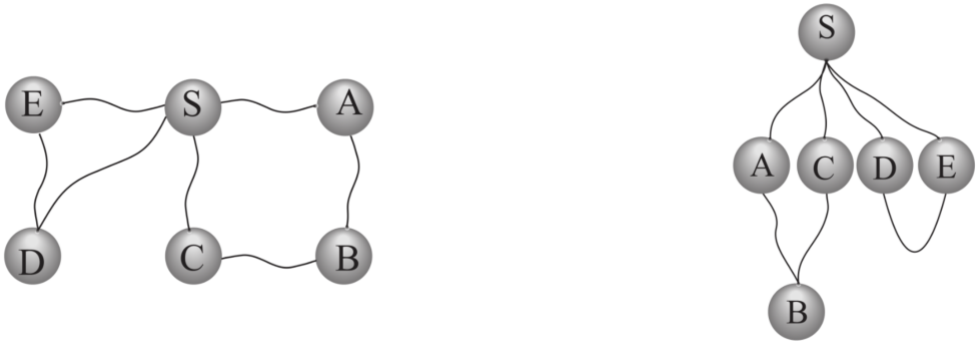


Figure 4.3 Breadth-first search.

```
procedure bfs(G, s)
Input: Graph G = (V, E), directed or undirected; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) ∈ E:
        if dist(v) = ∞:
            inject(Q, v)
            dist(v) = dist(u) + 1
```

Figure 4.1 (a) A simple graph and (b) its depth-first search tree.

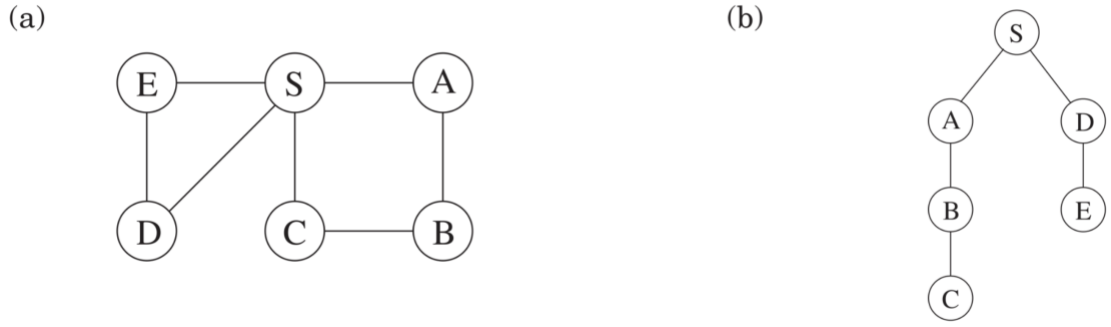
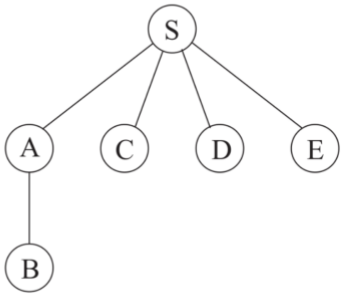


Figure 4.4 The result of breadth-first search on the graph of Figure 4.1.

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



Correctness of BFS:

Claim 2: *For every* $v \in V$, $\boldsymbol{bfs}(G, s)$ *results in the correct value for* $\boldsymbol{dist}(v)$.

Proof. Two cases.

(i) No path $s \rightsquigarrow v$ exists. Then, for BFS to be correct for that particular v, $\boldsymbol{dist}(v) = \infty$ at the end of the run of the algorithm.

(ii) If a path $s \rightsquigarrow v$ exists, then let $d \in \mathbb{Z}_0^+$ be the distance from s to v. Then prove by induction on d that $\boldsymbol{dist}(v) = d$. Use the fact that $\boldsymbol{dist}(v)$, once initialized, is then set only when v is enqueued, and never again thereafter.

Time efficiency of BFS: $O(|V| + |E|)$.

Unlike the manner in which we designed DFS, not every vertex and edge is necessarily visited by BFS. So, its time efficiency is $\Theta(|V| + |E|)$ in the worst case only.

5 PATHS IN GRAPHS, CONTINUED

5.1 Shortest Paths in Weighted Graphs

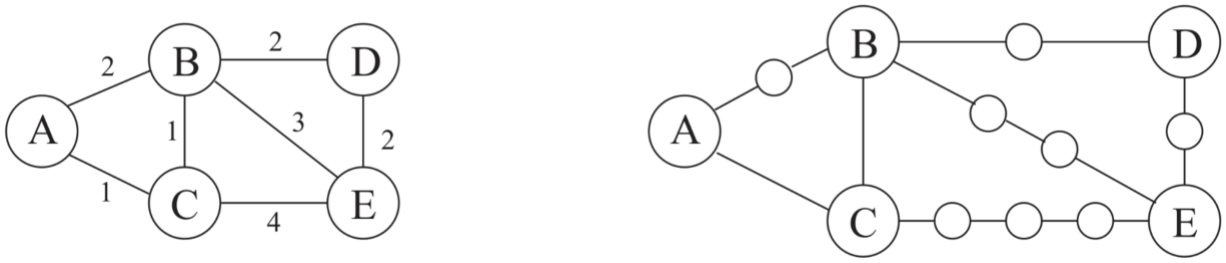
We now assume that a graph is a triple $G = \langle V, E, l \rangle$, where $l: E \to \mathbb{R}$.

Suppose $l_e \in \mathbb{Z}^+$ for all $e \in E$. Then, a candidate algorithm:

Introduce $l_e - 1$ "dummy" vertices on edge e.

*Run **bfs** on the resultant graph.*

Figure 4.6 Breaking edges into unit-length pieces.



Algorithm is correct, but inefficient. How inefficient?

Answer: In the worst case, algorithm runs in exponential time, i.e., in time exponential in the size of the input.

In binary encoding, each l_e can be encoded with $\Theta(\log l_e)$ bits only.

Better adaptation of BFS: Dijkstra's algorithm.

5.1.1 Dijkstra's Algorithm

An example of a greedy algorithm: Locally optimal choices result in global optimum.

Caution 1: Not every (optimization) problem possesses such a greedy choice property.

Caution 2: Not necessarily easy to identify such a greedy choice property even if problem possesses one.

Caution 3: Dijkstra's guaranteed to be correct if $l_e \in \mathbb{R}_0^+$ for all $e \in E$.

Dijkstra's assumes access to a priority queue data structure. API:

- $makequeue$: Make a priority queue with a given set of values. May be realized as repeated calls to an $insert$ algorithm.

- *deletemin*: Remove a queue item of smallest value and return it.
- *decreasekey*: Decrease the value of an item in the queue.

Figure 4.8 Dijkstra's shortest-path algorithm.

```
procedure dijkstra(G, l, s)
Input:  Graph G = (V, E), directed or undirected;
        positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
       if dist(v) > dist(u) + l(u, v):
          dist(v) = dist(u) + l(u, v)
          prev(v) = u
          decreasekey(H, v)
```

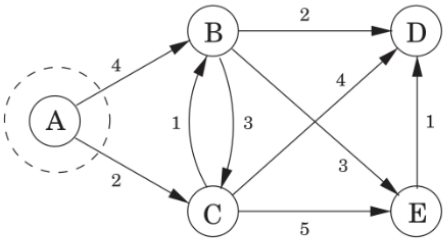
really, the dist values as the priority

"update" or "relaxation" of an edge <u,v>

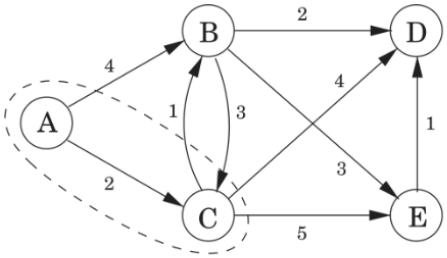
Semantics/properties of "∞":

- $\infty = \infty$. Or $a = \infty$ and $b = \infty \Longrightarrow a = b$.
- For any $r \epsilon \mathbb{R}, r < \infty$.
- For any $r \epsilon \mathbb{R}, r + \infty = \infty + r = \infty + \infty = \infty$.

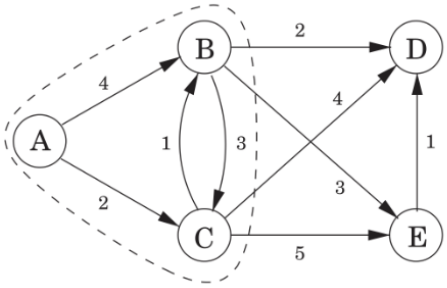
Figure 4.9 A complete run of Dijkstra's algorithm, with node A as the starting point. Also shown are the associated `dist` values and the final shortest-path tree.



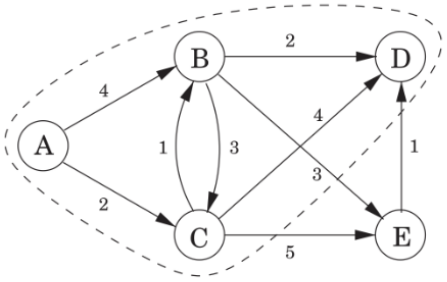
A: 0	D: ∞
B: 4	E: ∞
C: 2	



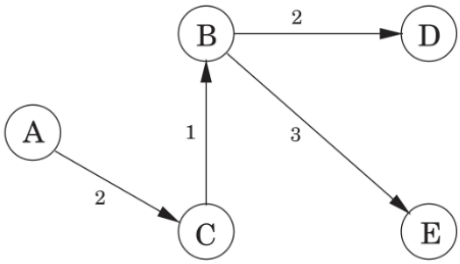
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



5.1.2 Correctness

A partial proof. First, denote as $\delta(x, y)$ the shortest distance from vertex x to y.
We prove:

> *For every vertex u that is reachable from s, at the moment u is extracted from the priority queue, it is the case that $\boldsymbol{dist}(u) = \delta(s, u)$.*

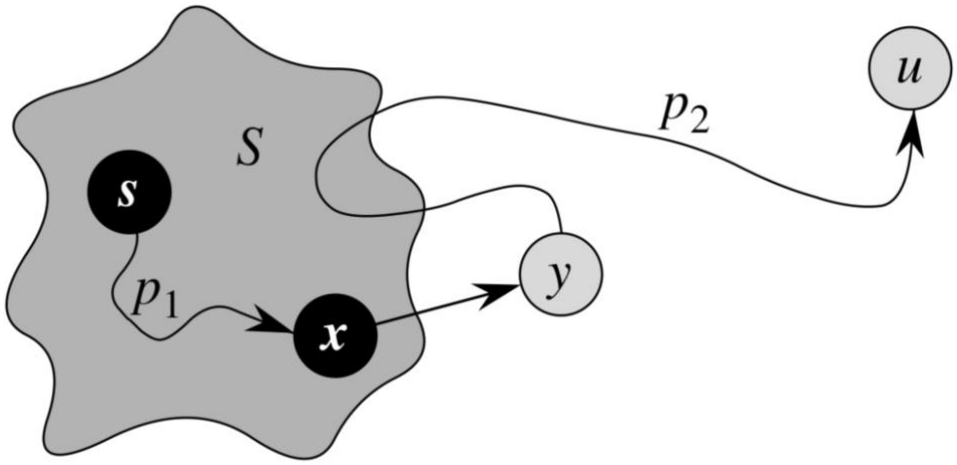
Claim 1: If $a \rightsquigarrow x \rightsquigarrow y \rightsquigarrow b$ *is a shortest path from a to b, then the x to y subpath is a shortest path from x to y.*
Proof. By contradiction.

Claim 2: *Suppose $s \rightsquigarrow x \rightarrow y$ is a shortest path from s to y, where $x \rightarrow y$ is an edge. Also suppose that we are at a moment that $\boldsymbol{dist}(x) = \delta(s, x)$. And we update $\boldsymbol{dist}(y)$ based on $l(x, y)$ as in Dijkstra's algorithm. Then, immediately after, $\boldsymbol{dist}(y) = \delta(s, y)$.*
Proof. By Claim (1), each of the subpaths $s \rightsquigarrow x$ and $x \rightarrow y$ is a shortest path. Therefore, $\delta(s, y) = \delta(s, x) + l(x, y)$.

Claim 3: *Once we initialize $\boldsymbol{dist}(u)$ to ∞, it is always the case that $\delta(s, u) \leq \boldsymbol{dist}(u)$.*

Note 1: Claims (1)-(3) hold even if we allow negative edge weights.
Note 2: Claim (1) refers to a kind of *optimal substructure*.

Now the proof for the claim about Dijkstra's:



Consider the moment in a run of Dijkstra's algorithm as shown above.

- The set S comprises those vertices that have already been extracted from the priority queue. At the minimum, $S = \{s\}$, but may contain additional vertices. We assume that for every vertex in S, its $dist$ value is indeed correct, i.e., equal to its $\delta(s,\cdot)$ value.
- We are about to extract the vertex u from the priority queue, i.e., its $dist$ value is the minimum in the priority queue at this moment. For the purpose of contradiction, assume that $dist(u) \neq \delta(s,u)$. That is, u is the first vertex in this run of Dijkstra's for which we have an error.
- Assume that the $s \dashrightarrow u$ path shown in the picture is a shortest path from s to u. That path can be decomposed into: $s \overset{p_1}{\dashrightarrow} x \rightarrow y \overset{p_2}{\dashrightarrow} u$, where: (i) all the vertices of p_1 are $\in S$, (ii) y is the first vertex on the $s \dashrightarrow u$ path that $\notin S$, (iii) we know $s \neq u$, and (iv) it is possible that $s = x$ and/or $y = u$.

We now derive a contradiction to the claim that $dist(u) \neq \delta(s,u)$.

- $dist(y) = \delta(s,y)$ – by Claim (2) above.
- $\delta(s,y) \leq \delta(s,u)$ – by Claim (1) above, and all edge weights are non-negative.
- $\delta(s,u) \leq dist(u)$ – by Claim (3) above.
- $dist(u) \leq dist(y)$ – we are about to extract u from the priority queue.

So, we have:

$$dist(y) = \delta(s,y) \leq \delta(s,u) \leq dist(u) \leq dist(y)^*$$
$$\Longrightarrow dist(y) = \delta(s,y) = \delta(s,u) = dist(u)$$

$* \; a \leq b \leq c \leq d \leq a \rightarrow a = b = c = d$

We have attained our desired contradiction.

5.1.3 Time Efficiency of Dijkstra

Implementation	`deletemin`	`insert/ decreasekey`	$\|V\| \times$ `deletemin` + $(\|V\| + \|E\|) \times$ `insert` or decreasekey
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$O\left(\frac{d \log \|V\|}{\log d}\right)$	$O\left(\frac{\log \|V\|}{\log d}\right)$	$O\left((\|V\| \cdot d + \|E\|) \frac{\log \|V\|}{\log d}\right)$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ (amortized)	$O(\|V\| \log \|V\| + \|E\|)$

Array implementation:

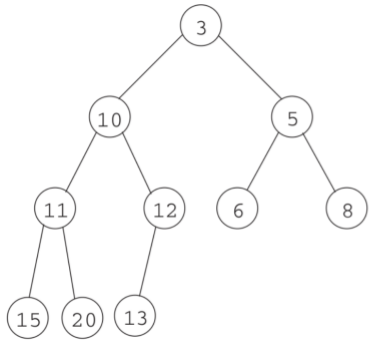
makequeue(S)
  $A \leftarrow$ new array of values from S
  return A

decreasekey$(A, i, newval)$
  $A[i] \leftarrow newval$

deletemin(A)
  $i \leftarrow$ index of minimum value in A
  $min \leftarrow A[i]$, $A[i] \leftarrow \infty$
  if $min = \infty$ **then return** NIL
  else return min, i

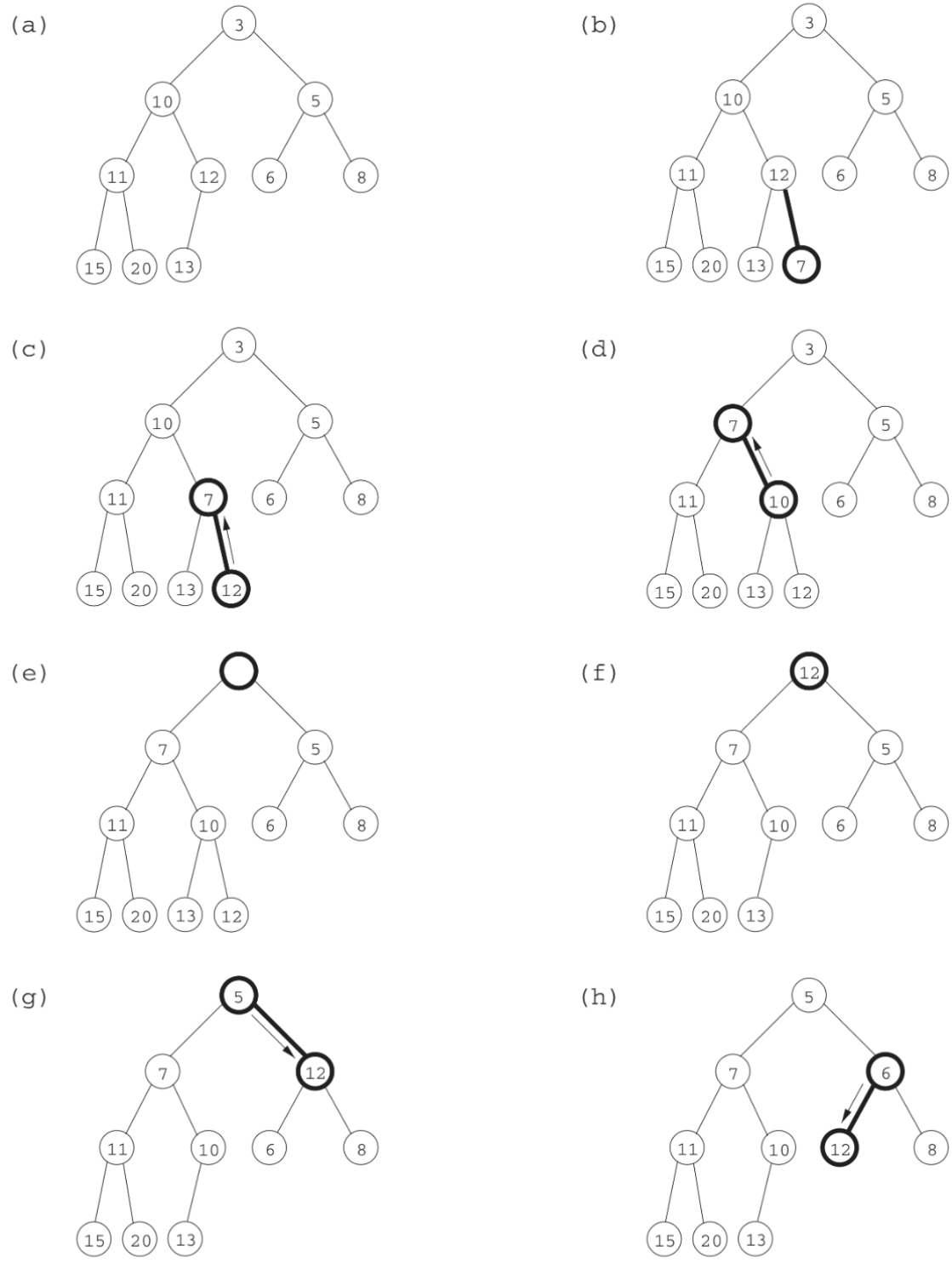
Binary heap:

1. It is a complete binary tree – every level filled from left to right and must be full before next level is started.
2. Key value of any node is $\leq$ the key values of its children.
3. Realization as array $A[1, \ldots, n]$: root at index 1, left child of a node at index i at index $2i$, right child at index $2i + 1$. Parent of node at index j is at index: $\left\lfloor \frac{j}{2} \right\rfloor$.



3	10	5	11	12	6	8	15	20	13

Figure 4.11 (a) A binary heap with 10 elements. Only the key values are shown. (b)–(d) The intermediate "bubble-up" steps in inserting an element with key 7. (e)–(g) The "sift-down" steps in a delete-min operation.



5.2 Shortest Paths and Negative Edge Weights

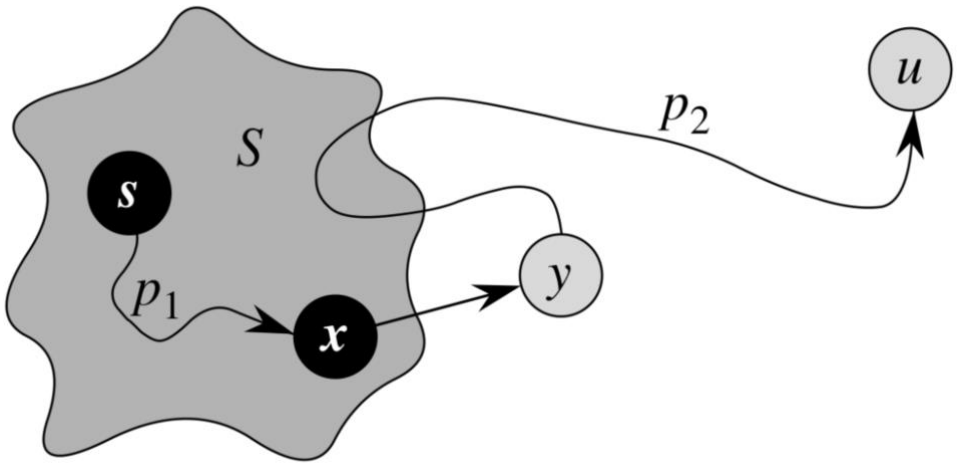
Dijkstra's not guaranteed to be correct. If we have a negative weight cycle reachable from source vertex, then notion of shortest path (distance) not well defined.

So, we assume graph has no negative weight cycles (reachable from s).

Claim 1: *In a graph that has no negative weight cycles, if there exists a path $u \rightsquigarrow v$, there exists a shortest path $u \rightsquigarrow v$ that is simple.*

Proof by construction: Suppose we are given a shortest path from u to v that contains a cycle. Just remove the cycle and now we have a path of weight at most the original.

Where proof for correctness of Dijkstra falls apart with negative weight edges:



- Given that the $s \rightsquigarrow u$ path in the picture is a shortest path, we know that the $s \rightsquigarrow y$ subpath is a shortest path.
- <u>But</u> that does not imply $\delta(s, y) \leq \delta(s, u)$ unless edge weights are ≥ 0.
    - E.g., p_2 is the single edge $y \rightarrow u$ with $l(y, u) = -1$.
    - Then, $\delta(s, u) = \delta(s, y) - 1$.

Key operation in Dijkstra's:

$$\underline{\texttt{procedure update}}((u, v) \in E)$$
$$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + l(u, v)\}$$

Claim 2: *Suppose* $\langle u_0, u_1, \ldots, u_k \rangle$ *is a shortest path* $u_o \rightsquigarrow u_k$. *Then, if we initialize* $dist(u_0) = 0$ *and for all* $i = 1, \ldots, k$, $dist(u_i) = \infty$, *and we perform at least one* $update$ *on every edge* $\langle u_i, u_{i+1} \rangle$ *in that shortest path in order, then afterwards,* $dist(u_k) = \delta(u_0, u_k)$.

Proof. By induction on k. Exploit Claim (2) from Lecture Notes 5(a).

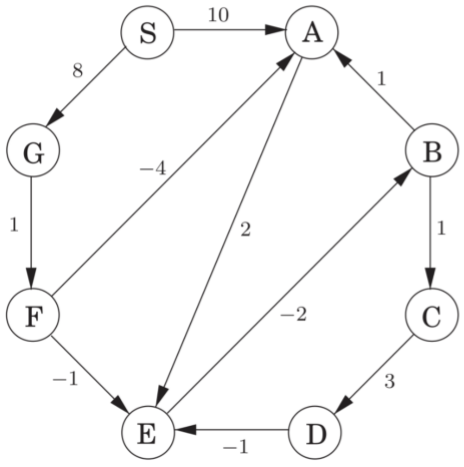
5.2.1 Bellman-Ford Algorithm

Figure 4.13 The Bellman-Ford algorithm for single-source shortest paths in general graphs.

```
procedure shortest-paths(G, l, s)
Input: Directed graph G = (V, E);
       edge lengths {l_e : e ∈ E} with no negative cycles;
       vertex s ∈ V
Output:For all vertices u reachable from s, dist(u) is set
       to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```

max # edges in any simple path

Figure 4.14 The Bellman-Ford algorithm illustrated on a sample graph.



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Time efficiency of Bellman-Ford: $\Theta(|V| \cdot |E|)$.

- $\Omega(|V|)$: Best case elicited by $E = \emptyset$ and $|E| = \Theta(1)$.
- $O(|V|^3)$: Worst case elicited by $|E| = \Theta(|V|^2)$.

5.2.2 Directed Acyclic Graphs (DAGs)

Shortest paths (distances) in DAGs:

After we linearize, all paths run left to right. Exploit Claim (2) above.

Figure 4.15 A single-source shortest-path algorithm for directed acyclic graphs.

```
procedure dag-shortest-paths(G, l, s)
Input:  Dag G = (V, E);
        edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
   for all edges (u, v) ∈ E:
      update(u, v)
```

6 GREEDY ALGORITHMS

6.1 Greedy Algorithms

Greedy Algorithms: Exist only if the problem possesses the greedy choice property.

Optimization Problem: Longest/shortest/cheapest/largest/smallest/etc.

- E.g., shortest paths in a graph.

Greedy Choice Property: Property that some (and only some) optimization problems possess.

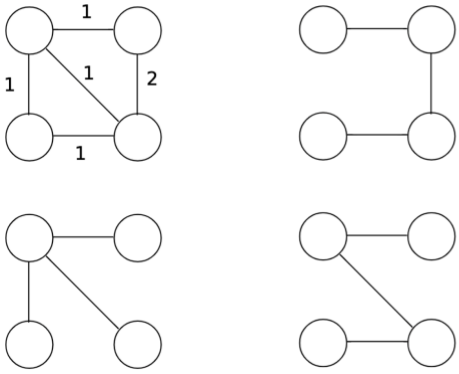
There exists a sequence of locally optimal choices that leads to a global optimum.

- E.g., shortest paths in a graph in which edge weights are non-negative.

6.1.1 Minimum Spanning Trees

- Tree: An acyclic connected undirected graph.
- Spanning tree of a connected undirected graph $G = \langle V, E \rangle$: A subgraph $T = \langle V_T, E_T \rangle$ of G where: (i) $V_T = V$, (ii) $E_T \subseteq E$, and (iii) T is a tree.
- Minimum spanning tree of an edge weighted connected undirected graph $G = \langle V, E, l \rangle$: A spanning tree $T = \langle V, E_T \rangle$ of G with the property:

  Given any spanning tree $T' = \langle V, E_{T'} \rangle$ of G, $\sum_{e \in E_T} l(e) \leq \sum_{e \in E_{T'}} l(e)$



Top left: a connected, edge-weighted graph G.
Top right: a spanning tree of G, but not an MST.
Bottom left and right: two MSTs of G.

Efficient algorithm for a spanning tree: BFS. Not correct if we seek MST.

An algorithm for MST: Kruskal's.

Perceive an MST of $G = \langle V, E \rangle$ as a set of edges, $E_T \subseteq E$

Start with $E_T \leftarrow \emptyset$

Sort edges in non-decreasing order of weight

For each edge in sorted order, add it to E_T provided this does not result in a cycle

Figure 5.1 The minimum spanning tree found by Kruskal's algorithm.

6.1.2 Correctness of Kruskal

Kruskal is an instantiation of a more general approach. First some definitions.

Definition 1. *We define cut, and what it means for an edge to cross a cut.*

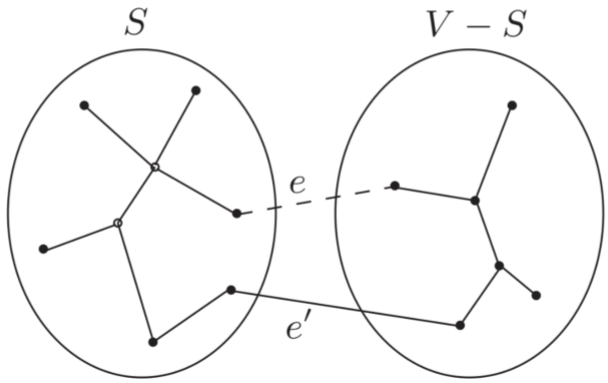
- *A cut of a graph $G = \langle V, E \rangle$ is a partition of V into two sets, $\langle S, V \setminus S \rangle$.*
- *An edge $\langle u, v \rangle$ is said to cross a cut $\langle S, V \setminus S \rangle$ if one of the vertices u, v is in S and the other is in $V \setminus S$. Otherwise, we say that the cut respects the edge.*
- *We say that a cut respects a set of edges X if no edge in X crosses the cut.*
- *An edge is said to be a light edge that crosses a cut if amongst all edges that cross the cut, the edge has the smallest weight.*

Now the general approach:

> *Suppose edges X are part of some MST T of $G = \langle V, E, l \rangle$. Pick a cut $\langle S, V \setminus S \rangle$ that respects X. Let e be light edge that crosses the cut. Then $X \cup \{e\}$ is part of some MST T'.*

Why is the above approach correct?

Figure 5.2 $T \cup \{e\}$. The addition of e (dotted) to T (solid lines) produces a cycle. This cycle must contain at least one other edge, shown here as e', across the cut $(S, V - S)$.



- $l(T) + l(e) - l(e') \leq l(T)$ – e is a light edge that crosses the cut.
    - So, the only issue: is $T \cup \{e\} \setminus \{e'\}$ a tree?
- $T \cup \{e\} \setminus e'$ has exactly $|V| - 1$ edges.
- It is acyclic because:
    - Removing e' splits T into two connected components.
    - Then adding e reconnects these components.

In what wat is Kruskal an instantiation of a general approach?

- Kruskal starts out with each vertex in its own connected component.
- In any step, Kruskal connects two distinct connected components, C, C'.
- Kruskal picks a light edge that crosses the cut $\langle C, V \setminus C \rangle$.

6.1.3 Time Efficiency of Kruskal

Need to write it down in more detail.

Figure 5.4 Kruskal's minimum spanning tree algorithm.

```
procedure kruskal (G, w)
Input:  A connected undirected graph G = (V, E) with edge weights w_e
output: A minimum spanning three defined by the edges X

for all u ∈ V:
   makeset (u)

X = {}
sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

Relies on a *union-find* data structure. API:

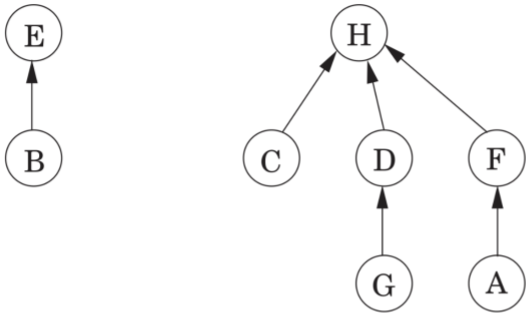
- $makeset(x)$: Creates a singleton set that contains x.
- $find(x)$: Return a unique identifier for the set that contains x. So, $find(x) = find(y)$ if and only if x and y are members of the same set.
- $union(x, y)$: Merge (i.e., union) the sets that contain x and y. So, henceforth, $find(x) = find(y)$.

$$Time\ efficiency\ of\ Kruskal = sort + |V| \times makeset + 2|E| \times find + (|V| - 1) \times union$$

A realization of a union-find data structure: union by rank.

- Maintain a tree of nodes for each disjoint set.
- Each node maintains two attributes:
    - *Its rank* – distance from it to a farthest leaf.
    - *Its parent* in the tree, $\pi(\cdot)$. For the root, r, $\pi(r) = r$, i.e., self-loop.

Figure 5.5 A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.



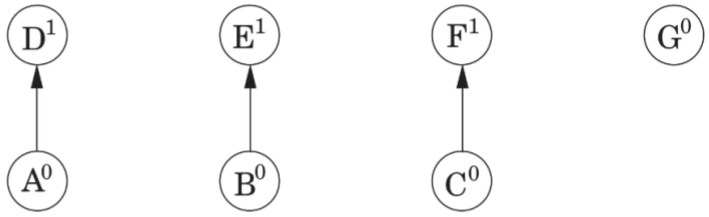

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x): x = π(x)
return x
```

```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y: return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y): rank(r_y) = rank(r_y) + 1
```

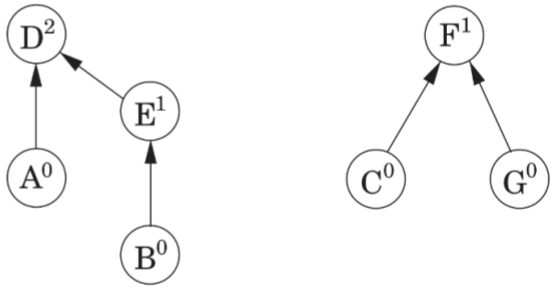
Figure 5.6 A sequence of disjoint-set operations. Superscripts denote rank.

After `makeset`(A),`makeset`(B),...,`makeset`(G):

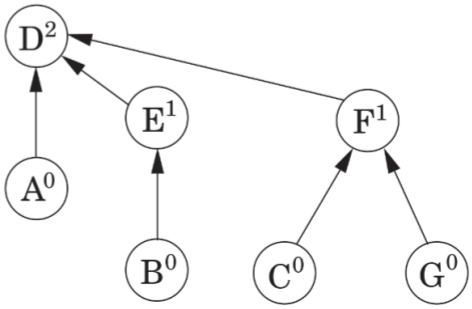
After `union`(A,D),`union`(B,E),`union`(C,F):



After `union`(C,G),`union`(E,A):



After `union`(B,G):



Time efficiency?

Claim 1. *In a union by rank tree every subtree rooted at a node of rank k has $\geq 2^k$ nodes.*

Proof. By induction on k. True if $k = 0$.

If $k = 1$, the tree rooted at the node is the result of a $union$. Node has at least one child.

If $k \geq 2$, consider the last $union$ operation of trees whose roots had rank $< k$. Must have been of two trees each of whose roots had rank $k - 1$. Those two trees each had $\geq 2^{k-1}$ nodes by the induction assumption. So, tree after $union$ has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

Claim 2. *The height (longest path from root to a leaf) in a union by rank tree is* $O(\log n)$.

So, time efficiencies of union-find as realized by union by rank:

- $makeset$: $O(1)$.
- $find, union$: $O(\log n)$.

So, time efficiency of Kruskal:

- First observe because $|V| - 1 \leq |E| \leq \binom{|V|}{2}$, $\log|E| = O(\log|V|)$.
- So, $sort = O(|E| \log|V|)$.
- $|V| \times makeset = O(|V|)$.
- $2|E| \times find = O(|E| \log|V|)$.
- $(|V| - 1) \times union = O(|V| \log|V|)$.

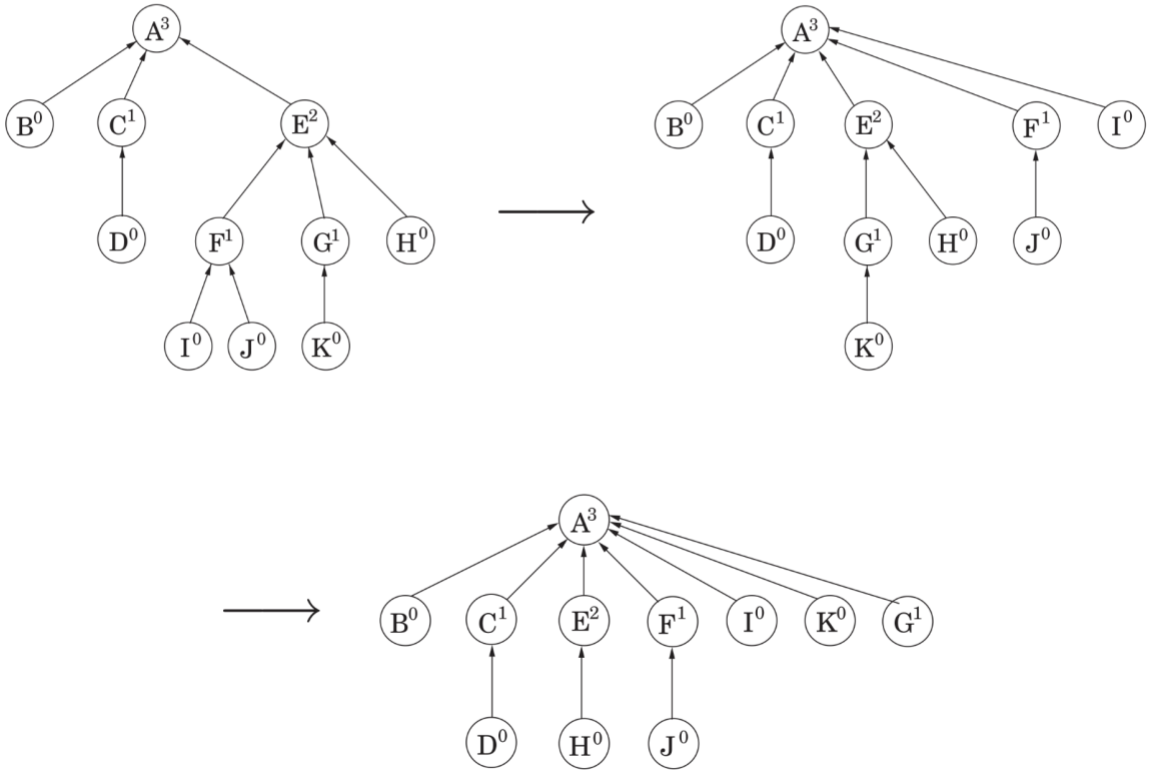
$Total = O(|E| \log|V|)$.

6.2 Path Compression

Path Compression: To improve the time efficiency of $find$ in the union by rank data structure.

```
function find(x)
if x ≠ π(x) :  π(x) = find(π(x))
return π(x)
```

Figure 5.7 The effect of path compression: `find`(I) followed by `find`(K).



6.2.1 Analyzing the Benefit from Path Compression

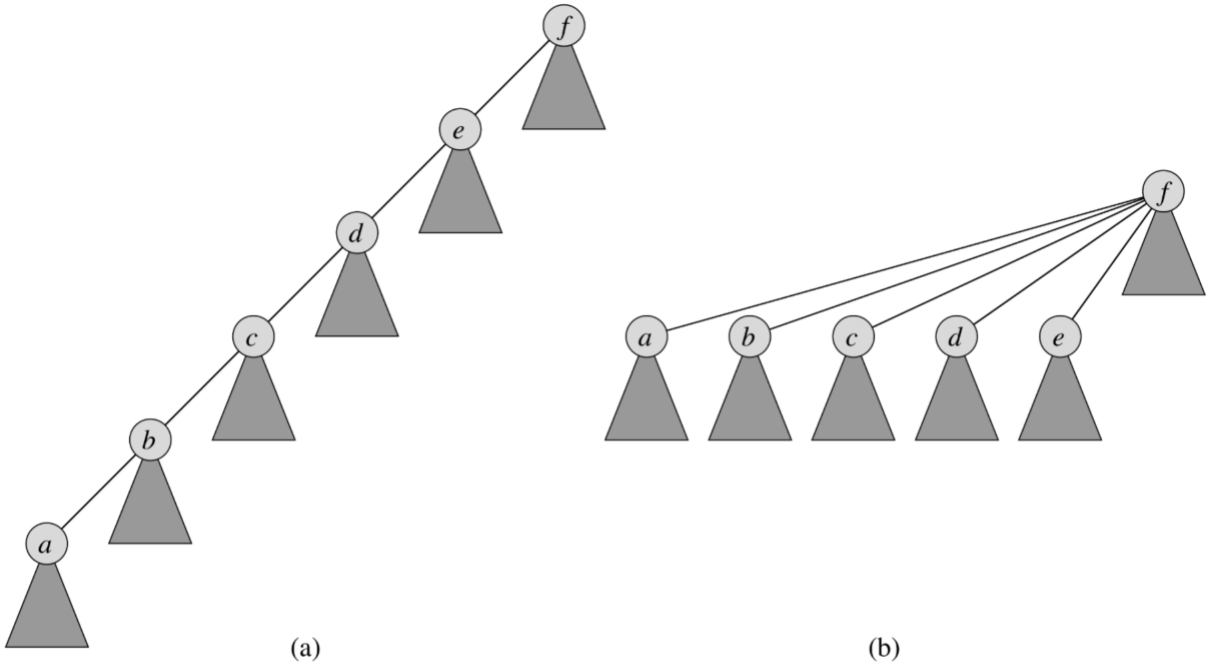


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

This analysis is simpler than the textbook but suffices to illustrate the benefit.

Claim 1. *Suppose we make a total of m calls to $makeset$, $union$ and $find$, where $find$ calls include the ones made in $union$, then the total time is $O(m)$.*

Proof. First, we assume that the following all have some constant cost, c:

- An invocation $makeset(x)$,
- Everything excluding the time for the two $find$'s in a call to $union$,
- An invocation of $find(x)$ to an x that is an immediate child of a root,
- An invocation of $find(x)$ to an x that is a root, and,
- The time to execute everything in an invocation to $find(x)$, but not the time for any recursive $find$'s that result. So, taking one "step up" the tree has a cost of c.

Thus, the only variable is if we have an invocation to $find(x)$ where x is of distance k from the root. This would end up costing us $k \cdot c$.

Now, in terms of our accounting, we do the following. For each invocation $makeset(x)$, rather than associating it with a cost of c only, we associate with a cost of $2c$. This is amortization – we are paying an up-front additional cost in anticipation of later cost for $find$.

Why $2c$? The reason is that we are paying an up-front cost of the additional c for the first (and only the first) invocation to $find(x)$ for each x that may happen in the future.

Now consider the cost of $find(x)$. We have two cases.

1. x is either a root or the immediate child of a root – then, our cost is c. Note that x may be an immediate child of a root because we have invoked $find$ before already on x, perhaps as a consequence of invoking $find$ on a descendent of x.
2. x is neither a root nor an immediate child. Then, we end up with k recursive invocations of $find$ until we hit an ancestor of x that is the immediate child of a root. Then, it must be the case that on x and all those ancestors, this is the first invocation of $find$. For which we have already paid up-front cost. So, the only cost for this $find$ is also c, which is for the ancestor that is an immediate child of the root.

Thus, our total cost is:

- $2c$ for each $makeset$,
- c for the work in each $union$ except for the calls to $find$, and,
- c for each call to $find$.

In total over m calls, this cost is $\leq 2c \cdot m = O(m)$.

6.3 Prim's Algorithm, and Min-Cut

6.3.1 Prim's Algorithm

Prim's Algorithm: Another one that uses the cut property but works differently from Kruskal's. But it is also a greedy algorithm.

In fact, Prim is exactly Dijkstra's, with a tweak.

Prim's Minimum Spanning Tree

Figure 4.8 ~~Dijkstra's shortest-path~~ algorithm.

prim

```
procedure dijkstra(G, l, s)                    ← choose any vertex as the start vertex, s
Input:  Graph G = (V, E), directed or undirected;   ← and connected
        positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.           "cost" of adding u to the MST.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0


H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):        dist(v) > l(u,v):
            dist(v) = dist(u) + l(u, v)        dist(v) = l(u,v)
            prev(v) = u
            decreasekey(H, v)
```

Example:

Set S	A	B	C	D	E	F
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

Proof of correctness for Prim: At any moment, we have a set S which are connected to the start vertex s by a path and whose edges are in some MST. Consider the cut $\langle S, V \setminus S \rangle$. In the next step, we pick as edge a light edge that crosses that cut.

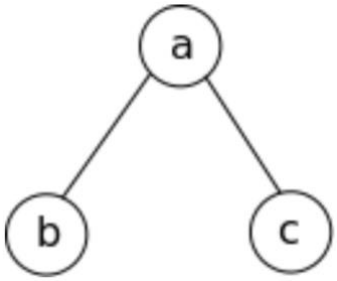
6.3.2 Min-Cut

Problem statement:

Input: Undirected, connected (unweighted) $G = \langle V, E \rangle$.

Output: A cut $\langle S, V \setminus S \rangle$ where $S \subset V, S \neq \emptyset$, such that the number of edges that cross the cut is minimized.

E.g., suppose we have input the following graph.



Then $\langle \{a\}, \{b, c\} \rangle$ is not a min-cut, but $\langle \{b\}, \{a, c\} \rangle$ is.

A candidate, randomized algorithm: repurpose Kruskal.

Assign same weight, e.g., 1, *to all edges.*

Shuffle edges so they will be chosen uniformly at random by Kruskal.

Run Kruskal.

Remove the last edge picked by Kruskal.

At the end, we end up with a cut $\langle S, V \setminus S \rangle$ such that each of $S, V \setminus S$ is a connected component.

Claim 1. $\Pr\{above\ algorithm\ outputs\ a\ min-cut\} \geq \frac{1}{|V|^2}$.

Proof. Let c be the size of a min-cut of G. At any stage of Kruskal just before the last edge is added, V is partitioned into ≥ 2 connected components. The number of edges that connects any connected component to any other $\geq c \implies$ number of edges incident on any connected component $\geq c$.

If we have $k \geq 2$ such connected components, then Kruskal has $\geq \frac{kc}{2}$ edges from which to pick an edge to add next.

So, $\Pr\{Kruskal\ picks\ an\ edge\ in\ a\ min-cut\ next\} \leq \frac{c}{kc/2} = \frac{2}{k}$.

So, $\Pr\{Kruskal\ does\ not\ pick\ an\ edge\ in\ a\ min-cut\ next\} \geq 1 - \frac{2}{k} = \frac{k-2}{k}$.

As this event in each round of edge-picking is independent of the other:

$$\Pr\{Kruskal\ never\ picks\ an\ edge\ in\ a\ min-cut\} \geq \frac{|V|-2}{|V|} \cdot \frac{|V|-3}{|V|-1} \cdot \frac{|V|-4}{|V|-2} \cdot \ldots \cdot \frac{1}{3}$$

$$= \frac{1}{|V|(|V|-1)} \geq \frac{1}{|V|^2}$$

6.4 Huffman Coding and Horn Formulas

More examples of problems that possess the greedy choice: Huffman coding and Horn formulas.

6.4.1 Huffman Coding

Huffman coding problem statement:

Input: A string in some alphabet, Σ

Output: A binary encoding of that string that is *unambiguous* and of minimum length

E.g., suppose you want to encode the string *aabaaccadddd* in binary.

An option: $fixed\ \#\ bits\ per\ symbol$. In this example, 2.

Then, $\#\ bits\ to\ encode\ string = 12 \times 2 = 24$.

Better option: Allocate $\#\ bits$ based on frequency.

Symbol	Frequency
a	5
b	1
c	2
d	4

E.g., $a \mapsto 0, d \mapsto 1, c \mapsto 00, b \mapsto 01$.

Ambiguous. Decoder does not know whether 0101 decodes to *adb* or *bb*.

Huffman coding:

- Bit-string b codes $\sigma \in \Sigma$ and c codes $\delta \in \Sigma \Longrightarrow$ neither b nor c is a prefix of the other.
    - E.g., $a \longmapsto 0, b \longmapsto 01$; then code for a is a prefix of code for b – not allowed.
- Maintain property that higher frequency $\Longrightarrow$ shorter code.

E.g., $a \longmapsto 0, d \longmapsto 10, c \longmapsto 110, b \longmapsto 111$.

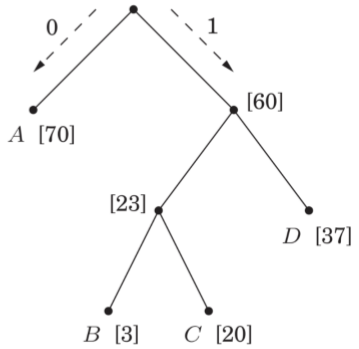
So, length of code for $aabaaccadddd = 5 \times 1 + 4 \times 2 + (1 + 2) \times 3 = 22\ bits$.

Another example:

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Symbol	Codeword
A	0
B	100
C	101
D	11

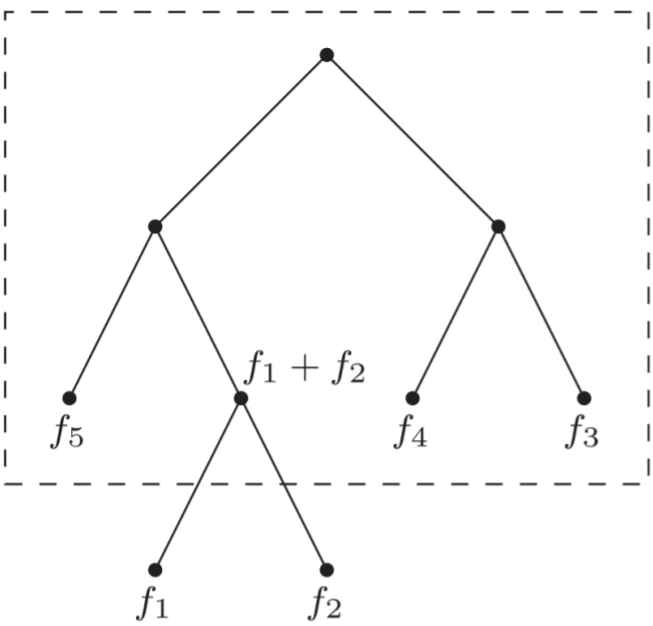


Turns out:

- Prefix-free codes such as the above can achieve optimal compression.
- Problem of building such an optimal tree possesses a greedy choice property:

    Starting with each symbol in a singleton set of itself, pick the least frequently occurring set to build the tree bottom-up.
- That is, designate leaves corresponding to each symbol $s_{\{1\}}, \ldots, s_{\{n\}}$. Assume they are ordered in non-decreasing frequencies $f_{\{1\}}, \ldots, f_{\{n\}}$.
- Make $s_{\{1\}}, s_{\{2\}}$ siblings by creating an internal node, call it $s_{\{1,2\}}$. Remove $s_{\{1\}}$ and $s_{\{2\}}$ from future consideration but add a new "symbol" $s_{\{1,2\}}$ with frequency $f_{\{1,2\}} = f_{\{1\}} + f_{\{2\}}$ for future steps. Then repeat.
- E.g., the above example would run as follows:
    - $\langle \{B\}, 3 \rangle, \langle \{C\}, 20 \rangle, \langle \{D\}, 37 \rangle, \langle \{A\}, 70 \rangle$.
    - $\langle \{B, C\}, 23 \rangle, \langle \{D\}, 37 \rangle, \langle \{A\}, 70 \rangle$.

- $\langle\{B, C, D\}, 60\rangle, \langle\{A\}, 70\rangle$.
- $\langle\{B, C, D, A\}, -\rangle$.




```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output: An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n: insert(H, i)
for k = n+1 to 2n−1:
   i = deletemin(H),  j = deletemin(H)
   create a node numbered k with children i, j
   f[k] = f[i] + f[j]
   insert(H, k)
```

Proof of correctness: Has one assuming prefix-free codes can achieve optimality. Proof is similar to the way we proved cut property for Minimum Spanning Tree.

6.4.2 Horn Formulas

Satisfiability of Horn formulas:

- Special case of particular interest of Boolean satisfiability problem.
- A *boolean variable*, x, takes a value of either **true** or **false**.
- We allow *propositions* or *clauses* or *formulas* to be built from boolean variables using boolean operators: $-,\ \wedge,\ \vee,\ \Longrightarrow,\ ()$.
- A proposition/clause/formula also has a truth value. A *semantics* tells us the truth value (i.e., **true** or **false**) of a clause given truth values of constituent variables.
- One more piece of commonly used terminology: a *literal* is either a boolean variable or its negation.

We adopt truth table based semantics.

p	$\overline{p}$
true	false
false	true

p	(p)
true	true
false	false

a	b	$a \wedge b$
true	true	true
true	false	false
false	true	false
false	false	false

a	b	$a \vee b$
true	true	true
true	false	true
false	true	true
false	false	false

a	b	$a \Longrightarrow b$
true	true	true
true	false	false
false	true	true
false	false	true

Boolean satisfiability problem:

> Given a boolean formula as defined above, does there exist an assignment of truth values to its constituent variables that causes the formula to evaluate to **true**?

Examples:

- The formula $(x_1 \vee \overline{x_2}) \vee \overline{x_1}$ is satisfiable.

  $x_1 = false, x_2 = true$ is a satisfying assignment.
- $x \wedge \bar{x}$ is not satisfiable.

In Horn satisfiability, we restrict the formula to a particular type.

- Formula is a conjunction of clauses, i.e., $f = C_1 \wedge C_2 \wedge \ldots \wedge C_n$.
- Each clause C_i constrained to be one of the following two types.
  1. Implication of a particular form: Left hand side is conjunction of zero or more positive literals; right hand side is a single positive literal.

     Examples:

     - $(z \wedge y \wedge w) \Longrightarrow u$
     - $a \Longrightarrow b$
     - $\Longrightarrow x$ (read as "if **true** then x)
  2. Pure negative clause: Disjunction of one or more negative literals.

     Examples:

     - $\bar{z} \vee \bar{y} \vee \bar{w}$
     - $\bar{x}$

Example of a formula:

$$
\begin{aligned}
& (w \wedge y \wedge z) \Longrightarrow x \\
\bigwedge\ & (x \wedge z) \Longrightarrow w \\
\bigwedge\ & x \Longrightarrow y \\
\bigwedge\ & \Longrightarrow x \\
\bigwedge\ & (x \wedge y) \Longrightarrow w \\
\bigwedge\ & (\bar{w} \vee \bar{x} \vee \bar{y}) \\
\bigwedge\ & \bar{z}
\end{aligned}
$$

A greedy (or "stingy") approach to determining Horn satisfiability:

*Set every variable to **false**.*

*Set variables to **true** one-by-one only if we must.*

In our example, in order:

- x must be set to **true** – clause: $\Longrightarrow x$.
- So, y must be set to **true** – clause: $x \Longrightarrow y$.
- So, w must be set to **true** – clause: $(x \vee y) \Longrightarrow w$.
- So, formula is not satisfiable – clause: $(\overline{w} \vee \bar{x} \vee \bar{y})$.

```
Input:  a Horn formula
Output: a satisfying assignment, if one exists

set all variables to false

while there is an implication that is not satisfied:
  set the right-hand variable of the implication to true

if all pure negative clauses are satisfied: return the
assignment
else: return "formula is not satisfiable"
```

Claim 1. *The above algorithm sets a variable to **true** if and only if in every satisfying assignment, that variable is **true**.*

Time efficiency of algorithm: $O(n^2)$.

- Where n is the length of input formula.

7 SET COVER, DYNAMIC PROGRAMMING

7.1 Set Cover

Inputs:

(i) A set of items B ("ground set"), and

(ii) A set of m subsets $S_1, \dots, S_m \subseteq B$ such that $\bigcup_{i=1}^{m} S_i = B$.

Output: a collection of the subsets $S_{j_1}, \dots, S_{j_k}$ such that:

(1) $\{j_1, \dots, j_k\} \subseteq \{1, \dots, m\}$,

(2) $\bigcup_{i=1}^{k} S_{j_i} = B$, and

(3) k is the minimum size of such a collection whose union is B.

A collection of subsets whose union is B is called a set cover – they "cover" the set B. We seek a minimum-sized set cover.

7.1.1 Example

- $B = \{a, b, \dots, k\}$
- Subsets:

$$S_1 = \{a, b, d, e, h, k\}$$
$$S_2 = \{a, b, c, d\}$$
$$S_3 = \{b, c, d\}$$
$$S_4 = \{a, e, f, g\}$$
$$S_5 = \{e, f, g\}$$
$$S_6 = \{a, h, i, j, k\}$$
$$S_7 = \{h, i, j, k\}$$

A minimum set cover: S_1, S_4, S_7. Another: S_2, S_5, S_7. Another: S_2, S_5, S_6.

7.1.2 Algorithm

A greedy algorithm for the problem is the following:

While some items from B remain uncovered

Include in our collection a subset that covers the most remaining items

In the example above, we would proceed as follows:

- We first pick S_1 as it covers most of the items.
- Then, S_4, S_5, S_6, S_7 each cover two new items, while S_2, S_3 each cover one only. Suppose we pick S_4.
- Then, S_2, S_3 still cover one new item, S_5 covers none, and S_6, S_7 cover two new items. Suppose we pick S_6.
- Then, S_6, S_7 each cover no new items, S_2, S_3 each cover one. Suppose we pick S_2.

Thus, our greedy algorithm:

- Is guaranteed to output a set cover, but
- One that is not necessarily of minimum size.

So, now we ask: how badly can our greedy algorithm perform?

One way to answer this question:

> Suppose for some input instance of the set cover problem, a minimum sized set cover, i.e., a set of subsets, is $\mathcal{C}^*$.
>
> Also suppose the output from our greedy algorithm for that instance is $\mathcal{C}$.
>
> How large can the ratio $|\mathcal{C}|/|\mathcal{C}^*|$ be, e.g., as a function of the size of the input instance?
>
> This ratio has to be ≥ 1.

Turns out we can answer this question. If the input instance of set cover is of size n, then the ratio is $O(\log n)$.

7.1.3 Proof

Proof. Define the n^{th} Harmonic number, $H(n)$ for $n \in \mathbb{Z}_0^+$:

$$H(n) = \begin{cases} 0, & if\ n = 0 \\ \sum_{i=1}^{n} \frac{1}{n}, & otherwise \end{cases}$$

Claim 1. $H(n) = O(\log n)$.

Proof shows that $|\mathcal{C}|/|\mathcal{C}| \leq H(x)$, where $x = \max\{|S_1|, |S_2|, \dots, |S_m|\}$.

Assume, without loss of any generality, that our greedy algorithm chooses $S_1, S_2, \ldots, S_{|\mathcal{C}|}$, in order. Associate a cost of 1 for every one of the $|\mathcal{C}|$ sets our greedy algorithm chooses. Also, attribute this cost equally across each of the new items from B that are covered by a set S_i. That is, suppose the item x is first covered by the subset S_i in our greedy algorithm. Then, the cost of x, denoted c_x, is:

$$c_x = \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

Now:

$$|\mathcal{C}| = \sum_{x \in B} c_x$$

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x$$

Our intent is to prove that for any such subset S in the input, $\sum_{x \in S} c_x \leq H(|S|)$.
Because if that is indeed true, then:

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \leq \sum_{S \in \mathcal{C}^*} H(|S|)$$

$$|\mathcal{C}| \leq |\mathcal{C}^*| \cdot H(\max\{|S| : S \textit{ is a subset of } B \textit{ in the input}\})$$

So now, we seek to prove: if S is any subset that is part of the input, $\sum_{x \in S} c_x \leq H(|S|)$.

We first adopt some notation for convenience. Let:

- $u_0 = |S|$ – these are the number of items in S that are not covered at the start of our greedy algorithm.
- For $S_1, S_2, \ldots, S_{|\mathcal{C}|}$, let $u_i = |S \setminus (S_1 \cup S_2 \cup \ldots \cup S_i)|$ – these are the items in S that are not covered after our greedy algorithm has selected, in sequence, $S_1, \ldots, S_i$.
- Let k be the smallest value such that $u_k = 0$, i.e., (i) every item in S is covered by $S_1 \cup S_2 \cup \ldots \cup S_k$, and (ii) for any $i < k$, $S_1 \cup \ldots \cup S_i$ leaves at least one item in S uncovered.

Now a couple of observations about the u_i's:

- For every $i = 1, \ldots, k$, $u_{i-1} \geq u_i$.
- Fir every $i = 1, \ldots, k$, # items in S first covered by $S_i = u_{i-1} - u_i$.

- So:

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}$$

Because:

1. $|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|$ from our greedy choice, and
2. $|S \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| = u_{i-1}$

So now we focus on that right hand side:

$$\sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \quad *$$

$$\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j}$$

$$= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right)$$

$$= \sum_{i=1}^{k} \big(H(u_{i-1}) - H(u_i) \big)$$

$$= \big(H(u_0) - H(u_1) \big) + \big(H(u_1) - H(u_2) \big) + \cdots + \big(H(u_{k-1}) - H(u_k) \big)$$

$$= H(u_0) - H(u_k) = H(u_0) - H(0) = H(u_0)$$

$$= H(|S|)$$

* # of values of j we "iterate" through in this summation: $u_{i-1} - (u_i + 1) + 1 = u_{i-1} - u_i$.

7.2 Dynamic Programming

Dynamic programming exploits optimal substructure.

> *Optimal substructure is the property that an optimal solution to a problem contains within it optimal solutions to subproblems.*

E.g., subpath of a shortest path is a shortest path.

Optimal substructure is elegantly expressed as a recurrence.

Dynamic programming realizes such a recurrence bottom-up.

> If solution to problem depends on solutions to subproblems, then compute the solutions to all those subproblems first, and then the solution to the problem.

Note: optimal substructure is a property of the (optimization) problem. Not of a particular algorithm. An algorithm may exploit an optimal substructure property of the problem for which it is an algorithm.

7.2.1 Shortest Path-Lengths in Directed Acyclic Graphs (DAGs)

Yes, we already have a highly efficient algorithm for this:

> Initialize all $dist$ values, except source's, to ∞.
>
> Linearize.
>
> In linearized order, update every edge once.

Recurrence that captures optimal substructure and linearization:

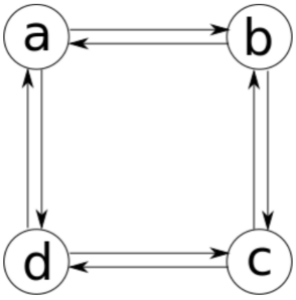
Suppose $V = 1, 2, \ldots, n$ in linearized order. Source s is one of those.

$$\delta(s,v) = \begin{cases} 0, & if\ v = s \\ \min\limits_{\substack{u \in \{1,\ldots,v-1\} \\ \langle u,v \rangle \in E}} \{\delta(s,u) + l(u,v), \infty\}, & otherwise \end{cases}$$

7.2.2 Longest Path-Lengths in DAGs

$$\delta(s,v) = \begin{cases} 0, & if\ v = s \\ \max\limits_{\substack{u \in \{1,\ldots,v-1\} \\ \langle u,v \rangle \in E}} \{\delta(s,u) + l(u,v), -\infty\}, & otherwise \end{cases}$$

Warning: longest (simple) paths have the optimal substructure we exploit above for acyclic graphs only. Not necessarily graphs that may contain cycles. Example:

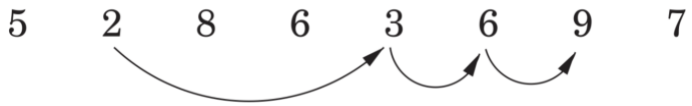


Longest simple path from a to d: $a \to b \to c \to d$.

But $b \to c$ is not longest simple path from b to c.

7.2.3 Longest Increasing Subsequences

- Input: sequence of integers.
- Output: (length of) longest increasing subsequence.

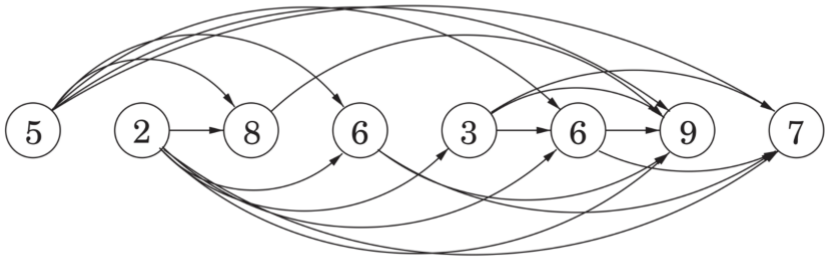


$5 \to 6 \to 7$ would be an increasing subsequence, but not the longest, which is $2 \to 3 \to 6 \to 9$.

Problem can be seen as generalization of longest path in DAGs:

- Edge $i \to j$ if and only if j^{th} number in the sequence $> i^{th}$ number.
- We seek length of longest path from any vertex as source to any destination.
- Edges are unweighted, i.e., all edges have same weight.

Figure 6.2 The dag of increasing subsequences.



Let the input sequence be $a[1], a[2], \ldots, a[n]$. Let $L(i)$ be length of longest increasing subsequence in the sequence $a[1], \ldots, a[i]$ that ends exactly at $a[i]$. Hypothesize $a[0] = -\infty$. Then:

$$L(i) = \begin{cases} 0, & if\ i = 0 \\ 1 + \max\limits_{\substack{0 \le j < i \\ a[j] < a[i]}} \{L(j)\}, & otherwise \end{cases}$$

Our final solution is: $\max\limits_{i \in \{1,\ldots,n\}} \{L(i)\}$.

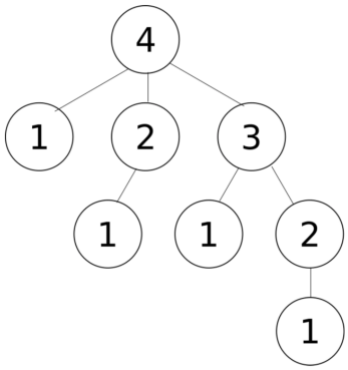
Can we realize such a recurrence top-down, i.e., using recursion?

Yes, but it may result in an inefficient (e.g., exponential-time) algorithm, when bottom-up would be polynomial-time.

```
LCS_OUTER_RECURSIVE(a[1, ..., n])
    L ← new array [1, ..., n]
    foreach i from 1 to n do
        L[i] ← LCS_INNER_RECURSIVE(a, i)
    return max{L[1], ..., L[n]}
```

```
LCS_INNER_RECURSIVE(a, i)
    ret ← 1
    foreach j from 1 to i − 1 do
        if a[j] < a[i] then
            x = LCS_INNER_RECURSIVE(a, j)
            if ret < 1 + L[j] then
                ret ← 1 + L[j]
    return ret
```

Worst-case recursion tree to compute $L(4)$:



A number of redundant computations. Also, number of nodes in the tree $= 2^{n-1}$.

Proof by induction on n. True for base case, $n = 1$. For the step, root has $n - 1$ children with labels $1, \ldots, n - 1$. So total number of nodes $=$ root $+$ nodes in those subtrees, where latter is from induction assumption.

$$1 + 2^0 + 2^1 + \cdots + 2^{n-2} = 1 + (2^{n-1} - 1) = 2^{n-1}$$

Bottom-up algorithm, i.e., algorithm based on Dynamic Programming:

```
LCS_DP(a[1, ..., n])
    L ← new array [1, ..., n]
    foreach i from 1 to n do
        L[i] ← 1
        foreach j from 1 to i − 1 do
            if a[j] < a[i] and 1 + L[j] > L[i] then
                L[i] ← 1 + L[j]
    return max{L[1], ..., L[n]}
```

This algorithm is $O(n^2)$.

Another option is to go top-down, i.e., use recursion, but with *memoization*.

```
LCS_OUTER_MEMOIZE(a[1, ..., n])
    L ← new array [1, ..., n]
    foreach i from 1 to n do
        L[i] ← 0
    LCS_INNER_MEMOIZE(a, L, n)
    return max{L[1], ..., L[n]}
```

```
LCS_INNER_MEMOIZE(a, L, i)
    if L[i] > 0 then return
    L[i] ← 1
    foreach j from 1 to i − 1 do
        if a[j] < a[i] then
            LCS_INNER_MEMOIZE(a, L, j)
            if L[i] < 1 + L[j] then
                L[i] ← 1 + L[j]
```

This algorithm also has time-efficiency of $O(n^2)$.

Does not apply to this problem/algorithm because we compute every $L[i]$, but going top-down can preclude unnecessary solving of sub-problems.

7.3 Dynamic Programming – Some More Examples

7.3.1 Edit Distance

- Input: two strings $x = x_1x_2 \ldots x_n$, $y = y_1y_2 \ldots y_m$, each $x_i, y_j \in$ some alphabet.
- Output: the edit distance to turn x into y. Four operations allowed:
    - Delete character; $cost = 1$.
    - Insert character; $cost = 1$.
    - Substitute character; $cost = 1$.
    - Leave character unchanged; $cost = 0$.

Edit distance is one of lowest total cost.

Example: two ways to turn SNOWY → SUNNY.

S	_	N	O	W		Y
S	U	N	N	_		Y

Cost = 3

_	S	N	O	W	_	Y
S	U	N	_	_	N	Y

Cost = 5

Each of the two above ways in which we write one string on top of the other in a manner that we have the same number of characters including _'s is called an *alignment*. Cost of an alignment = # spots in which the two strings differ. We seek an alignment of lowest cost.

Optimal substructure that leads to a recurrence:
Given problem instance $\langle x_1 \dots x_i, y_1 \dots y_j \rangle$, only three options going right-to-left:

- Substitute y_j for x_i, or leave it unchanged if $y_j = x_i$.
  - Subproblem that results: $\langle x_1 \dots x_{i-1}, y_1 \dots y_{j-1} \rangle$.
- Insert y_j.
  - Subproblem that results: $\langle x_1 \dots x_i, y_1 \dots y_{j-1} \rangle$.
- Delete x_i.
  - Subproblem that results: $\langle x_1 \dots x_{i-1}, y_1 \dots y_j \rangle$.

We pick the option that minimizes our total cost.

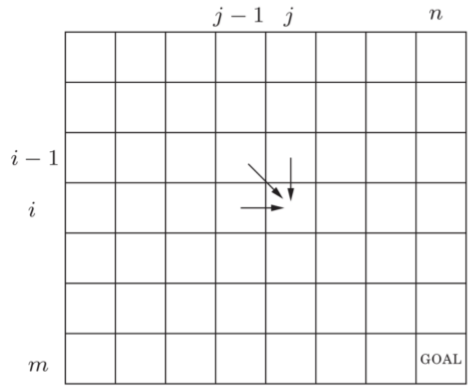
Let $E(i,j)$ represent the edit distance to turn $x_1 \dots x_i$ into $y_1 \dots y_j$. Then:

$$E(i,j) = \begin{cases} i, & if\ j = 0 \\ j, & if\ i = 0 \\ \min\{1 + E(i-1,j), 1 + E(i,j-1), E(i-1,j-1)\}, & if\ i > 0, j > 0\ and\ x_i = y_j \\ \min\{1 + E(i-1,j), 1 + E(i,j-1), 1 + E(i-1,j-1)\}, & otherwise \end{cases}$$

Bottom-up algorithm fills in a table $E[0 \dots n, 0 \dots m]$. Time-efficiency: $O(nm)$.

Figure 6.4 (a) The table of subproblems. Entries $E(i-1, j-1)$, $E(i-1, j)$, and $E(i, j-1)$ are needed to fill in $E(i, j)$. (b) The final table of values found by dynamic programming.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

7.3.2 Fractional Knapsack

- Can be thought of as a continuous, as opposed to discrete, version.
- Inputs:
  1. n items $\langle v_1, w_1\rangle, \ldots, \langle v_n, w_n\rangle$.
     - Each v_i is the total value of the i^{th} item.
     - Each w_i is the total weight of the i^{th} item.
  2. A capacity, W, for the knapsack.
- Output: the maximum total value we can achieve without exceeding W by taking any fraction of the items.

E.g., Items: $\langle 8, 4\rangle, \langle 5, 3\rangle, \langle 4, 3\rangle$; $W = 6$.

Solution: take all 4 units of Item (1) and 2 units of Item (2). Total value: $4/4 \times 8 + 2/3 \times 5$.

This version of knapsack possesses greedy choice: item that maximizes v_i/w_i.

7.3.3 0-1 Knapsack

- A discrete version:
- Inputs:
  1. n items $\langle v_1, w_1\rangle, \ldots, \langle v_n, w_n\rangle$,

- Each v_i is the total value of the i^{th} item.
- Each w_i is the total weight of the i^{th} item.

2. A capacity, W, for the knapsack.

- Output: the maximum total value we can achieve without exceeding W by taking all or none of an item.

This version does not possess the particular greedy choice that fractional knapsack does.
No polynomial-time algorithm exists unless $\mathrm{P} = \mathrm{NP}$.

An optimal substructure for 0-1 knapsack:
Should you take item n?

- Yes, if it maximizes total value, given our total capacity.
- Suppose $m[n, W]$ is maximum value we can achieve given items $1, \ldots, n$ and capacity W.
- Then we should take item n if and only if $v_n + m[n-1, W-w_n] \geq m[n-1, W]$.

More generally, suppose $m[i, C]$ is the maximum value we can achieve with items $1, \ldots, i$ and knapsack capacity C. Then:

$$m[i,C] = \begin{cases} -\infty, & if\ C < 0 \\ 0, & if\ i = 0\ or\ C = 0 \\ \max\{v_i + m[i-1, C-w_i], m[i-1, C]\}, & otherwise \end{cases}$$

Bottom-up algorithm populates $m[0 \ldots n, 0 \ldots W]$.

```
1 m ← new array[0...n, 0...W]
2 foreach i from 0 to n do
3     foreach j from 0 to W do
4         if i = 0 or j = 0 then m[i, j] ← 0
5         else if w_i ≤ j then m[i, j] ← max{v_i + m[i − 1, j − w_i], m[i − 1, j]}
6         else m[i, j] ← m[i − 1, j]
7 return m[n, W]
```

Time-efficiency: $O(nW)$, which is exponential in binary encoding of W.

7.3.4 Chain Matrix Multiplication

- Input: the dimensions of n matrices, $A_1, A_2, \dots, A_n$ such that the matrix product $A_1 \cdot \dots \cdot A_n$ exists.
- Output: to compute the matrix product $A_1 \cdot A_2 \cdot \dots \cdot A_n$. How should we parenthesize to minimize total number of scalar multiplications?

Note:

- Suppose A is $n \times m$, B is $p \times q$, then product $A \cdot B$ exists if and only if $m = p$.
- So, product $A_1 \cdot \dots \cdot A_n$ exists if and only if A_1 is $p_1 \times p_2$, A_2 is $p_2 \times p_3$, …, A_n is $p_n \times p_{n-1}$.
- If A is $n \times m$, B is $m \times p$, then $AB = [c_{i,j}]$ is $n \times p$ where $c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$. So, to compute each $c_{i,j}$, number of scalar multiplications $= m$. So, to compute $AB, n \times m \times p$ scalar multiplications.
- To compute ABC, can parenthesize in one of two ways: $(AB)C$ or $A(BC)$.
- Can make a difference in number of scalar multiplications. E.g., suppose A is 10×1, B is 1×10, C is 10×1.
    - $A(BC)$ costs $(1 \times 10 \times 1) + (10 \times 1 \times 1) = 20$ scalar multiplications.
    - $(AB)C$ costs $(10 \times 1 \times 10) + (10 \times 10 \times 1) = 200$ scalar multiplications.

Optimal substructure:

- Consider the product $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ for some $j > i$.
- If $j > i + 1$, we must parenthesize, i.e., "break" at some k between i and j, i.e., compute the product as: $(A_i \cdot \dots \cdot A_k) \cdot (A_{k+1} \cdot \dots \cdot A_j)$.
- Which k would we choose? The one that minimizes our cost.
- So now, suppose $m[i, j]$ is the minimum cost to compute the product $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$. We can write down a recurrence for it.
- Our final answer would be $m[1, n]$

$$m[i,j] = \begin{cases} \infty, & if\ j < i \\ 0, & if\ j = i \\ p_i \cdot p_{i+1} \cdot p_{i+2}, & if\ j = i+1 \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_i \cdot p_{k+1} \cdot p_{j+1}\}, & otherwise \end{cases}$$

Bottom-up algorithm:

$m \leftarrow$ new array ...
foreach i *from* 1 *to* $n-1$ **do**
    foreach j *from* $i+1$ *to* n **do**
        foreach k *from* i *to* $j-1$ **do**
            ⋮

8 SHORTEST PATHS, INDEPENDENT SETS, LINEAR PROGRAMMING

8.1 Dynamic Programming – Some More Examples

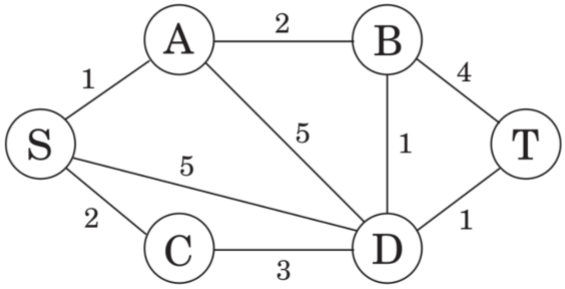
8.1.1 Two New Versions of Shortest Paths

New version 1:

- Inputs: (i) graph $G = \langle V, E, l \rangle$, (ii) source vertex $s \in V$, (iii) destination vertex $t \in V$, (iv) integer $k \in \mathbb{Z}_0^+$.
- Output: the length of a shortest path $s \dashrightarrow t$ of at most k edges, denote this as $\delta(s, t, k)$.

Example:

Figure 6.8 We want a path from s to t that is both short *and* has few edges.



$\delta(S, T, 4) = 5$. $\delta(S, T, 2) = 6$.

Recurrence that can be realized using dynamic programming:

$$\delta(s, u, m) = \begin{cases} \infty, & if\ m < 0\ or\ (m \leq 0\ and\ u \neq s) \\ 0, & if\ m \geq 0\ and\ u = s \\ \min\limits_{(v,u) \in E} \{\delta(s, v, m-1) + l(u, v), \infty\}, & otherwise \end{cases}$$

New version 2, of the shortest paths problem: all source-all destinations.

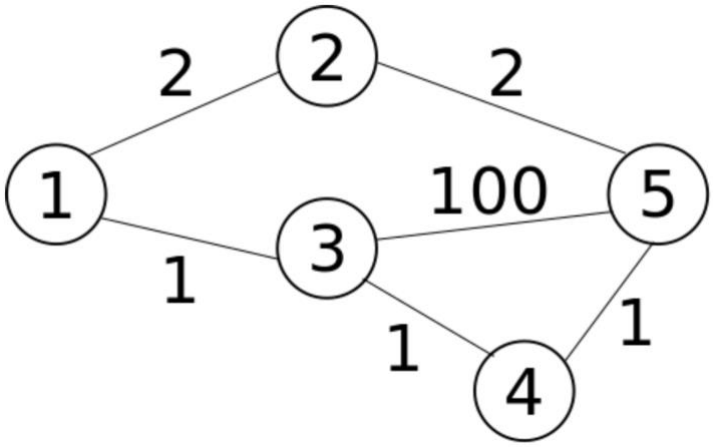
- Input: graph $G = \langle V, E, l \rangle$ with no negative weight cycles. Assume $V = \{1, 2, \ldots, n\}$.
- Output: a $|V| \times |V|$ matrix, $D[\cdot, \cdot]$ such that $D[i, j] = \delta(i, j)$.

Absolutely ingenious algorithm, Floyd-Warshall.

Define $\delta^{(k)}(i,j)$ as: shortest distance $i \rightsquigarrow j$ along any path whose intermediate vertices are allowed to be those in the set $\{1,2,\ldots,k\}$ only.

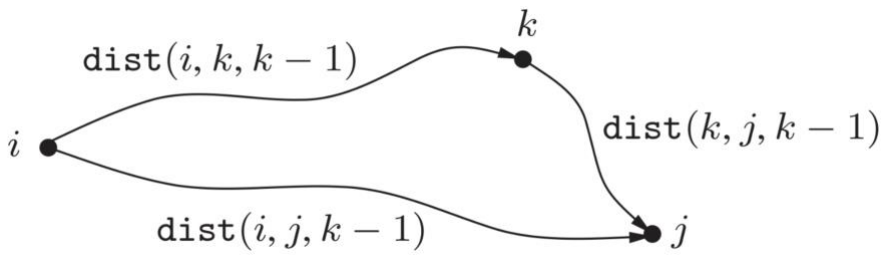
- Intermediate vertex means in a path that is the sequence $\langle i,\ldots,j\rangle$, the "…" are all vertices $\in \{1,2,\ldots,k\}$.

Example:



- $\delta^{(0)}(1,5) = \delta^{(1)}(1,5) = \infty$.
  - No edge $1 \rightarrow 5$.
- $\delta^{(2)}(1,5) = \delta^{(3)}(1,5) = 4$.
  - Allowing 3 in addition to 2 as intermediate vertex yields no benefit.
- $\delta^{(4)}(1,5) = \delta^{(5)}(1,5) = 3$.
  - Shortest path yielded if we allow intermediate vertices to be from $\{1,2,3,4\}$.
  - Allowing all $\{1,\ldots,5\}$ as intermediate vertices yield no additional benefit as every shortest path is simple.
  - Our final answer, $D[1,5] = \delta^{(5)}(1,5)$.

Optimal substructure:



Note: $dist(i,j,k) = \delta^{(k)}(i,j)$.

For a recurrence, first generalize weight function for convenience, $w: V \times V \to \mathbb{R} \cup \{\infty\}$.

$$w(u,v) = \begin{cases} 0, & if\ u = v \\ l(u,v), & if \langle u,v \rangle \in E \\ \infty, & otherwise \end{cases}$$

$$\delta^{(k)}(u,v) = \begin{cases} w(u,v), & if\ k = 0 \\ \min\{\delta^{(k-1)}(i,k) + \delta^{(k-1)}(k,j), \delta^{(k-1)}(i,j)\}, & otherwise \end{cases}$$

Bottom-up algorithm has time-efficiency $O(n^3)$.

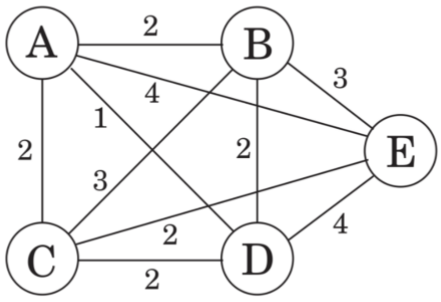
```
for i = 1 to n:
    for j = 1 to n:
        dist(i, j, 0) = ∞
for all (i, j) ∈ E:
    dist(i, j, 0) = ℓ(i, j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k − 1),  dist(i, j, k − 1)}
```

8.1.2 Travelling Salesman

- Input: a complete undirected graph $G = \langle V, E, l \rangle$, with $l: V \times V \to \mathbb{R}^+$, $V = \{1, \dots, n\}$.
- Output: (the weight of) a simple cycle $1 \rightsquigarrow 1$ of all vertices, of least total weight.

Example:

Figure 6.9 The optimal traveling salesman tour has length 10.



We assume $A \mapsto 1, B \mapsto 2, \dots, E \mapsto 5$. A TSP tour: $A \to D \to B \to E \to C \to A$.

Optimal substructure:

- Suppose $1 \rightsquigarrow 1$ is a TSP tour, and the vertex that immediately precedes 1 at the end is some $j \in V$, i.e., the tour is $1 \rightsquigarrow j \to 1$.
- Then, amongst all vertices in V, j yields the minimum: weight of simple path of all vertices starting at $1 + l(j, 1)$.
- Similarly, consider the vertex i that immediately precedes j in that TSP tour, i.e., the tour is $1 \rightsquigarrow i \to j \to 1$.
- Then, i is the one that minimizes: weight of simple path of all vertices in subgraph induced by $V \setminus \{j\} + l(i, j)$.
    - Given undirected $G = \langle V, E \rangle$ a subgraph $F = \langle V_f, E_f \rangle$ of G induced by $V_f \subseteq V$ has the property: $\langle u, v \rangle \in E_f \Leftrightarrow u, v \in V_F \wedge \langle u, v \rangle \in E$.
- Now, suppose $S \subseteq \{1, 2, \dots, n\}$ includes both 1 and $j \neq 1$. And let $C(S, j)$ be a shortest simple path of all vertices in S that starts at 1 and ends at j. Then:

$$C(S, j) = \begin{cases} 0, & if\ S = \{1\} \wedge j = 1 \\ \infty, & if |S| > 1 \wedge j = 1 \\ \min\limits_{i \in S \setminus \{j\}} \{C(S \setminus \{j\}, i) + l(i, j)\}, & otherwise \end{cases}$$

- Our final solution: $\min\limits_{j \in \{1, \dots, n\}} \{C(\{1, \dots, n\}, j) + l(j, 1)\}$.
- Bottom-up algorithm has time-efficiency $O(n^2 \cdot 2^n)$.
    - Better than enumerating all possibilities: $(n-1)! = \Theta\big((n-1)^{(n-1)}\big)$.

8.1.3 Independent Sets in Trees

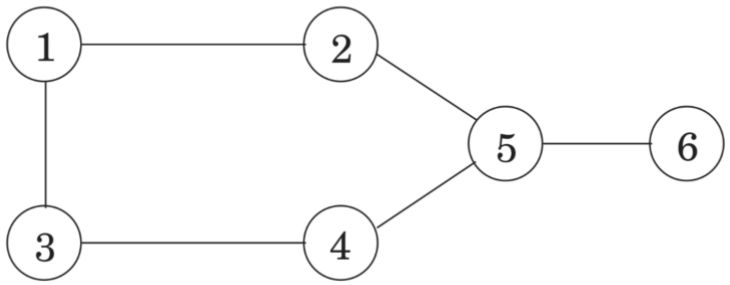
Definition: given an undirected graph $G = \langle V, E \rangle$, an independent set in it is a subset of the vertices $S \subseteq V$ with the property: $u, v \in S \Longrightarrow \langle u, v \rangle \notin E$.

Optimization problem: given input an undirected graph, identify (the size of) a maximum-sized independent set.

Here, we focus on trees only, i.e., input is a connected acyclic undirected graph.

Example:

Figure 6.10 The largest independent set in this graph has size 3.



That is <u>not</u> a tree. $\{1, 4\}$ is an independent set. So is $\{1, 4, 6\}$. So is $\{2, 3, 6\}$.

If we remove the $\langle 3, 4 \rangle$ edge, then the graph is a tree. Then, $\{2, 3, 4, 6\}$ is a maximum-sized independent set.

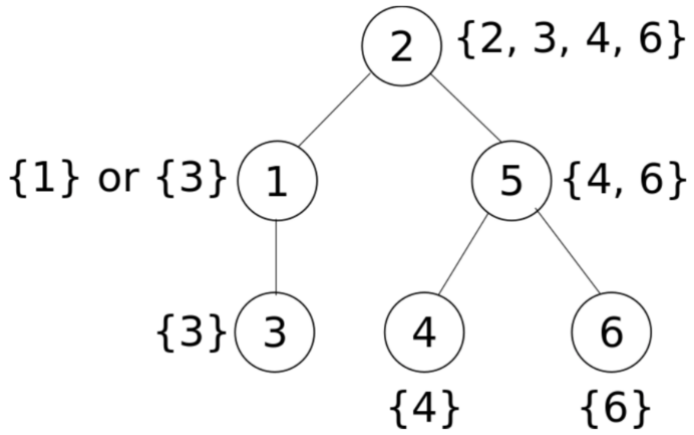
Optimal substructure:

- First, root the tree, i.e., pick some node and anoint it the root.
- Then, each node induces a subtree rooted at it.
- Define: $I(u) =$ size of max-sized independent set of subtree rooted at u.
- Then, we have a recurrence:

$$I(u) = \max\left\{1 + \sum_{grandchildren\ w\ of\ u} I(w), \sum_{children\ v\ of\ u} I(v)\right\}$$

- Mindset: either u is a member of a max-sized independent set rooted at u, or it is not.

Above example without the $\langle 3, 4 \rangle$ edge, rooted at 2.



8.2 Linear Programming

8.2.1 Definition

A linear program is an instance of the following problem.

Inputs:

1. An $n \times 1$ matrix c,
2. An $m \times n$ matrix A, and
3. An $m \times 1$ matrix b.

Output:

- An $n \times 1$ matrix x such that:

$$Ax \leq b, and$$
$$c^T x \; is \; maximized$$

Example: a company sells two products, Product 1 and 2. There is daily demand for at most d_1 of Product 1 and d_2 of Product 2. The company can manufacture at most u combined instances of Products 1 and 2 per day. The company makes a profit of p_1 on each unit of Product 1 and p_2 of each unit of Product 2 it sells. The company would like to maximize its total profit. How many instances of each of Product 1 and 2 should it manufacture per day?

This problem can be expressed as a linear program. Suppose x_1 is the number of Product 1 and x_2 of Product 2 the company manufactures. Then, we seek to solve for x_1, x_2 such that:

$$x_1 \leq d_1$$
$$x_2 \leq d_2$$
$$x_1 + x_2 \leq u$$
$$-x_1 \leq 0 \; (same \; as \; x_1 \geq 0)$$
$$-x_2 \leq 0$$
$$p_1 x_1 + p_2 x_2 \; is \; maximized$$

"Continuous" version of the problem: allow each $x_i \in \mathbb{R}$.

- Polynomial-time algorithm exists.

Discrete version: constrain $x_i \in \mathbb{Z}$.

- Or even $x_i \in \{0, 1\}$.
- But that can be captured by adding $x_i \geq 0, x_i \leq 1$.
- Polynomial-time algorithm highly unlikely to exist.

Note: problem may be *infeasible*, i.e., we may not be able to satisfy $Ax \leq b$ even if we allow $x \in \mathbb{R}^n$. E.g., maximize x under: $x \leq 0$ and $x \geq 1$.

The "maximize" part is called the *objective function*.

Turns out: lots of optimization and decision problems can be translated to linear programs.

Where "translate" has some precise properties.

In fact, we can prove: if a discrete optimization or decision problem has a polynomial-time algorithm, then such a translation to linear programming is possible.

Solving in practice: using a solver. E.g., CPLEX, Gurobi, etc.

In this course, we will not discuss algorithm(s) for linear programming, but rather introduce it as a segue to comparing problems from the standpoint of computational difficulty.

- E.g., if a polynomial-time algorithm exists for Linear Programming, then a polynomial-time algorithm exists for some other problem.

Such a comparison between problems underlies computational complexity.

8.2.2 Example

A version of the shortest-paths problem as an Integer Linear Program.

Inputs:

- A directed graph $G = \langle V, E, l \rangle$, with $l \in \mathbb{Z}^+$.
- Two vertices $s, d \in V$ with $s \neq d$.
- An integer $k \in [1, |V| - 1]$.

Output: the shortest distance $s \rightsquigarrow d$ in G on a simple path of exactly k edges.

An Integer Linear Program – certainly not unique

What unknowns should we adopt? How about the following:

- For each edge $e \in E$, adopt k variables $x_{e,i}$.
- Semantics: $x_{e,i} = 1$ if and only if the edge is the i^{th} edge on the path.

Right off the bat then, we get the following sets of constraints:

C1 For every $e \in E, i \in \{1, \dots, k\}, x_{e,i} \leq 1, x_{e,i} \geq 0$.

C2 We require exactly one edge in each of the k slots:

For every $i \in \{1, \dots, k\}$, a constraint: $\sum_{e \in E} x_{e,i} = 1$ (same as "$\geq$ and $\leq$").

C3 Each edge should show up at most once in the path:

For every edge $e \in E$, a constraint: $\sum_{i=1}^{k} x_{e,i} \leq 1$.

Next, we need to ensure that edges are chained one after the other in a meaningful way to comprise a path from s to d.

C4 The first edge has to leave s.

A constraint: $\sum_{e=\langle s,\cdot \rangle} {}_{e \in E} \; x_{e,1} \geq 1$.

C5 The last edge has to be incident on d.

A constraint: $\sum_{e=\langle \cdot,d \rangle} {}_{e \in E} \; x_{e,k} \geq 1$.

C6 Finally, we need to ensure that the path is indeed a path for the intermediate edges. That is, if edge $\langle u, v \rangle$ is the i^{th} edge, then the $(i-1)^{st}$ edge has to be incident on u, and the $(i+1)^{st}$ edge has to leave v.

To turn this into an "if and only if," think of it per vertex, instead of per edge. The i^{th} edge leaves u, if and only if the $(i-1)^{st}$ edge is incident on u. And the i^{th} edge is incident on v if and only if the $(i+1)^{st}$ edge leaves v.

The former constraint applies to all edges $2, \dots, k$ on the path, and the latter, to all edges $1, \dots, k-1$ on the path.

Thus, for every vertex $u \in V$, two sets of constraints:

For each $i = 2, \dots, k$, $\sum_{e=\langle u,\cdot \rangle} {}_{e \in E} \; x_{e,i} = \sum_{f\langle \cdot,u \rangle} {}_{f \in E} \; x_{f,i-1}$

For each $i = 1, \dots, k-1$, $\sum_{e=\langle \cdot,u \rangle} {}_{e \in E} \; x_{e,i} = \sum_{f\langle u,\cdot \rangle} {}_{f \in E} \; x_{f,i+1}$

We are all done with our feasibility constraints. And now our objective function.

F We want to add weights of exactly those edges that have been chosen for the path. And each edge may be chosen in any of the $1, \dots, k$ postitions.

So, we want to:

$$minimize \sum_{e \in E} \sum_{i=1}^{k} x_{e,i} \cdot l(e) \ \ (same\ as\ maximizing\ negation)$$

8.2.3 Properties

What properties does this translation to ILP have?

- A particular path $\langle s, u_1, \dots, u_{k-1}, d \rangle$ is a solution to the input instance of shortest paths if and only if $x_{\langle s,u_1 \rangle,1} = 1, x_{\langle u_1,u_2 \rangle,2} = 1, \dots, x_{\langle u_{k-1},d \rangle,k} = 1$, and all other $x_{i,j} = 0$ is a solution to the ILP instance.

  Also: part of this "if and only if" is (at least, should be) that the shortest-paths problem instance has no solution if and only if the ILP instance we produce has no solution, i.e., is infeasible.

- The ILP instance is computable in polynomial-time.

  Towards reasoning about this, what we are doing is computing a function r: $\mathcal{S} \to \mathcal{I}$, where $\mathcal{S}$ is the set of all possible instances of that particular shortest-paths problem, and $\mathcal{I}$ is the set of all possible instances of ILP, such that the "if and only if" property of the previous bullet holds.

  It turns out also that the function r is computable in time polynomial in the size of the input instance $s \in \mathcal{S}$.

  A meaningful characterization of the size of the output instance of $\mathcal{I}$:

  $$O(max\ length\ of\ a\ constraint \times number\ of\ constraints)$$

  C1, **C2**, **C3**: $O(k \cdot |E|)$

  C4, **C5**: $O(|E|)$

  C6: Naïve analysis suggests: $O(|V| \cdot k \cdot |E|)$

  More careful analysis tells us that each edge $e \in E$ appears in at most one set of "For each $i = \cdots$". So, $O(k \cdot |E|)$.

  F: $O(k \cdot |E| \cdot \log(\max\{l(e)\}))$

And as $k = O(|V|)$, the function r can be computed in time $O(|V||E| \log(\max\{l(e)\}))$, i.e., cubic time in the worst-case.

Some terminology: such a translation is called a *reduction*.

9 CIRCUIT EVALUATION, NP-COMPLETENESS

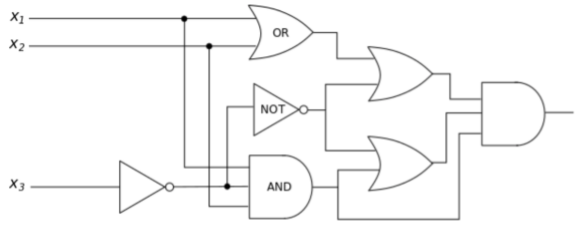
9.1 Circuit Evaluation

9.1.1 Definition

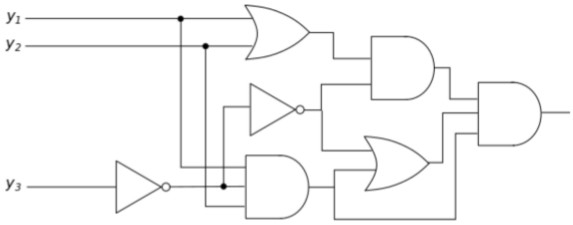
A *Boolean circuit*, in the context of this course, is a kind of directed acyclic graph.

- n input wires, 1 output wire.
- Gates: AND, OR and NOT.
    - Every gate has exactly 1 output wire.
    - AND and OR gates may be restricted to 2 input wires only or allowed to have $k \geq 2$ input wires.
    - NOT gate has exactly 1 input wire.

Examples:



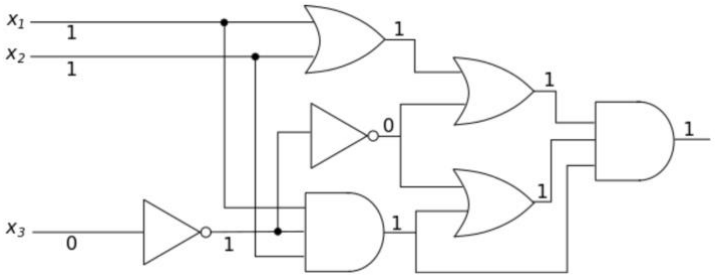
Circuit 1



Circuit 2

The problem: given as input such a circuit, is it *satisfiable*? That is, does there exist an assignment of 0 or 1 of each input wire that causes the output wire to evaluate to 1?

Claim: Circuit 1 is satisfiable. Proof: by construction; specifically, a satisfying assignment to the input wires. $x_1 = x_2 = 1, x_3 = 0$. Way to verify that that is a satisfying assignment: for that assignment to the input wires, label each distinct wire with the value it takes.



May not be the only satisfying assignment.

Claim: Circuit 2 is not satisfiable. Proof (admittedly not "efficient"): try every one of the 8 possible assignments to the input wires. Observe that the output wire takes value 0 for each of those input-assignments.

There exists a function, call it f, from the set of all such circuits to the set of all instances of Integer Linear Programming (ILP) such that: (i) i is a satisfiable instance of Boolean circuit satisfiability if and only if $f(i)$ is a true instance of ILP, and (ii) the size of $f(i)$ is polynomial in i (and $f(i)$ is computable in time polynomial in the size of i).
Note that because Boolean circuit satisfiability is a decision problem, we are presumably considering a decision version of ILP only, i.e., given $\boldsymbol{A} \in \mathbb{Z}^{n \times m}, \boldsymbol{b} \in \mathbb{Z}^n$, does there exist $\boldsymbol{x} \in \mathbb{Z}^m$ such that $\boldsymbol{Ax} \leq \boldsymbol{b}$?

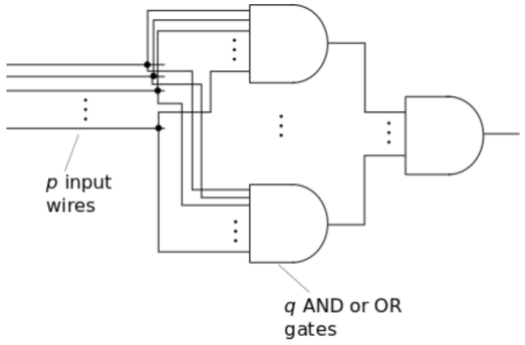
9.1.2 Proof

Proof by construction:

1. Introduce an unknown x in the ILP instance for every distinct wire in the circuit.
   Semantics: $x = 1$ if and only if the wire takes value 1 for the values the input wires take.
2. For every wire x including the input wires, constraints: $x \geq 0, x \leq 1$.
3. For an OR gate, suppose its input wires are $x_1, \ldots, x_k$ and output wire is y. Then constraints:
   (a) For each $i = 1, \ldots, k, y \geq x_i$, and
   (b) A constraint $y \leq x_1 + \cdots + x_k$.
4. For an AND gate, suppose its input wires are $x_1, \ldots, x_k$ and output wire is y. Then constraints:
   (a) For each $i = 1, \ldots, k, y \leq x_i$, and
   (b) A constraint $y \geq x_1 + \cdots + x_k - (k - 1)$.
5. For a NOT gate, suppose its input wire is x and output wire is y. Then constraint: $y = 1 - x$.
6. For the output wire x, a constraint, $x \geq 1$.

If our input circuit has n distinct wires and m gates, then output ILP instance has size $O(nm)$.

An input that elicits worst-case size of ILP instance:



Note, however, that natural encoding of circuit instance has size $\Theta(nm)$. So, $f(i)$ has size linear in i.

Turns out it works the other way as well. That is, there exists a function, call it g, from the set of all instances of ILP to the set of all Boolean circuits such that: (i) i is a true instance of ILP if and only if $g(i)$ is a satisfiable instance of Boolean circuit satisfiability, and (ii) the size of $g(i)$ is polynomial in i (and g is computable in time polynomial in size of i).
By "ILP" here we presumably mean a decision version, i.e., an instance is $\boldsymbol{Ax} \leq \boldsymbol{b}$.

Proof by construction again. We encode $\boldsymbol{Ax} \leq \boldsymbol{b}$ as a Boolean circuit. How do we do that?

- Assume binary encoding for all integers $a_{i,j}$, x_k and b_l.
- Assume we encode integers in the two's complement approach from Lecture 2(a): integers in the range $[-2^{n-1}, 2^{n-1} - 1]$ are represented using n bits as non-negative integers modulo 2^n.
- Inputs to the circuit are $x_{i,j}$ each bit for each of the x_i's.
- Note that there exists an $\boldsymbol{x}$ that satisfies $\boldsymbol{Ax} \leq \boldsymbol{b}$ if and only if there exists each x_i modulo $\max\{|b_l|\}$. So, we need to allocate only as many bits to each x_i as is needed to encode $\max\{|b_l|\}$.

Building-block circuits:

- Circuit to add two non-negative integers encoded in binary.
- Circuit to multiply two non-negative integers encoded in binary.

- Circuit to check for two integers encoded in binary $x \leq y$.

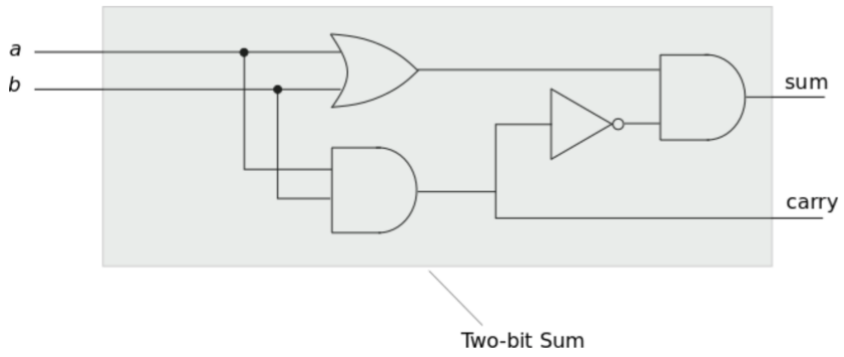
9.1.3 Addition

Addition: is done bit-by-bit. But need to account for a carry bit. That is, suppose $\max\{n, m\} = n$. Then, adding $x_{n-1} \ldots x_0$ and $y_{m-1} \ldots y_0$ can be done as follows:

$$\begin{array}{cccccc} c_{n-1} & c_{n-2} & \cdots & c_0 \\ + & + & \cdots & + \\ x_{n-1} & x_{n-2} & \cdots & x_0 \\ + & + & \cdots & + \\ y_{n-1} & y_{n-2} & \cdots & y_0 \\ \hline s_n & s_{n-1} & s_{n-2} & \cdots & s_0 \end{array}$$

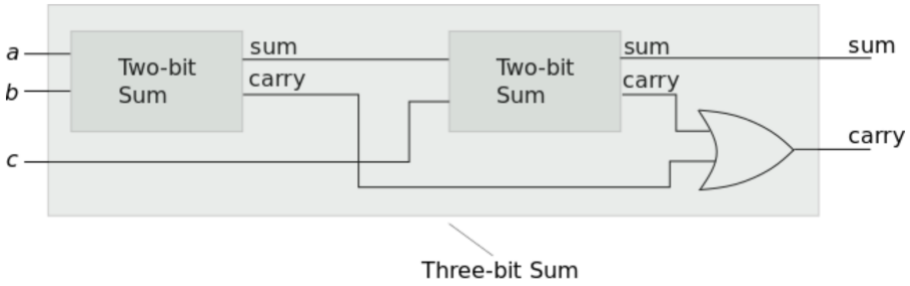
Where $c_0 = 0$, and $y_{n-1} = y_{n-2} = \cdots = y_m = 0$.

So, a building-block circuit would give us the two-bit output for the sum $a + b + c$, for $a, b, c \in \{0, 1\}$. We start with a circuit that gives us the sum and carry for two bits, i.e., $a + b$, for $a, b \in \{0, 1\}$.

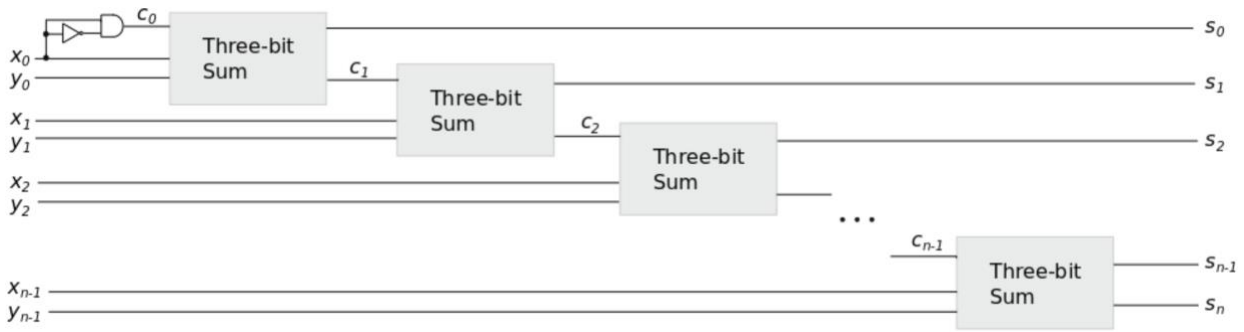


sum = a XOR b
= (a OR b) AND (NOT (a AND b))

And now a circuit for $a + b + c, a, b, c \in \{0, 1\}$:



And finally, a circuit for $(x_{n-1} \ldots x_0) + (y_{n-1} \ldots y_0)$, each $x_i, y_j \in \{0, 1\}$.



So, size of circuit to add an n-bit integer to an m-bit integer: $\Theta(\max\{n, m\})$.

9.1.4 Multiplication

Multiplication: $(x_{n-1} \dots x_0) \times (y_{m-1} \dots y_0)$.

Can realize either repeated halving, or school style. The latter for $n = 3, m = 4$:

			x_2	x_1	x_0	
		y_3	y_2	y_1	y_0	
0	0	0	$(x_2 \wedge y_0)$	$(x_1 \wedge y_0)$	$(x_0 \wedge y_0)$	
$+$	$+$	$+$	$+$	$+$	$+$	
0	0	$(x_2 \wedge y_1)$	$(x_1 \wedge y_1)$	$(x_0 \wedge y_1)$	0	
$+$	$+$	$+$	$+$	$+$	$+$	
0	$(x_2 \wedge y_2)$	$(x_1 \wedge y_2)$	$(x_0 \wedge y_2)$	0	0	
$+$	$+$	$+$	$+$	$+$	$+$	
$(x_2 \wedge y_3)$	$(x_1 \wedge y_3)$	$(x_0 \wedge y_3)$	0	0	0	
m_6	m_5	m_4	m_3	m_2	m_1	m_0

So: $\Theta(nm)$ AND's and $\Theta\big(m(n + m - 1)\big)$ +'s. So, resultant circuit has size $\Theta(nm + m^2)$.

9.1.5 Comparison

And $(x_{n-1} \dots x_0) \overset{?}{\leq} (y_{n-1} \dots y_0)$ assuming each of those encodes a non-negative integer.

This is $\equiv (y_{n-1} \dots y_0) - (x_{n-1} \dots x_0) \overset{?}{\geq} 0$.

We can perform subtraction bit-by-bit, with a borrow, instead of carry, bit. Finally, to check that the result is ≥ 0, we check the final borrow bit.

Following are examples $0110 - 1001$ vs. $1001 - 0110$. The negation of the final borrow bit tells us whether the result is ≥ 0 or not.

$$\begin{array}{cccc} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 1 \\ \boxed{1} & 0 & 0 & 1 \end{array} \leftarrow \text{borrow} \qquad \begin{array}{cccc} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 \\ \boxed{0} & 1 & 1 & 0 \end{array} \leftarrow \text{borrow}$$

So, circuit size is $\Theta(n)$ for $\leq$.

9.1.6 Summary

So, we first encode each $a_{i,1} \cdot x_1 + a_{i,2} \cdot x_2 + \cdots + a_{i,m} \cdot x_m \leq b_i$ as a circuit.

Each of those circuits has a single output wire, call it w_i.

We finally do an AND of $w_1, \ldots, w_n$ – this is the output wire of our circuit that encodes $\boldsymbol{Ax} \leq \boldsymbol{b}$.

Our circuit has the properties: (i) it can be computed in quadratic-time, and (ii) the circuit is satisfiable if and only if the input ILP instance is true.

So, we can claim:

Claim 1. *There is a polynomial-time algorithm for (the decision version of) ILP if and only if there is a polynomial-time algorithm for Boolean circuit satisfiability*.

Aside, but important observation: neither the function f nor g is invertible in the sense we know for functions. Because neither f nor g is onto, i.e., surjective.

9.2 Non-Determinism and the Class NP

Two classes of algorithmic problems: optimization problems and decision problems.

Optimization problem: given as input (*something*), what is the shortest/fastest/minimum…

Examples:

- Given a graph $G = \langle V, E, l \rangle$ where l: $E \to \mathbb{R}_0^+$, and two vertices s, t, what is the shortest distance $s \rightsquigarrow t$ on any simple path?
- Given a connected undirected graph $G = \langle V, E, l \rangle$, what is the smallest sum of weights of edges of any spanning tree of G?
- Given n items, each a pair of positive integers $\langle v_i, w_i \rangle$, and a positive integer C, what is $\max \sum_{v_{i_j} \in V'} v_{i_j}$ for any $I' \subseteq \{\langle v_1, w_1 \rangle, \ldots, \langle v_n, w_n \rangle\}, V' = \{v \mid \langle v, \cdot \rangle \in I'\}$ for which $\sum_{w_{i_j} \in W'} w_{i_j} \leq C$, where $W' = \{w \mid \langle \cdot, w \rangle \in I'\}$?

Decision problem: a function whose co-domain is $\{true, false\}$.

Examples:

- Given a graph $G = \langle V, E, l \rangle$ where l: $E \to \mathbb{R}_0^+$, two vertices s, t and $d \in \mathbb{R}$, does there exist a simple path $s \rightsquigarrow t$ of length $\leq d$?
- Given a connected undirected graph $G = \langle V, E, l \rangle$ and $k \in R$, does there exist a spanning tree of G whose sum of edge weights $\leq k$?
- Given n items, each a pair of positive integers $\langle v_i, w_i \rangle$, a positive integer C and a positive integer k, does there exist $I' \subseteq \{\langle v_1, w_1 \rangle, \ldots, \langle v_n, w_n \rangle\}$ such that $\sum_{v_{i_j} \in V'} v_{i_j} \geq k$ and $\sum_{w_{i_j} \in W'} w_{i_j} \leq C$, where $V' = \{v \mid \langle v, \cdot \rangle \in I'\}$ and $W' = \{w \mid \langle \cdot, w \rangle \in I'\}$?

Relating the two classes of problems: there are lots of optimization problems with the following property.

> Given an optimization problem i, there exists a decision problem d such that: there is a polynomial-time algorithm for i if and only if there is a polynomial-time algorithm for d.

Consequence of the above observation: we can focus on whether there exists a polynomial-time algorithm for the decision version. That is, we can restrict our attention entirely to decision problems only.

Examples of problems that possess the above property: all three examples from above.

- "Only if" direction: given an algorithm, call it O, for the optimization problem, a polynomial-time algorithm for the decision problem, call that algorithm D, would be as follows. Given input $\langle G, s, t, d \rangle$, invoke $O(G, s, t)$, and get the output o. Now compare $o \overset{?}{\leq} d$. If $O(\cdot)$ runs in polynomial-time, so does $D(\cdot)$ because running-time of $O(\cdot)$ must be $\Omega(size\ of\ encoding\ of\ its\ output)$.
- "If" direction: given an algorithm, call it $D(\cdot)$ for the decision version, following is a polynomial-time algorithm, call it $O(\cdot)$, for the optimization version. Let $m = \max_{e \in E}\{l(e)\}$. Carry out a binary-search on d between 0 and $(|V| - 1) \cdot m$ by repeatedly invoking $D(\cdot)$. For the time-efficiency of this approach, consider the following. Suppose the size to encode the input to $O(\cdot)$, i.e., $\langle G = \langle V, E, l \rangle, s, t \rangle$, is n. Then, $(|V| - 1) \cdot m$ can

be encoded with size $O(n)$. Suppose the running-time of $D(\cdot)$ on input of size x is x^k, for some constant k. Then, in our approach, we invoke $D(\cdot)$ $O(\log(|V| \cdot m))$, i.e., $O(\log n)$, times, each time with an input size of $O(n)$. So, our total running-time is $O(n^k \cdot \log n)$, which is polynomial in n.

- The proof for the "only if" and "if" directions for the other two problems is similar.

So, to summarize:

- There exist optimization problems for which it suffices to focus on a carefully articulated decision problem. "Carefully articulated" – such that a polynomial-time algorithm exists for the optimization problem provided one exists for the decision problem.
- We are <u>not</u> claiming that for any given optimization problem, this is possible. Just that there appears to be lots of optimization problems for which this is possible.

Focusing on decision problems, in particular *search problems*, a search problem is a decision problem D with the following property.

> There exists an algorithm, call it C, for each such D such that: an instance i of D is **true**, if and only if there exists a string s such that (i) the size of s is at worst polynomial in the size of i, (ii) $C(i, s) = \boldsymbol{true}$, and (iii) C is polynomial time.

Examples:

Each of the three examples from the previous page is a search problem. For the first problem, an instance of the decision problem $i = \langle G, x, y, d \rangle$. For such an instance that is **true**, an s is a path $\langle x, u_1, \dots, u_k, y \rangle$. An algorithm C, given input such an i and s would check that s is indeed a valid path in G, and that the sum of its edge weights is indeed d.

An s and C for the other two problems are similar. For the second problem, an s that works is a minimum-spanning tree of G. For the third problem, it is a subset I'.

Such a string s is a *solution* to the search problem. Note that s and C live together.

Other names for such an s folks use: a *certificate* or *witness*. Why? Because such an s testifies to the truth of the instance of the decision problem i. E.g., for the first problem, suppose given an

instance $\langle G, x, y, d \rangle$, Alice says that the instance is **true**. And you tell her, "prove it." What she can do is provide you with $\langle C, s \rangle$, where s is exactly a path as we discuss above.

Claim 1. *Suppose D is a decision problem for which there exists a polynomial-time algorithm. Then, D is a search problem*.

Proof. Let L be a polynomial-time algorithm for D. Adopt $C = L$, $s =$ the empty string.

Conjecture 1. *There exist decision problems for which no polynomial-time algorithm exists but are search problems*.

We emphasize that Conjecture 1 is a conjecture only; it has not been proven to be true.

Some decision problems that are search problems, but for which we conjecture no polynomial-time algorithm exists:

1. Decision version of ILP. That is, given as input $\langle \boldsymbol{A}, \boldsymbol{b} \rangle$ where $\boldsymbol{A} \in \mathbb{Z}^{n \times m}$ and $\boldsymbol{b} \in \mathbb{Z}^n$, does there exist $\boldsymbol{x} \in \mathbb{Z}^m$ such that $\boldsymbol{Ax} \leq \boldsymbol{b}$?

   Proof that it is a search problem: by construction. Given an instance of the problem that is **true**, a solution/witness/certificate s is such an $\boldsymbol{x}$. A corresponding C checks that indeed for that $\boldsymbol{x}$, it is true that $\boldsymbol{Ax} \leq \boldsymbol{b}$.
2. Boolean circuit satisfiability, CIRCUIT-SAT. Given as input a Boolean circuit, is it satisfiable?

   Proof that it is a search problem: by construction. An s is an assignment of 0 or 1 to each input wire for which the output wire of the circuit evaluates to 1.
3. Boolean satisfiability for formulas in propositional logic, SAT. Given a formula f in propositional logic, is it satisfiable?

   A *Boolean variable*, x, takes value either 0 (**false**) or 1 (**true**).

   A *literal* is a Boolean variable x or its negation $\neg x$ or $\overline{x}$.

   A formula in propositional logic comprises Boolean variables and operators $\vee$,$\wedge$, $\neg$ and (). E.g., given Boolean variables x, y, z, the following is a formula:

$$\Big(x \wedge \big((\neg y \vee \neg x) \wedge z\big)\Big) \vee y$$

The *satisfiability* problem is: does there exist an assignment of 0 or 1 to each variable that causes the formula to evaluate to 1?

4. Boolean satisfiability for formulas in propositional logic that are in Conjunctive Normal Form, CNF-SAT. Given a formula f in propositional logic that is in CNF, is it satisfiable?

   A *clause* for a formula in CNF is some literals connected by ∨. E.g., $(\neg x \vee y)$ is a clause.

   A formula in CNF is a sequence of clauses connected by ∧. E.g., $(\neg x \vee y) \wedge (x \vee \neg z \vee y) \wedge (x \vee y \vee \neg z)$

   Note that such a formula is satisfiable if and only if every clause is simultaneously satisfiable. And every clause is satisfied if and only if at least one of the literals in it is 1.

5. Boolean satisfiability for formulas in propositional logic that are in Conjunctive Normal Form with at most 3 literals per clause, Atmost-3-CNF-SAT. Given a formula f in propositional logic that is in CNF with at most 3 literals per clause, is it satisfiable?

   Every clause is allowed at most 3 literals.

   For example, $(\neg x \vee y) \wedge (x \vee \neg z \vee y) \wedge (x \vee y \vee \neg z)$ is in Atmost-3-CNF.

   But $(\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_2)$ is not.

6. Boolean satisfiability for formulas in propositional logic that are in Conjunctive Normal Form with exactly 3 literals per clause, Exactly-3-CNF-SAT. Given a formula f in propositional logic that is in CNF with exactly 3 literals per clause, is it satisfiable?
7. Travelling Salesman, TSP. Given as input (i) an undirected graph $G = \langle V, E, l \rangle$ which is complete and l: $E \to \mathbb{R}^+$, (ii) $s \in V$, and (iii) a "budget" $b \in \mathbb{R}^+$, does there exist a simple cycle $s \rightsquigarrow s$ whose sum of edge weights is $\leq b$?
8. (Decision version of) longest paths in graphs, 3D matching, Knapsack, Independent Set, Rudrata/Hamiltonian path, Balanced cut, …
9. Some (but not necessarily all) puzzles: Sudoku, Crossword.

Is there something common to all these problems? Yes.

- They are decision problems for which no polynomial-time algorithm is conjectured to exist, but which are search problems.

- Or equivalently: all of them admit polynomial-time *non-deterministic* algorithms.

What is non-determinism? A property of the underlying computer to make "correct choices." No such computer exists in practice. So, a theoretical model. But one that helps understand the characteristics of search problems.

For examples of non-deterministic algorithms, consider the following three decision problems.

- Given as input an undirected graph G, is it connected?
- Given as input an array $A[1, \ldots, n]$ and an item i, is $i \in A$?
- Given as input a Boolean circuit, is it satisfiable?

Given as input an undirected graph G, is it connected?

Deterministic algorithm: BFS with some vertex as source; then check if $\boldsymbol{dist}(\cdot)$ value is $< \infty$ for every vertex. Worst-case time-efficiency: $\Theta(|V| + |E|)$.

A non-deterministic algorithm:

```
CONNECTED-NONDET(G = ⟨V, E⟩)
 1  foreach u ∈ V do  visited(u) ← false
 2  Done ← ∅
 3  u ← some vertex from V
 4  while visited(u) = false do
 5      visited(u) ← true
 6      Done ← Done ∪ {u}
 7      Non-deterministically pick v ∈ Done
 8      if adj[v] ≠ ∅ then
 9          Non-deterministically pick a neighbour w of v
10          u ← w        if visited(w) = false then:
11  foreach u ∈ V do
12      if visited(u) = false then  return false
13  return true
```

Two distinguishing aspects under the non-deterministic model of computation:

- We are allowed to make a "non-deterministic" choice in the algorithm. Lines (7) and (9) in the above algorithm. Each such choice is assumed to have same time-cost as a corresponding deterministic choice.

  What kind of choices are we allowed to make non-deterministically? Typically, from a set that we can assemble in at worst polynomial-time.
- We change the notion of correctness to the following.

A non-deterministic algorithm for a decision problem D is correct if:

Given a **true** instance of D as input, there exists a sequence of non-deterministic choices the algorithm makes for which the output of the algorithm is **true**; and given a **false** instance of D as input, the algorithm is guaranteed to output **false**.

Claim 2. *$CONNECTED - NONDET$ is correct.*

Claim 3. *$CONNECTED - NONDET$ runs in time $O(|V|)$.*

Given as input an array $A[1, \dots, n]$ and an item i, is $i \in A$?

A deterministic algorithm: linear search. Time-efficiency: $O(n)$.

A non-deterministic algorithm:

```
FIND-NONDET(A[1, ..., n], i)
1 Non-deterministically pick j ∈ {1, ..., n}
2 if A[j] = i then return true
3 else return false
```

Claim 4. *$FIND - NONDET$ is correct.*

Claim 5. *$FIND - NONDET$ runs in time $O(1)$.*

Given as input a Boolean circuit, is it satisfiable?

No deterministic polynomial-time algorithm known to exist.

A non-deterministic, polynomial- (linear-) time, correct algorithm:

```
CIRCUIT-SAT-NONDET(C)
1 foreach input wire in C do
2     Non-deterministically pick an assignment from {0, 1}
3 Evaluate value for each wire in C for those input-wire assignments
4 if output wire's value = 1 then return true
5 else return false
```

Observation: non-deterministic model of computation is at least as powerful as deterministic model. And may be strictly more powerful.

Whether second statement is indeed true is an open question. Underlies $\boldsymbol{P \overset{?}{=} NP}$.

Connection to search problems: a sequence of non-deterministic choices that leads to a **true** output in such an algorithm is exactly a solution s.

And finally, the notion of a computational complexity class, and the classes $\boldsymbol{P}$ and $\boldsymbol{NP}$.
We define a *computational complexity class* or simply, complexity class, as a set of decision problems. We use n to denote the size of the problem instance, i.e., input-size.

The complexity class $\boldsymbol{P}$ comprises those decision problems for each of which there exists a deterministic algorithm whose time-efficiency is $O(n^k)$ for some constant k.
The complexity class $\boldsymbol{NP}$ comprises those decision problems for each of which there exists a non-deterministic algorithm whose time-efficiency is $O(n^k)$ for some constant k.

Claim 6. $\boldsymbol{P} \subseteq \boldsymbol{NP}$.
Proof. From the definitions of $\boldsymbol{P}$ and $\boldsymbol{NP}$, and the fact that every deterministic polynomial-time algorithm is a non-deterministic polynomial-time algorithm.

Claim 7. *A decision problem q is in $\boldsymbol{NP}$ if and only if q is a search problem.*
Proof (*informal*). Given a non-deterministic algorithm that is correct, adopt its deterministic version as C, and a sequence of non-deterministic choices that leads to an output of **true** as s. Given that q is a search problem, adopt a non-deterministic version of C as an algorithm for q, with s as the sequence of non-deterministic choices.

$\boldsymbol{P}$ and $\boldsymbol{NP}$ are not the only complexity classes of interest. Some others:

- $\boldsymbol{L}$: the set of decision problems for each of which there exists a deterministic algorithm whose space-efficiency is $O(\log n)$.
- $\boldsymbol{NL}$: the set of decision problems for each of which there exists a non-deterministic algorithm whose space-efficiency is $O(\log n)$.
    - Example of a problem in $\boldsymbol{NL}$: given as input an undirected graph, two vertices, s, t, and an integer k, does there exist a path $s \rightsquigarrow t$ of $\leq k$ edges?
    - Open questions: $\boldsymbol{L} \stackrel{?}{=} \boldsymbol{NL}$, $\boldsymbol{NL} \stackrel{?}{=} \boldsymbol{P}$.

- $\boldsymbol{PSPACE}$: the set of decision problems for each of which there exists a deterministic algorithm whose space-efficiency is at-worst polynomial in n.
- $\boldsymbol{NPSPACE}$: the set of decision problems for each of which there exists a non-deterministic algorithm whose space-efficiency is at-worst polynomial in n.
    - Fact: $\boldsymbol{PSPACE = NPSPACE}$.
    - Fact: $\boldsymbol{L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE}$.
        - So, $\boldsymbol{P \neq NP \Leftrightarrow P \subset NP}$.

9.3 The Notion of a Reduction

Given a decision problem, the identification that it belongs to some complexity class $\mathcal{C}$ is a characterization of the computational difficulty, or *hardness*, of that problem.
E.g., if I have a decision problem D, and I am able to say that $D \in \boldsymbol{P}$, then that characterizes that a polynomial-time algorithm suffices for D.
Specifically, $D \in \mathcal{C}$ identifies $\mathcal{C}$ as an upper-bound for the hardness of D. It establishes the worst kind of algorithm we need.

Another way to characterize the hardness of a problem: by comparing it to another problem.
E.g., we know from Lecture 9(a): $ILP \in \boldsymbol{P} \Longrightarrow CIRCUIT - SAT \in \boldsymbol{P}$. This can be taken as a statement that $CIRCUIT - SAT$ is no harder than ILP.
We know also from Lecture 9(a): $CIRCUIT - SAT \in \boldsymbol{P} \Longrightarrow ILP \in \boldsymbol{P}$. Thus, ILP is no harder than $CIRCUIT - SAT$. Taken together then, ILP and $CIRCUIT - SAT$ are equivalently computationally hard.

A statement of the form $A \in \boldsymbol{P} \Longrightarrow B \in \boldsymbol{P}$ is called a *reduction* from B to A.
A reduction from problem B to problem A establishes that:

B is no harder than A

Or equivalently:

A is at least as hard as B.

I find the notation $B \leq A$ for "B reduces to A" to be useful and appropriate.
Warning: the direction of the reduction is important. It is possible, for two problems A, B, that $B \leq A$, but $A \not\leq B$.

There are many different kinds of reductions people have proposed for various kinds of properties.

E.g., suppose I have invented a new padlock, and I want to claim that it is the "best" that is available in the market.

One way I could do this is to reduce the security afforded by every other padlock in the market to the security afforded by my padlock. This would be a reduction from every other padlock to my padlock.

Equivalently, I could reduce the insecurity from using my padlock to the insecurity of every other padlock. This would establish that my padlock is not more insecure than any other padlock.

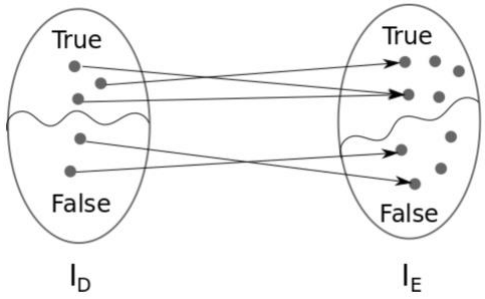
The particular notion of "reduction" I would need to have would be to do with security, and not computational hardness.

The particular kind of reduction that is adopted in our context of decision problems in $\boldsymbol{P}$ and $\boldsymbol{NP}$ is the following.

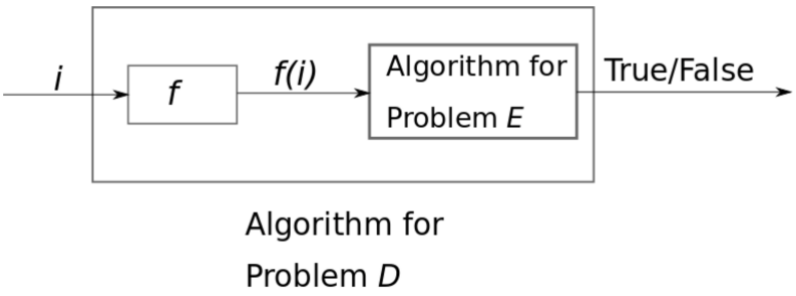
Definition 1. *Suppose D and E are decision problems. Let I_D be the set of all instances of D and I_E the set of all instances of E. We say that D reduces to E, written $D \to E$ or $D \leq E$ if there exists a function $f: I_D \to I_E$ such that*:

(i) $d \in I_D$ *is* ***true*** *if and only if* $f(d) \in I_E$ *is* ***true****, and*

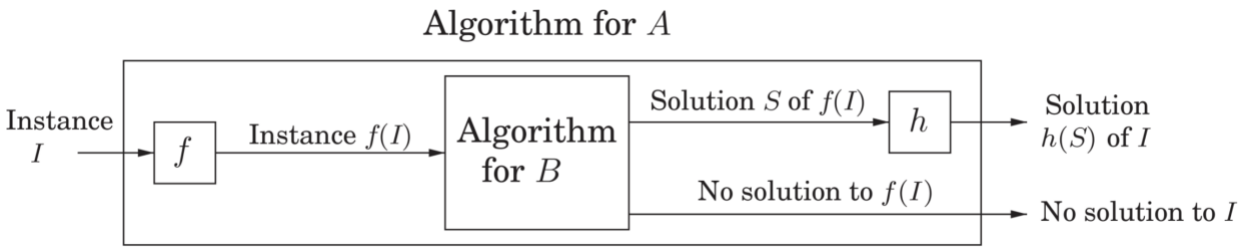
(ii) f *is polynomial-time computable.*



A consequence of $D \leq E$: a polynomial-time algorithm for D given one for E.



Corresponding picture, but for search problems, with the intent of finding a solution, and not just **true** or **false**. The picture is when $A \leq B$.



Their definition requires the existence of not one, but two polynomial-time computable functions:

(i) f, which maps an instance of A to an instance of B such that a is a **true** instance of A if and only if $f(a)$ is a **true** instance of B, and

(ii) A function h, which given input a solution s_b for $f(a)$, has output s_a which is a solution for a.

To recap where we are so far, given a decision problem D:

- $D \in \mathcal{C}$ for some complexity class $\mathcal{C}$ identifies an upper-bound hardness for D, and
- $E \leq D$ identifies a lower-bound hardness for D.
- Note: $D \leq F$ identifies an upper-bound hardness for D as well but ignore that for a moment.

The "$\in \mathcal{C}$" and "$\leq$" notions of hardness have been unified as follows.

Definition 2. *Given a complexity class $\mathcal{C}$, and a notation of a reduction $\leq$, we say that a decision problem D is $\mathcal{C}$-hard (under $\leq$) if for every $C \in \mathcal{C}, C \leq D$.*

For example, for the class $\boldsymbol{NP}$, suppose for every decision problem $N \in \boldsymbol{NP}$, it is the case that $N \leq D$ for some decision problem D. Then, we say that D is $\boldsymbol{NP}$-hard.
So, if a problem is $\boldsymbol{NP}$-hard, it means the problem is at least as hard as every problem in $\boldsymbol{NP}$.

Definition 3. *Given a complexity class $\mathcal{C}$, and a notion of a reduction $\leq$, we say that a decision problem D is $\mathcal{C}$-complete (under $\leq$) if (i) D is $\mathcal{C}$-hard, and (ii) $D \in \mathcal{C}$.*
For example, if some decision problem D is $\boldsymbol{NP}$-hard and in $\boldsymbol{NP}$, then we say that D is $\boldsymbol{NP}$-complete.
So, if a problem is $\boldsymbol{NP}$-complete, then it is hard, but not too hard. That is, it is at least as hard as every problem in $\boldsymbol{NP}$, but still lies within $\boldsymbol{NP}$.

Before we move to concrete examples of problems, following are two useful claims.

Claim 1. *The notion of reduction we have adopted, $\leq$, is transitive. That is, $A \leq B$ and $B \leq C$ implies $A \leq C$.*
Proof. By construction.
$A \leq B$ means that there exists a polynomial-time computable function $f: I_A \to I_B$ such that $a \in A$ is **true** $\Leftrightarrow f(a) \in B$ is **true**. Assume that an algorithm to compute f runs in time $O(n^{k_1})$ for some constant k_1 on input-size n.
Similarly, $B \leq C$ means that there exists a polynomial-time computable function $g: I_B \to I_C$ such that $b \in B$ is **true** $\Leftrightarrow g(b) \in C$ is **true**. Assume that an algorithm to compute g runs in time $O(n^{k_2})$ for some constant k_2 on input-size n.
Consider the composition $g\big(f(\cdot)\big)$. It is a function with domain I_A and codomain I_C. And a is a **true** instance of A if and only if $f(a)$ is a **true** instance of B if and only if $g\big(f(a)\big)$ is a **true** instance of C. So, the "if and only if" property is satisfied.
Also, as our algorithm to compute f runs in time $O(n^{k_1})$, the size of the output from that algorithm is $O(n^{k_1})$. Therefore, there exists an algorithm to compute $g\big(f(\cdot)\big)$ in time $O((n^{k_1})^{k_2}) = O(n^{k_1 \cdot k_2})$. As each of k_1, k_2 is a constant, so it $k_1 \cdot k_2$.
So, $g\big(f(\cdot)\big)$ serves as a proof by construction that $A \leq C$.

Claim 2. *If A is $\boldsymbol{NP}$-hard and $A \leq B$, then B is $\boldsymbol{NP}$-hard.*

Proof. As A is $\boldsymbol{NP}$-hard, given any C in $\boldsymbol{NP}$, $C \leq A$. Also, because $A \leq B$, by Claim 1 above, $C \leq B$. Thus, given any C in $\boldsymbol{NP}$, $C \leq B$. And therefore B is $\boldsymbol{NP}$-hard.

The usefulness of Claim 2 is that to show that a problem B is $\boldsymbol{NP}$-hard, all we need to do is show that some problem A that is $\boldsymbol{NP}$-hard reduces to B.

For example, suppose you know that ILP is $\boldsymbol{NP}$-hard. Then, because $ILP \leq CIRCUIT - SAT$, you can infer that $CIRCUIT - SAT$ is $\boldsymbol{NP}$-hard as well.