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

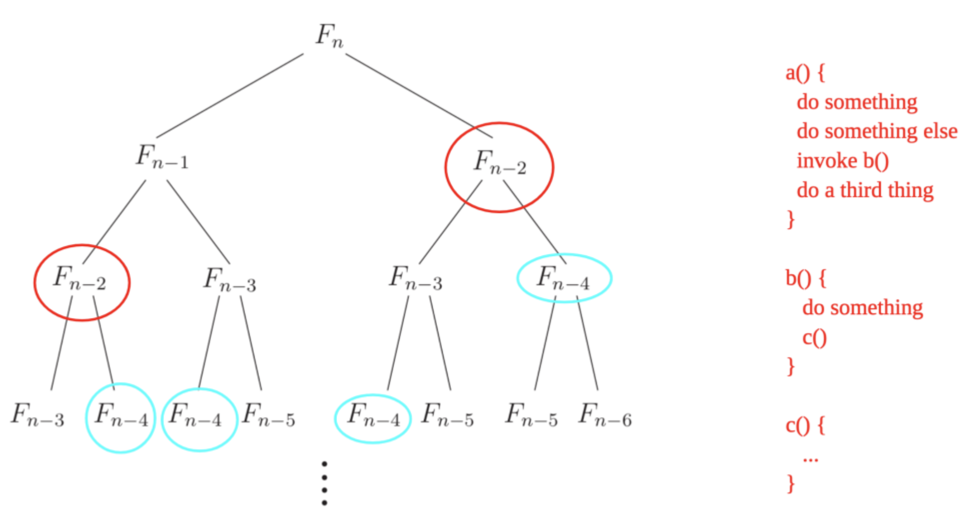
For all $n \in Z_o^+, F(n) \geq \left(\sqrt{2}\right)^n$.

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

$FIB_2(n)$

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

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

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

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

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

$return\ f[n]$

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 $g = O(f)$

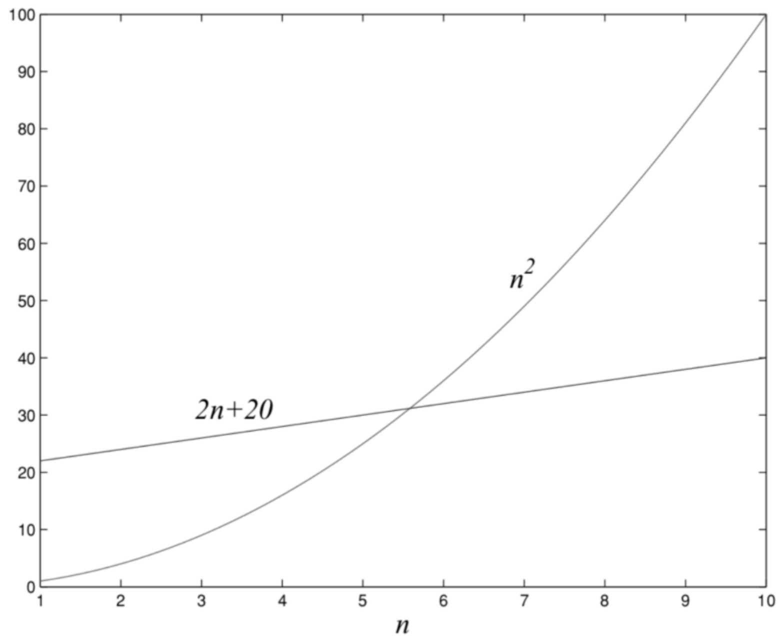
1.2.3 Definition 3 (Θ)

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 $T : \{0, 1\} \times \{0, \dots, 9\} \times \{0, \dots, 9\} \rightarrow \{0, 1\} \times \{0, \dots, 9\}$:

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:

$$C \rightarrow 1\ 1\ 1\ 0\ 0\ 0$$

$$1 \rightarrow 0\ 0\ 7\ 8\ 1\ 4$$

$$2 \rightarrow 0\ 9\ 3\ 4\ 0\ 4$$

$$T \rightarrow 1\ 0\ 1\ 2\ 1\ 8$$

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:

- $Let\ \#\ bits\ to\ encode\ each\ of\ x\ and\ y = n$
- $\#\ recursive\ calls = \Theta(\mathrm{n})$
- $In\ each\ call$:
  - 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 $x \,\epsilon\, Z_o^+, y \,\epsilon\, Z^+$, the pair $\langle q, r \rangle$ where $q \,\epsilon\, Z_o^+$, $r \,\epsilon\, \{0, 1, \ldots, y - 1\}$ of x divided by y are those that satisfy:

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

Claim 1: For every $x \,\epsilon\, Z_o^+$, $y \,\epsilon\, Z^+$, $\langle q, r \rangle$ 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:** $x \equiv y\ (mod\ N)\ if\ N\ divides\ x - y$. "$\equiv$" 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:

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

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

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

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

All arithmetic performed modulo 2^n:

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

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

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

2.1.2 Modular Addition, Subtraction

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

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

- 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:

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

- 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: $x, y \in \mathbb{Z}^+, x \geq y \rightarrow \gcd(x, y) = \gcd\ (x\ mod\ y, y)$

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

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: $a \geq b \geq 0 \rightarrow a \ mod \ b < a/2$

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

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

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

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

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

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

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

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

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 $a \epsilon \{1, \dots, N-1\}$.

Proof: If there exists no $b \epsilon \{1, \dots, N-1\}$ 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

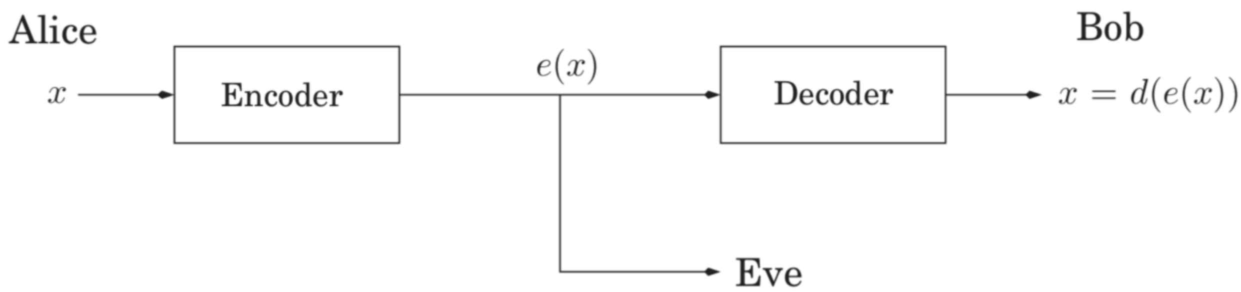
In a Bernoulli trial if $\Pr\{success\} = p$, then $expected\ \#\ trials\ to\ see\ a\ success = 1/p$

- 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:

- $e(\cdot) = d(\cdot) \to Alice\ and\ Bob\ both\ know\ e(\cdot)\ and\ d(\cdot)$
    - 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 $f: \{0, 1, \dots, N-1\} \to \{0, 1, \dots, N-1\}$ 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
OvDHY6cNcN4vpjc0fZbieeJWCOwjFV8qt5VZDTdvtLJSBilH1j
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.

- $CreateNew(\ldots)$: 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

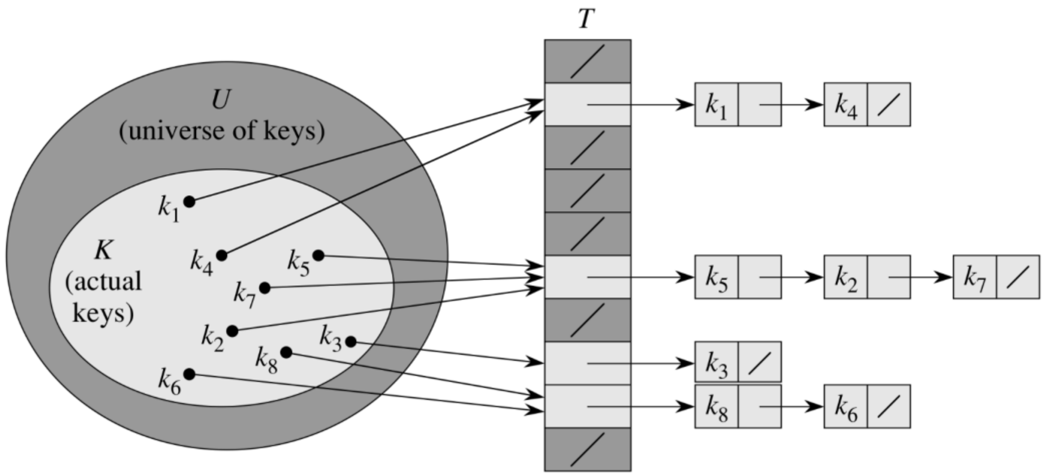
- Time Efficiency: $Search$ to run in time $\approx \Theta(1)$.
- Space Efficiency: If we seek to store n records, space we should have to allocate $\approx \Theta(n)$.

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: $U \rightarrow \{0, \dots, m-1\}$.
- 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, $h\colon U \to [0, m)$
  Create a table $T[0, \ldots, m-1]$
  Each $T[i] \leftarrow$ NEWLIST( )
  Store h as part of T
  return T

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

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)

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 in expectation, $length\ of\ each\ chain = n/m$.
  - 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 $U \subseteq \{0, 1, \ldots, p-1\}$.
- For $a \in \{1, 2, \ldots, p-1\}, b \in \{0, 1, \ldots, p-1\}$, let $h_{a,b}(k) = \big((ak + b)\ mod\ p\big)\ mod\ m$.
- Adopt $\mathcal{H} = \Big\{h_{a,b} \mid a \in \{1, 2, \ldots, p-1\}, b \in \{0, 1, \ldots, p-1\}\Big\}$.

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

For the following claims, adopt:

- $k, l \in \{0, 1, \ldots, p-1\}, k \neq l$.
- 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:*

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

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

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

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. *

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

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

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

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 $k, l \epsilon U$:

- 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:

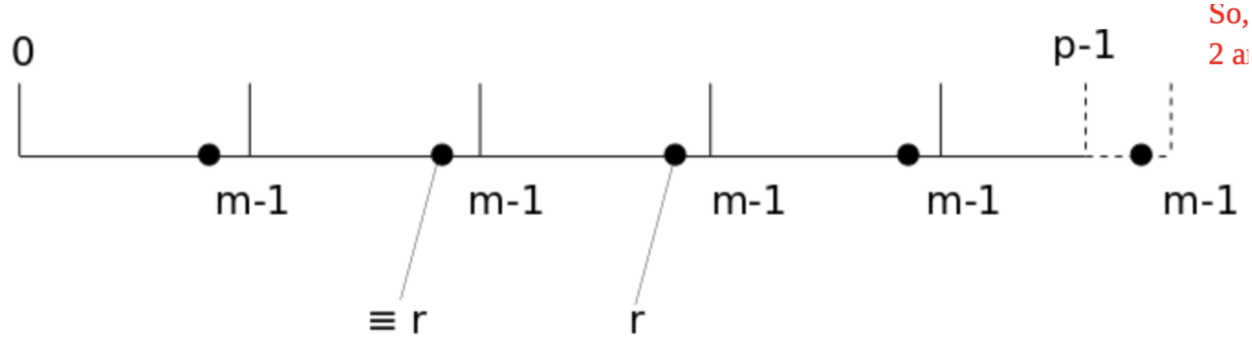
- Suppose each of a, b is chosen uniformly at random.
- Then, for $k \neq l$, $\Pr\left\{h_{a,b}(k) = h_{a,b}(l)\right\} = \Pr\{r \equiv s\ (mod\ m)\}$, r, s chosen uniformly.

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

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

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

So,
2 a



*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:

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

**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.

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

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

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

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

...

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

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

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

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

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 $f: \mathbb{N} \to \mathbb{R}^+$, with constants $a, b, d \in \mathbb{R}, a > 0, b > 1, d \geq 0$:

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

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 $A[1, \dots, n]$ whose items are drawn from a totally ordered set, return the sorted permutation of A.

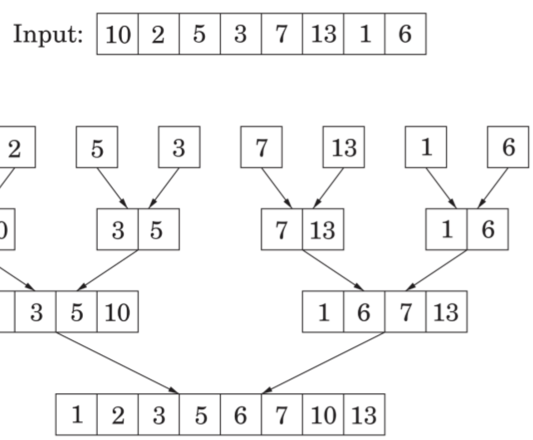
```
function mergesort(a[1...n])
Input: An array of numbers a[1...n]
Output: A sorted version of this array

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.*

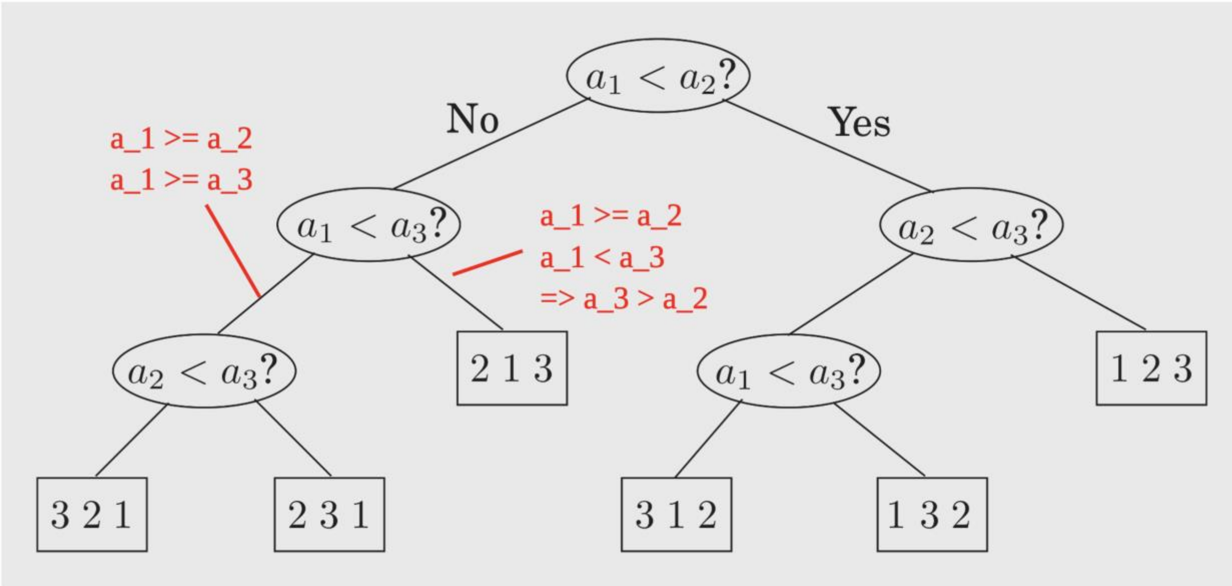
Proof. Induction on # $recursive\ calls$.

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

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

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 $n = 30$, $8 \leq rank[v] \leq 22$.

Then, recurrence for running time of $selection$:

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

To pick such a "good" v:

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

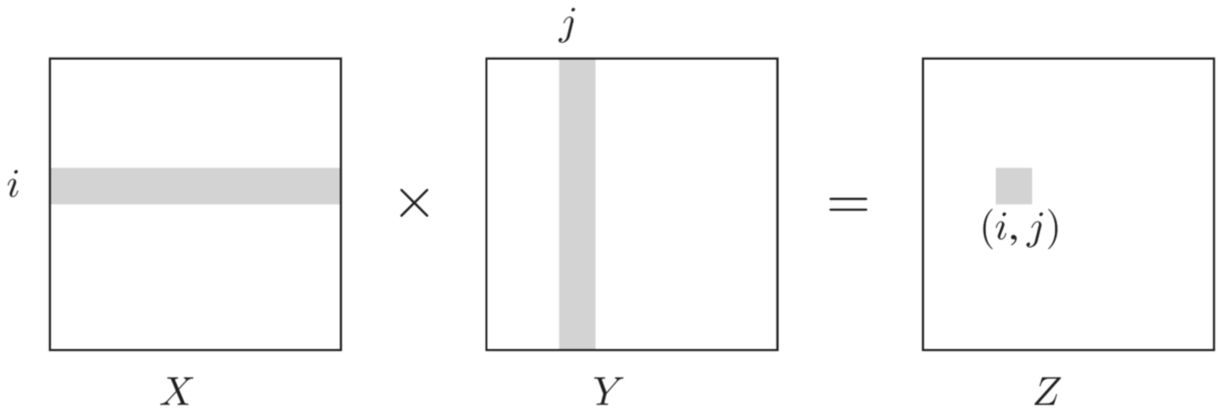
$$\Pr\{we\ pick \text{ "good" v in a trial}\} = \frac{1}{2}$$

So # trials <u>in expectation</u> 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

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) & & \end{aligned}$$

Now, recurrence for # scalar multiplications and additions:

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

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