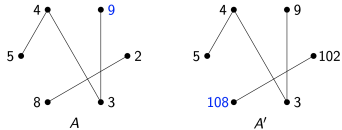


- **algorithm** → a well-defined procedure for finding the correct solution to the input

- **worst-case correctness** → correct on *every valid input*
- other types of correctness: correct on random input/with high probability/approximately correct
- **efficiency / running time** → measures the number of steps executed by an algorithm as a function of the *input size* (depends on computational model used)
 - number input: typically the length of binary representation
 - **worst-case** running time → *max* number of steps executed when run on an input of size n

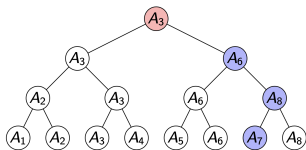
- algorithm can **compare** any two elements in one time unit
($x > y, x < y, x = y$)
- running time = number of comparisons made
- array can be manipulated at no cost

- *problem:* find the largest element in an array A of n distinct elements
- *proof* that $n - 1$ comparisons are needed:
 - fix an algorithm M that solves the Maximum problem on all inputs using $< n - 1$ comparisons. construct graph G where nodes i and j are adjacent iff M compares i and j .



- M cannot differentiate A and A' .
- **adversary argument** \rightarrow inputs are decided such that they have different solutions

- *problem*: find the second largest element in $< 2n - 3$ comparisons ($2 \times \text{Maximum} \Rightarrow (n-1) + ((n-1)-1) = 2n-3$)
- *solution*: **knockout tournament** $\Rightarrow n + \lceil \lg n \rceil - 2$



1. bracket system: $n - 1$ matches
 - every non-winner has lost exactly once
2. then compare the elements that have lost to the largest
 - the second-largest element must have lost to the winner
 - compares $\lceil \lg n \rceil$ elements that have lost to the winner using $\lceil \lg n \rceil - 1$ comparisons

- there is a sorting algorithm that requires $\leq n \lg n - n + 1$ comparisons.
- *proof*: every sorting algorithm must make $\geq \lg(n!)$ comparisons.
 1. let set \mathcal{U} be the set of all permutations of the set $\{1, \dots, n\}$ that the adversary could choose as array A .
 $|\mathcal{U}| = n!$
 2. for each query "is $A_i > A_j$?",
 if $\mathcal{U}_{yes} = \{A \in \mathcal{U} : A_i > A_j\}$ is of size $\geq |\mathcal{U}|/2$, set $\mathcal{U} := \mathcal{U}_{yes}$. else: $\mathcal{U} := \mathcal{U} \setminus \mathcal{U}_{yes}$
 3. the size of \mathcal{U} decreases by at most half with each comparison
 4. for $> \lg(n!)$ comparisons, \mathcal{U} will still contain at least 2 permutations

$$\begin{aligned} n! &\geq \left(\frac{n}{e}\right)^n \\ \Rightarrow \lg(n!) &\geq n \lg\left(\frac{n}{e}\right) = n \lg n - n \lg e \\ &\approx n \lg n - 1.44n \end{aligned}$$

⇒ roughly $n \lg n$ comparisons are **required** and **sufficient** for sorting n numbers

- input: string of n bits
- each query: find out **one bit** of the string
- n queries are **necessary** and **sufficient** to check if the input string is all 0s.

- input: (symmetric) adjacency matrix of an n -node undirected graph
- each query: find out if an edge is present between two chosen nodes
- *proof*: $\binom{n}{2}$ queries are necessary to decide whether the graph is connected or not
 1. suppose M is an algorithm making $\leq \binom{n}{2}$ queries.
 2. whenever M makes a query, the algorithm tries not adding this edge, but adding all remaining unqueried edges.
 - 2.1. if the resulting graph is connected, M replies 0 (i.e. edge does not exist)
 - 2.2. else: replies 1 (edge exists)
 3. after $< \binom{n}{2}$ queries, at least one entry of the adjacency matrix is unqueried.

- **algorithm** → a *finite* sequence of well-defined instructions to solve a given computational problem
- **runtime** → measured in number of instructions taken in **word-RAM** model
 - operators, comparisons, if, return, etc

upper bound (\leq): $f(n) = O(g(n))$
 if $\exists c > 0, n_0 > 0$ such that $\forall n \geq n_0, \quad 0 \leq f(n) \leq cg(n)$

lower bound (\geq): $f(n) = \Omega(g(n))$
 if $\exists c > 0, n_0 > 0$ such that $\forall n \geq n_0, \quad 0 \leq cg(n) \leq f(n)$

tight bound: $f(n) = \Theta(g(n))$
 if $\exists c_1 > 0, c_2 > 0, n_0 > 0$ such that
 $\forall n \geq n_0, \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

o notation ($<$): $f(n) = o(g(n))$
 if $\forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0,$
 $0 \leq f(n) < cg(n)$

ω -notation ($>$): $f(n) = \omega(g(n))$
 if $\forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0,$
 $0 \leq cg(n) < f(n)$

- *upper*: $O(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \mid \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$
- *lower*: $\Omega(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \mid \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$

Proof. that $2n^2 = O(n^3)$
 let $f(n) = 2n^2$. then $f(n) = 2n^2 \leq n^3$ when $n \geq 2$.
 set $c = 1$ and $n_0 = 2$.
 we have $f(n) = 2n^2 \leq c \cdot n^3$ for $n \geq n_0$. □

Assume $f(n), g(n) > 0$.

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= 0 \Rightarrow f(n) = o(g(n)) \\ \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &< \infty \Rightarrow f(n) = O(g(n)) \\ 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &< \infty \Rightarrow f(n) = \Theta(g(n)) \\ \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &> 0 \Rightarrow f(n) = \Omega(g(n)) \\ \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \infty \Rightarrow f(n) = \omega(g(n))\end{aligned}$$

Proof. using delta epsilon definition

$$\overline{\Theta(g(n))} = \overline{O(g(n)) \cap \Omega(g(n))}$$

- **transitivity** - applies for $O, \Theta, \Omega, o, \omega$
 $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
 - **reflexivity** - for O, Ω, Θ , $f(n) = O(f(n))$
 - **symmetry** - $f(n) = O(g(n)) \iff g(n) = \Theta(f(n))$
 - **complementarity** -
 - $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
 - $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$
- insertion sort: $O(n^2)$ with worst case $\Theta(n^2)$

$$\log \log n < \log n < (\log n)^k < n^k < k^n$$

Iterative Algorithms

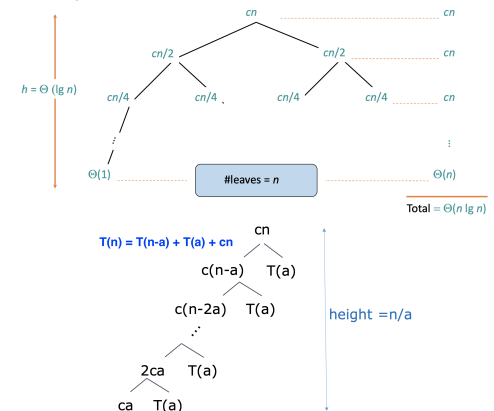
- *initialisation* - true before the first iteration
- *maintenance* - if true before an iteration, remains true at the beginning of the next iteration
- *termination* - true at the end

powering a number

- **problem:** compute $f(n, m) = a^n \pmod m$ for all integer n, m
- **observation:** $f(x + y, m) = f(x, m) * f(y, m) \pmod m$
- **naive solution:** recursively compute and combine $f(n - 1, m) * f(1, m) \pmod m$

- $T(n) = T(n-1) + T(1) + \Theta(1) \Rightarrow T(n) = \Theta(n)$
- **better solution:** divide and conquer
 - divide: trivial
 - conquer: recursively compute $f(\lfloor n/2 \rfloor, m)$
 - combine:
 - $f(n, m) = f(\lfloor n/2 \rfloor, m)^2 \pmod m$ if n is even
 - $f(n, m) = f(1, m) * f(\lfloor n/2 \rfloor, m)^2 \pmod m$ if odd
- $T(n) = T(n/2) + \Theta(1) \Rightarrow \Theta(\log n)$

for a sub-problems of size $\frac{n}{b}$ where $f(n)$ is the time to divide and combine, $T(n) = aT(\frac{n}{b}) + f(n)$

$$\text{total} = \text{height} \times \text{number of leaves}$$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

harmonic series: $\sum_{k=1}^{\infty} \frac{1}{k} \approx \ln k = \Theta(\lg n)$

1. guess that $T(n) = O(f(n))$.
i.e. $\exists c$ such that $T(n) \leq c \cdot f(n)$, for $n \geq n_0$.
2. verify by induction:
 - 2.1. set $c = \max\{2, q\}$ and $n_0 = 1$
 - 2.2. base case ($n = n_0 = 1$)
 - 2.3. recursive case ($n > 1$):
 - by strong induction, assume $T(k) = c \cdot f(k)$ for $n > k > 1$
 - $T(n) = \langle \text{recurrence} \rangle \dots \leq c \cdot f(n)$
 - 2.4. hence $T(n) \leq c \cdot f(n)$ for $n \geq 1$.

example

$$T(n) = 4T(n/2) + n^2/\lg n \Rightarrow \Theta(n^2 \lg \lg n)$$

Proof. $T(n) = 4T(n/2) + \frac{n^2}{\lg n}$

$$= 4(4T(n/4) + \frac{(n/2)^2}{\lg n - \lg 2}) + \frac{n^2}{\lg n}$$

$$= 16T(n/4) + \frac{n^2}{\lg n - \lg 2} + \frac{n^2}{\lg n}$$

$$= \sum_{k=1}^{\lg n} \frac{n^2}{\lg n - k}$$

$$= n^2 \lg \lg n \text{ by approx. of harmonic series } (\sum \frac{1}{k})$$

04. AVERAGE-CASE ANALYSIS & RANDOMISED ALGORITHMS

Quicksort Analysis

- divide & conquer, linear-time $\Theta(n)$ partitioning subroutine
- assume we select the first array element as pivot
- if the pivot produces subarrays of size j and $(n - j - 1)$, then $T(n) = T(j) + T(n - j - 1) + \Theta(n)$

time analysis

- worst-case:** $T(n) = T(0) + T(n - 1) + \Theta(n) \Rightarrow \Theta(n^2)$
- average case** $A(n) \rightarrow$ expected running time when the input is chosen uniformly at random from the set of all $n!$ permutations
- average case,** $A(n) = \frac{1}{n!} \sum_{\pi} Q(\pi)$ where $Q(\pi)$ is the time complexity when the input is permutation π .

Proof. for quicksort, $A(n) = O(n \log n)$

let $P(i)$ be the set of all those permutations of elements $\{e_1, e_2, \dots, e_n\}$ that begins with e_i .

Let $G(n, i)$ be the average running time of quicksort over $P(i)$. Then

$$G(n) = A(i - 1) + A(n - i) + (n - 1).$$

$$A(n) = \frac{1}{n} \sum_{i=1}^n G(n, i)$$

$$= \frac{1}{n} \sum_{i=1}^n (A(i - 1) + A(n - i) + (n - 1))$$

$$= \frac{2}{n} \sum_{i=1}^n A(i - 1) + n - 1$$

$$= \dot{O}(n \log n) \text{ by taking it as area under integration}$$

quicksort vs mergesort

	average	best	worst
quicksort	$1.39n \lg n$	$n \lg n$	$n(n - 1)$
mergesort	$n \lg n$	$n \lg n$	$n \lg n$

- disadvantages of mergesort:
 - overhead of temporary storage
 - cache misses
- advantages of quicksort
 - in place
 - reliable (as $n \uparrow$, chances of deviation from avg case \downarrow)
- issues with quicksort
 - distribution-sensitive** \rightarrow time taken depends on the initial (input) permutation

Randomised Algorithms

- randomised algorithms** \rightarrow output and running time are **functions** of the **input** and **random bits chosen**
 - vs non-randomised: output & running time are functions of the *input only*
- randomised quicksort:** choose pivot at random
 - probability that the runtime of *randomised* quicksort exceeds average by $x\%$ = $n^{-\frac{x}{100} \ln \ln n}$
 - P(time takes at least double of the average) = 10^{-15}
 - distribution insensitive

Randomised Quicksort Analysis

$$T(n) = n - 1 + T(q - 1) + T(n - q)$$

Let $A(n) = \mathbb{E}[T(n)]$ where the expectation is over the randomness in expectation.

Taking expectations and applying linearity of expectation:

$$A(n) = n - 1 + \frac{1}{n} \sum_{q=1}^n (A(q - 1) + A(n - q))$$

$$= n - 1 + \frac{2}{n} \sum_{q=1}^{n-1} A(q)$$

$$A(n) = n \log n \quad \Rightarrow \text{same as average case quicksort}$$

Randomised Quickselect

- $O(n)$ to find the k^{th} smallest element
- randomisation: unlikely to keep getting a bad split

Types of Randomised Algorithms

- randomised **Las Vegas** algorithms
 - output is always correct
 - runtime is a *random variable*
 - e.g. randomised quicksort
- randomised **Monte Carlo** algorithms
 - output may be incorrect with some small probability
 - runtime is *deterministic*

examples

- smallest enclosing circle:* given n points in a plane, compute the smallest radius circle that encloses all n points
 - best **deterministic** algorithm: $O(n)$, but complex
 - las vegas: average $O(n)$, simple solution
- minimum cut:* given a connected graph G with n vertices and m edges, compute the smallest set of edges whose removal would disconnect G .
 - best **deterministic** algorithm: $O(mn)$
 - monte carlo:** $O(m \log n)$, error probability n^{-c} for any c
- primality testing:* determine if an n bit integer is prime
 - best **deterministic** algorithm: $O(n^6)$
 - monte carlo:** $O(kn^2)$, error probability 2^{-k} for any k

Geometric Distribution

Let X be the number of trials repeated until success.

X is a random variable and follows a geometric distribution

with probability p .

$$\text{Expected number of trials, } E[X] = \frac{1}{p}$$

$$Pr[X = k] = q^{k-1}p$$

Linearity of Expectation

For any two events X, Y and a constant a ,

$$E[X + Y] = E[X] + E[Y]$$

$$E[aX] = aE[X]$$

Coupon Collector Problem

n types of coupon are put into a box and randomly drawn with replacement. What is the expected number of draws needed to collect at least one of each type of coupon?

- let T_i be the time to collect the i -th coupon after the $i - 1$ coupon has been collected.
 - Probability of collecting a new coupon, $p_i = \frac{(n-(i-1))}{n}$
 - T_i has a **geometric distribution**
 - $E[T_i] = 1/p_i$

- total number of draws, $T = \sum_{i=1}^n T_i$

- $E[T] = E[\sum_{i=1}^n T_i] = \sum_{i=1}^n E[T_i]$ by linearity of expectation

$$= \sum_{i=1}^n \frac{n}{n-(i-1)} = n \cdot \sum_{i=1}^n \frac{1}{i} = \Theta(n \lg n)$$

helpful approximations

harmonic number, $H_n = \sum_{k=1}^n \frac{1}{k} = \Theta(\lg n)$