CS3230
AY21/22 SEM 2
github/jovyntls

01. COMPUTATIONAL MODELS

- **algorithm** $\rightarrow$ a well-defined procedure for finding the correct solution to the input
- **correctness**
  - **worst-case correctness** $\rightarrow$ correct on *every valid input*
  - other types of correctness: correct on random input/with high probability/approximately correct
- **efficiency** / **running time** $\rightarrow$ 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 $\rightarrow$ *max* number of steps executed when run on an input of size n

adversary argument $\rightarrow$
inputs are decided such that they have different solutions

Comparison Model

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

Decision Tree

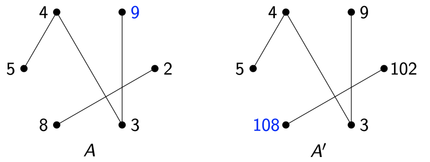
- each node is a comparison
- each branch is an outcome of the comparison
- each leaf is a class label (decision after *all* comparisons)
- **worst-case** runtime = height of tree
- # of leaves = # of permutations $\Rightarrow \lg(n!) = \Theta(n \lg n)$

`Max` Problem

problem: find largest element in array A of n distinct elements

Proof. $n-1$ comparisons are needed

fix an algorithm M that solves the `Max` problem on all inputs using $< n-1$ comparisons. construct graph G where nodes i and j are adjacent iff M compares i & j.

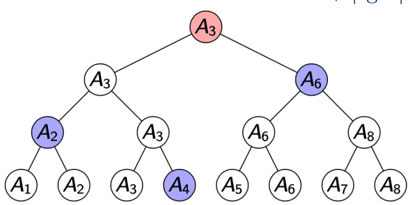


M cannot differentiate A and A'.

Second Largest Problem

problem: find the second largest element in $< 2n-3$ comparisons (2x 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 2nd 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

Sorting

Claim. 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 $\{1, \ldots, n\}$ 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: $\mathcal{U} := \mathcal{U} \backslash \mathcal{U}_{yes}$
3. the size of $\mathcal{U}$ decreases by at most half with each comparison
4. with $< \lg(n!)$ comparisons, $\mathcal{U}$ will still contain at least 2 permutations

$$n! \geq (\tfrac{n}{e})^n$$
$$\Rightarrow \lg(n!) \geq n \lg(\tfrac{n}{e}) = n \lg n - n \lg e$$
$$\approx n \lg n - 1.44n$$

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

String Model

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.
- **query complexity** $\rightarrow$ number of bits of the input string queried by the algorithm
- **evasive** $\rightarrow$ a problem requiring n query complexity

Graph Model

input	(symmetric) adjacency matrix of an n-node undirected graph
each query	find out if an edge is present between two chosen nodes (one entry of G)

- **evasive** $\rightarrow$ requires $\binom{n}{2}$ queries
- *Proof.* determining whether the graph is connected is evasive (requires $\binom{n}{2}$ queries)
  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.

□

02. ASYMPTOTIC ANALYSIS

- **algorithm** $\rightarrow$ a *finite* sequence of well-defined instructions to solve a given computational problem
- **word-RAM model** $\rightarrow$ runtime is the total number of instructions executed
  - operators, comparisons, if, return, etc
  - each instruction operates on a *word* of data (limited size) $\Rightarrow$ fixed constant amount of time

Asymptotic Notations

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

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

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

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

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

Limits

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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad \Rightarrow f(n) = o(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = O(g(n))$$
$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = \Theta(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \qquad \Rightarrow f(n) = \Omega(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \Rightarrow f(n) = \omega(g(n))$$

Proof. using delta epsilon definition

Properties of Big O

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

- **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 $O, \Omega, \Theta,\quad f(n) = O(f(n))$
- **symmetry** - $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
- **complementarity** -
  - $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
  - $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$
- **misc**
  - if $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$
  - if $f(n) = o(g(n))$, then $f(n) = O(g(n))$

$\boxed{\log \log n < \log n < (\log n)^k < n^k < k^n}$

insertion sort: $O(n^2)$ with worst case $\Theta(n^2)$

03. ITERATION, RECURSION, DIVIDE-AND-CONQUER

Iterative Algorithms

- **iterative** $\rightarrow$ loop(s), sequentially processing input elements
- **loop invariant** implies correctness if
  - *initialisation* - true before the first iteration of the loop
  - *maintenance* - if true before an iteration, it remains true at the beginning of the next iteration
  - *termination* - true when the algorithm terminates

examples

- **insertionSort**: with loop variable as j, $A[1..J-1]$ is sorted.
- **selectionSort**: with loop variable as j, the array $A[1..j-1]$ is sorted and contains the $j-1$ smallest elements of A.
- **Misra-Gries algorithm** (determines which bit occurs more in an n-bit array A):
  - if there is an equal number of 0's and 1's, then $id = \perp$ and $count = 0$
  - if $z \in \{0, 1\}$ is the majority element, then $id = z$ and $count$ equals the difference between the count of the bits.

Divide-and-Conquer

powering a number

problem: compute $f(n, m) = a^n \pmod m$ for all $n, m \in \mathbb{Z}$

- 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
  - $T(n) = T(n/2) + \Theta(1) \Rightarrow \Theta(\log n)$

Solving Recurrences

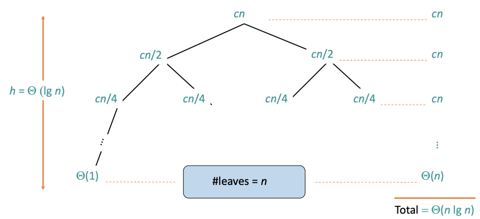
for a sub-problems of size $\frac{n}{b}$ where $f(n)$ is the time to divide and combine,

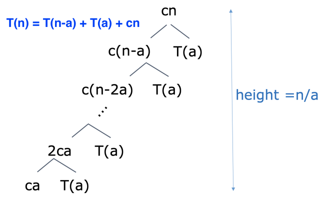
$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

Recursion tree

total = height $\times$ number of leaves

- each node represents the cost of a single subproblem
- height of the tree = longest path from root to leaf





Master method

$a \geq 1, b > 1,$ and f is asymptotically positive

$T(n) = aT(\frac{n}{b}) + f(n) =$

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

three common cases

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ by n^ϵ factor.
   - then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a}\log^k n)$ for some $k \geq 0$,
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.
   - then $T(n) = \Theta(n^{\log_b a}\log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
   - and $f(n)$ satisfies the **regularity condition**
     - $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n,
     - this guarantees that the sum of subproblems is smaller than $f(n)$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ by n^ϵ factor
   - then $T(n) = \Theta(f(n))$.

Substitution method

1. guess that $T(n) = O(f(n))$.
2. verify by induction:
   - 2.1. to show that for $n \geq n_0, T(n) \leq c \cdot f(n)$
   - 2.2. set $c = \max\{2, q\}$ and $n_0 = 1$
   - 2.3. verify base case(s): $T(n_0) = q$
   - 2.4. recursive case ($n > n_0$):
     - by strong induction, assume $T(k) \leq c \cdot f(k)$ for $n > k \geq n_0$
     - T(n) = <recurrence> ... $\leq c \cdot f(n)$
   - 2.5. hence $T(n) = O(f(n))$.

! may not be a tight bound!

example

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

$$\begin{aligned} T(n) &= 4T(n/2) + \frac{n^2}{\lg n} \\ &= 4(4T(n/4) + \frac{(n/2)^2}{\lg n - lg2}) + \frac{n^2}{\lg n} \\ &= 16T(n/4) + \frac{n^2}{\lg n - \lg 2} + \frac{n^2}{\lg n} \\ &= \sum_{k=1}^{\lg n} \frac{n^2}{\lg n - k} \\ &= n^2 \lg\lg n \text{ by approx. of harmonic series } (\textstyle\sum \frac{1}{k}) \end{aligned}$$

Proof. $T(n) = 4T(n/2) + n \Rightarrow O(n^2)$

To show that for all $n \geq n_0, T(n) \leq c_1 n^2 - c_2 n$

1. Set $c_1 = q + 1, c_2 = 1, n_0 = 1$.
2. Base case ($n = 1$): subbing into $c_1 n^2 - c_2 n$, $T(1) = q \leq (q+1)(1)^2 - (1)(1)$
3. Recursive case ($n > 1$):

- by strong induction, assume $T(k) \leq c_1 \cdot k^2 - c_2 \cdot k$ for all $n > k \geq 1$
- $T(n) = 4T(n/2) + n$

$$\begin{aligned} &= 4(c_1(n/2)^2 - c_2(n/2)) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n + (1 - c_2)n \\ &= c_1 n^2 - c_2 n \quad \text{since } c_2 = 1 \Rightarrow 1 - c_2 = 0 \end{aligned}$$

□

04. AVERAGE-CASE ANALYSIS & RANDOMISED ALGORITHMS

- **average case** $A(n) \to$ expected running time when the input is chosen uniformly at random from the set of all $n!$ permutations
  - $A(n) = \frac{1}{n!}\sum_\pi Q(\pi)$ where $Q(\pi)$ is the time complexity when the input is permutation π.
  - $A(n) = \underset{x \sim \mathcal{D}_n}{\mathbb{E}}$ [Runtime of Alg on x]
    - $\mathbb{E}_{x \sim \mathcal{D}_n}$ is a probability distribution on U restricted to inputs of size n.

Quicksort Analysis

- divide & conquer, linear-time $\Theta(n)$ partitioning subroutine
- assume we select the first array element as pivot
- $T(n) = T(j) + T(n - j - 1) + \Theta(n)$
  - if the pivot produces subarrays of size j and $(n - j - 1)$
- **worst-case**: $T(n) = T(0) + T(n-1) + \Theta(n) \Rightarrow \Theta(n^2)$

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

$$\begin{aligned} G(n) &= A(i-1) + A(n-i) + (n-1). \\ A(n) &= \frac{1}{n}\sum_{i=1}^n G(n,i) \\ &= \frac{1}{n}\sum_{i=1}^n (A(i-1) + A(n-i) + (n-1)) \\ &= \frac{2}{n}\sum_{i=1}^n A(i-1) + n - 1 \\ &= O(n \log n) \text{ by taking it as area under integration} \end{aligned}$$

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** $\to$ time taken depends on the initial (input) permutation

Randomised Algorithms

- **randomised algorithms** $\to$ 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*
- expected running time = worst-case running time = $E(n) = \max_{\text{input } x \text{ of size } n} \mathbb{E}[\text{Runtime of RandAlg on } x]$
- **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:

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

$A(n) = n\log n \quad \Rightarrow$ 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 quickselect
- 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.

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

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

05. HASHING

Dictionary ADT

- different types:
  - **static** - fixed set of inserted items; only care about queries
  - **insertion-only** - only insertions and queries
  - **dynamic** - insertions, deletions, queries
- implementations
  - sorted list (static) - $O(\log N)$ query
  - balanced search tree (dynamic) - $O(\log N)$ all operations
  - direct access table
    - × needs items to be represented as non-negative integers (**prehashing**)
    - × huge space requirement
- using $\mathcal{H}$ for dictionaries: need to store both the hash table and the matrix A.
  - additional storage overhead $= \Theta(\log N \cdot \log|U|)$, if $M = \Theta(N)$
  - other universal hashing constructions may have more efficient hash function evaluation
- *associative array* - has both key and value (dictionary in this context has only key)

Hashing

- **hash function**, $h : U \to \{1, \dots, M\}$ gives the location of where to store in the hash table
  - notation: $[M] = \{1, \dots, M\}[M] = \{1, \dots, M\}$
  - storing N items in hash table of size M
- **collision** $\to$ for two different keys x and y, $h(x) = h(y)$
  - resolve by **chaining**, **open addressing**, etc
- desired properties
  - ✓ minimise collisions - `query(x)` and `delete(x)` take time $\Theta(|h(x)|)$
  - ✓ minimise storage space - aim to have $M = O(N)$
  - ✓ function h is easy to compute (assume constant time)
- if $|U| \geq (N-1)M + 1$, for any $h : U \to [M]$, there is a set of N elements having the same hash value.
  - *Proof*: pigeonhole principle
- use **randomisation** to overcome the adversary

- e.g. randomly choose between two *deterministic* hash functions h_1 and h_2
$\Rightarrow$ for any pair of keys, with probability $\geq \frac{1}{2}$, there will be no collision

Universal Hashing

Suppose $\mathcal{H}$ is a set of hash functions mapping U to $[M]$.

$$\mathcal{H} \text{ is } \textbf{universal} \text{ if } \forall x \neq y,\ \frac{|h\in\mathcal{H}: h(x)=h(y)|}{|H|} \leq \frac{1}{M}$$
$$\text{or } \Pr_{h\sim\mathcal{H}}[h(x)=h(y)] \leq \frac{1}{M}$$

- aka: for any $x\neq y$, if h is chosen uniformly at random from a universal $\mathcal{H}$, then there is at most $\frac{1}{M}$ probability that $h(x)=h(y)$
- probability where h is sampled uniformly from $\mathcal{H}$
- aka: for any $x \neq y$, the fraction of hash functions with collisions is at most $\frac{1}{M}$.

Properties of universal hashing

Collision Analysis

- for any N elements $x_1,\dots,x_N\in\mathcal{U}$, the **expected number of collisions** between x_N and other elements is $< N/M$.
  - it follows that for K operations, the expected cost of the last operation is $< K/M = O(1)$ if $M > K$.

Proof. by definition of Universal Hashing, each element $x_1,\dots,x_{N-1}\in\mathcal{U}$ has at most $\frac{1}{M}$ probability of collision with x_N (over random choice of h).
by indicator r.v., $E[A_i]=P(A_i=1)\leq\frac{1}{M}$. expected number of collisions = $(N-1)\cdot\frac{1}{M} < \frac{N}{M}$.

- if $x_1,\dots,x_N$ are added to the hash table, and $M>N$, the expected **number of pairs** (i,j) with collisions is $<2N$.

Proof. let A_{ij} be an indicator r.v. for collision.
$$\mathbb{E}[\sum_{1\leq i,j\leq N} A_{ij}] = \sum_{i=1}^{N}\mathbb{E}[A_{ii}] + \sum_{i\neq j}\mathbb{E}[A_{ij}]$$
$$\leq N\cdot 1 + N(N-1)\cdot\frac{1}{M} < 2N$$

Expected Cost

- for any sequence of N operations, if $M>N$, then the **expected total cost** for executing the sequence is $O(N)$.

Proof. linearity of expectation; sum up expected costs

Construction of Universal Family

Obtain a universal family of hash functions with $M=O(N)$.
- Suppose U is indexed by u-bit strings and $M=2^m$.
- For any $m\times u$ binary matrix A, $h_A(x)=Ax \pmod 2$
  - each element `x => x % 2`
  - x is a $u\times 1$ matrix $\Rightarrow Ax$ is $m\times 1$
- *Claim*: $\{h_A : A\in\{0,1\}^{m\times u}\}$ is universal
- e.g. $U=\{00,01,10,11\}, M=2$
  - h_{ab} means $A=[a\ b]$

	00	01	10	11
h_{00}	0	0	0	0
h_{01}	0	1	0	1
h_{10}	0	0	1	1
h_{11}	0	1	1	0

Proof. Let $x\neq y$. Let $z=x-y$. We know $z\neq 0$.
Collision: $P(Ax=Ay)=P[A(x-y)=0]=P(Az=0)$.
To show $P(Az=0)\leq\frac{1}{M}$.

Special case - Suppose z is 1 at the i-th coordinate but 0 everywhere else. Then Az is the i-th column of A. Since the i-th column is uniformly random, $P(Az=0)=\frac{1}{2^m}=\frac{1}{M}$.

General case - Suppose z is 1 at the i-th coordinate. Let $z=[z_1\ z_2\ \dots\ z_u]^T$. $A=[A_1\ A_2\ \dots\ A_u]$ hence A_k is the k-th column of A.
Then $Az = z_1A_1+z_2A_2+\cdots+z_uA_u$.
$Az=0\Rightarrow z_1A_1 = -(z_2A_2+\cdots+z_uA_u)\quad(*)$
We fix z_1A_1 to be an arbitrary $m\times 1$ matrix of 1s and 0s. The probability that $(*)$ holds is $\frac{1}{2^m}$.

Perfect Hashing

static case - N fixed items in the dictionary $x_1,x_2,\dots,x_N$
To perform `Query` *in $O(1)$ worst-case time.*

Quadratic Space: $M=N^2$

if $\mathcal{H}$ is universal and $M=N^2$, and h is sampled uniformly from $\mathcal{H}$, then the expected number of collisions is <1.

Proof. for $i\neq j$, let indicator r.v. A_{ij} be equal to 1 if $h(x_i)=h(x_j)$, or 0 otherwise.
By universality, $E[A_{ij}]=P(A_{ij}=1)\leq 1/N^2$
$E[\text{\# collisions}] = \sum_{i<j} E[A_{ij}] \leq \binom{N}{2}\frac{1}{N^2} < 1$

It follows that there exists $h\in\mathcal{H}$ causing no collisions (because if not, $\mathbb{E}$[#collisions] would be ≥ 1).

2-Level Scheme: $M=N$

- No collision and less space needed

Construction

Choose $h: U\to[N]$ from a universal hash family.
- Let L_k be the number of x_i's for which $h(x_i)=k$.
- Choose $h_1,\dots,h_N$ **second-level** hash functions $h_k:[N]\to[(L_k)^2]$ s.t. there are no collisions among the L_k elements mapped to k by h.
  - quadratic second-level table $\rightarrow$ ensures no collisions using quadratic space

Analysis

if $\mathcal{H}$ is universal and h is sampled uniformly from $\mathcal{H}$, then
$$E\left[\sum_k L_k^2\right] < 2N$$

Proof. For $i,j\in[1,N]$, define indicator r.v. $A_{ij}=1$ if $h(x_i)=h(x_j)$, or 0 otherwise.
A_{ij} = # possible collisions = # pairs * 2 $= L_k^2$
Hence $\sum_k L_k^2 = \sum_{i,j} A_{ij}$
$$E[\sum_{i,j}A_{ij}] = \sum_i E[A_{ii}] + \sum_{i\neq j}E[A_{ij}]$$
$$\leq N\cdot 1 + N(N-1)\cdot\frac{1}{N}$$
$$< 2N$$

Hash Table Resizing

- when number of inserted items, N is not known
  - **reshashing** - choose a new hash function of a larger size and re-hash all elements
  - costly but infrequent $\Rightarrow$ amortize

06. FINGERPRINTING & STREAMING

String Pattern Matching

problem: does the pattern string P occur as a substring of the text string T?

m = length of P, n = length of T, ℓ = size of alphabet

- assumption: operations on strings of length $O(\log n)$ can be executed in $O(1)$ time. (word-RAM model)
- naive solution: $\Theta(n^2)$

Fingerprinting approach (Karp-Rabin)

- faster string equality check:
  - for substring X, check $h(X)==h(P)$ for a hash function $h\Rightarrow\Theta(1)$ + cost of hashing instead of $\Theta(|X|)$
- **Rolling Hash**: $O(m+n)$
  - update the hash from what we already have from the previous hash - $O(1)$
  - compute $n-m+1$ hashes in $O(n)$ time
  - *Monte Carlo* algorithm

Division Hash

Choose a random **prime** number p in the range $\{1,\dots,K\}$.
For integer x, $h_p(x)=x \pmod p$

- if p is small and x is b-bits long in binary, hashing $\Rightarrow O(b)$
- hash family $\{h_p\}$ is approximately universal
- if $0\leq x<y<2^b$, then $\Pr_h[h_p(x)=h_p(y)] < \frac{b\ln K}{K}$

Proof. $h_p(x)=h_p(y)$ when $y-x=0\pmod p$.
Let $z=y-x$.
Since $z<2^b$, then z can have at most b distinct prime factors.
p divides z if p is one of these $\leq b$ prime factors.
number of primes in range $\{1,\dots,K\}$ is $>\frac{K}{\ln K}$,
hence the probability is $b/\frac{K}{\ln K} = \frac{b\ln K}{K}$

values of K

- higher K = lower probability of false positive
  - for $\delta=\frac{1}{100n}$, P(false positive) < 1%.

$\forall\delta>0$, if $X\neq Y$ and $K=\frac{2m}{\delta}\cdot\lg\ell\cdot\lg(\frac{2m}{\delta}\lg\ell)$, then $Pr[h(X)=h(Y)]<\delta$

Streaming

problem: Consider a sequence of insertions or deletions of items from a large universe $\mathcal{U}$. At the end of the stream, the *frequency* f_i of item i is its net count.

Let M be the sum of all frequencies at the end of stream.

naive solutions

- direct access table - $\Omega(U)$ space
- sorted list - $\Omega(M)$ space, no $O(1)$ update
- binary search tree - $O(M)$ space

Frequency Estimation

an approximation $\hat{f}_i$ is **ϵ-approximate** if
$$f_i-\epsilon M\leq\hat f_i\leq f_i+\epsilon M$$

Using Hash Table

$$f_i\leq\mathbb{E}[\hat f_i]\leq f_i+M/k$$

- increment/decrement $A[h(j)]$ on an empty table A of size k
- collision $\Rightarrow$ false positives $\Rightarrow$ may give overestimate of f_i
  - $A[h(i)]=\sum_{j:h(j)=h(i)} f_j\geq f_i$
- if h is drawn from a universal family, overestimate, $\mathbb{E}[A[h(i)]-f_i]\leq M/k$
- space: $O(\frac{1}{\epsilon}\cdot\lg M+\lg U\cdot\lg M)$
let $k=\frac{1}{\epsilon}$ for some $\epsilon>0$.
  - number of rows $=O(\frac{1}{\epsilon})$
  - size of each row $=O(\lg M)$
  - size of hash function (using universal hash family from ch.05) $=O(\lg U\cdot\lg M)$
- **Count-Min Sketch** $\rightarrow$ gives a bound on the probability that $\hat f_i$ deviates from f_i instead of a bound on the expectation of the gap

07. AMORTIZED ANALYSIS

- **amortized analysis** $\rightarrow$ guarantees the *average* performance of each operation in the *worst case*.
- For a sequence of n operations $o_1,o_2,\dots,o_n$,
  - let $t(i)$ be the time complexity of the i-th operation o_i
  - let $f(n)$ be the *worst-case* time complexity for *any* of the n operations
  - let $T(n)$ be the time complexity of all n operations

$$T(n)=\sum_{i=1}^n t(i)=nf(n)$$

Types of Amortized Analysis

Aggregate method

- look at the whole sequence, sum up the cost of operations and take the average - simpler but less precise
- e.g. binary counter - amortized $O(1)$
- e.g. queues (with `INSERT` and `EMPTY`) - amortized $O(1)$

Accounting method

- charge the i-th operation a fictitious amortized cost $c(i)$
  - **amortized cost** $c(i)$ is a fixed cost for each operation
  - **true cost** $t(i)$ depends on when the operation is called
- amortized cost $c(i)$ must satisfy:
$$\sum_{i=1}^n t(i)\leq\sum_{i=1}^n c(i) \text{ for all } n$$
- take the extra amount for cheap operations early on as "credit" paid in advance for expensive operations
  - **invariant**: bank balance never drops below 0
- the total amortized cost provides an **upper bound** on the total true cost

Potential method

- ϕ : potential function associated with the algo/DS
- $\phi(i)$: potential at the end of the i-th operation
- c_i : amortized cost of the i-th operation
- t_i : true cost of the i-th operation
$$c_i=t_i+\phi(i)-\phi(i-1)$$
$$\sum_{i=1}^n c_i=\phi(n)-\phi(0)+\sum_{i=1}^n t_i$$
- hence as long as $\phi(n)\geq 0$, then amortized cost is an upper bound of the true cost.
$$\sum_{i=1}^n c_i\geq\sum_{i=1}^n t_i$$
- usually take $\phi(0)=0$

- **e.g.** for queue:
  - let $\phi(i)$ = # of elements in queue after the i-th operation
  - amortized cost for insert: $c_i = t_i + \phi(i) - \phi(i-1) = 1 + 1 = 2$
  - amortized cost for empty (for k elements): $c_i = t_i + \phi(i) - \phi(i-1) = k + 0 - k = 0$

e.g. Dynamic Table (insertion only)

Aggregate method

cost of n insertions = $\sum_{i=1}^{n} t(i) \le n + \sum_{j=1}^{\lfloor \log(n-1) \rfloor} 2^j \le 3n$

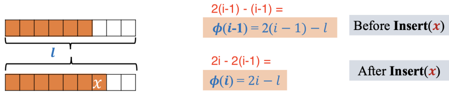
i	1	2	3	4	5	6	7	8	9	10	
$size_i$	1	2	4	4	8	8	8	8	16	16	
$t(i)$	1	1	1	1	1	1	1	1	1	1	Cost for i th insert
		1	2		4				8		Cost for copying due to overflow

Accounting method

- charge \$3 per insertion
  - \$1 for insertion itself
  - \$1 for moving itself when the table expands
  - \$1 for moving one of the existing items when the table expands

Potential method

let $\phi(i) = 2i - size(T)$



Operation Insert(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1	2	3
Case 2: when table is already full	i	$3 - i$	3

Amortized cost of n insertions = $3n = O(n)$

Actual cost of n insertions = $O(n)$

08. DYNAMIC PROGRAMMING

- **cut-and-paste proof** $\rightarrow$ proof by contradiction - suppose you have an optimal solution. Replacing ("cut") subproblem solutions with this subproblem solution ("paste" in) should improve the solution. If the solution doesn't improve, then it's not optimal (contradiction).

Longest Common Subsequence

- for sequence $A : a_1, a_2, \dots, a_n$ stored in array
- C is a **subsequence** of $A \rightarrow$ if we can obtain C by removing zero or more elements from A.

problem: given two sequences $A[1..n]$ and $B[1..m]$, compute the *longest* sequence C such that C is a subsequence of A and B.

brute force solution

- check *all* possible subsequences of A to see if it is also a subsequence of B, then output the longest one.
- analysis: $O(m2^n)$
  - checking each subsequence takes $O(m)$
  - 2^n possible subsequences

recursive solution

let $LCS(i, j)$: longest common subsequence of $A[1..i]$ and $B[1..j]$

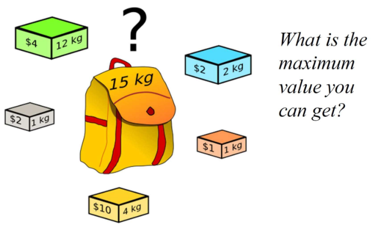
- base case: $LCS(i, 0) = \emptyset$ for all i, $LCS(0, j) = \emptyset$ for all j
- general case:
  - if last characters of A, B are $a_n = b_m$, then $LCS(n, m)$ must terminate with $a_n = b_m$
    - the optimal solution will match a_n with b_m
  - if $a_n \ne b_m$, then either a_n or b_m is not the last symbol
- **optimal substructure**: (general case)
  - if $a_n = b_m$, $LCS(n, m) = LCS(n-1, m-1) :: a_n$
  - if $a_n \ne b_m$, $LCS(n, m) = LCS(n-1, m) \mid\mid LCS(n, m-1)$
- **simplified problem**:
  - $L(n, m) = 0$ if $n = 0$ or $m = 0$
  - if $a_n = b_m$, then $L(n, m) = L(n-1, m-1) + 1$
  - if $a_n \ne b_m$, then $L(n, m) = \max(L(n, m-1), L(n-1, m))$

analysis

- number of distinct subproblems $= (n+1) \times (m+1)$
- to use $O(\min\{m, n\})$ space: bottom-up approach, column by column
- memoize for DP $\Rightarrow$ makes it $O(mn)$ instead of exponential time

Knapsack Problem

- input: $(w_1, v_1), (w_2, v_2), \dots, (w_n, v_n)$ and capacity W
- output: subset $S \subseteq \{1, 2, \dots, n\}$ that maximises $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \le W$



- 2^n subsets $\Rightarrow$ naive algorithm is costly
- **recursive solution**:
  - let $m[i, j]$ be the maximum value that can be obtained using a subset of items $\{1, 2, \dots, i\}$ with total weight no more than j.
  - $m[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j-w_i] + v_i, m[i-1, j]\}, & \text{if } w_i \le j \\ m[i-1, j], & \text{otherwise} \end{cases}$
- **analysis**: $O(nW)$
  - **!** $O(nW)$ is **not** a polynomial time algorithm
  - not polynomial in input bitsize
    - W can be represented in $O(\lg W)$ bits
    - n can be represented in $O(\lg n)$ bits
  - polynomial time is strictly in terms of the number of bits for the input

Changing Coins

problem: use the fewest number of coins to make up n cents using denominations $d_1, d_2, \dots, d_n$. Let $M[j]$ be the fewest number of coins needed to change j cents.

- **optimal substructure**:
  - $M[j] = \begin{cases} 1 + \min_{i \in [k]} M[j - d_i], & j > 0 \\ 0, & j = 0 \\ \infty, & j < 0 \end{cases}$

Proof. Suppose $M[j] = t$, meaning $j = d_{i_1} + d_{i_2} + \dots + d_{i_t}$ for some $i_1, \dots, i_t \in \{1, \dots, k\}$.

Then, if $j' = d_{i_1} + d_{i_2} + \dots + d_{i_t - 1}$, $M[j'] = t - 1$, because otherwise if $M[j'] < t - 1$, by **cut-and-paste** argument, $M[j] < t$.

09. GREEDY ALGORTIHMS

- solve only one subproblem at each step
- beats DP and divide-and-conquer when it works
- **greedy-choice property** $\rightarrow$ a locally optimal choice is globally optimal

Examples

Fractional Knapsack

- $O(n \log n)$
- **greedy-choice property**: let j^* be the item with *maximum* value/kg, v_j / w_i. Then there exists an optimal knapsack containing $\min(w_{j^*}, W)$ kg of item j^*.
- **optimal substructure**: if we remove w kg of item j from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most $W - w$ kgs that one can take from $n - 1$ original items and $w_j - w$ kg of item j.

Proof. cut-and-paste argument

Suppose the remaining load after removing w kgs of item j was *not* the optimal knapsack weighing ...

Then there is a knapsack of value $> X - v_j \cdot \frac{w}{w_j}$ with weight ...

Combining this knapsack with w kg of item j gives a knapsack of value $> X \Rightarrow$ contradiction!

Minimum Spanning Trees

for a connected, undirected graph $G = (V, E)$, find a spanning tree T that connects all vertices with minimum weight. Weight of spanning tree T, $w(T) = \sum_{(u,v) \in T} w(u, v)$.

- **optimal substructure**: let T be a MST. remove any edge $(u, v) \in T$. then T is partitioned into T_1, T_2 which are MSTs of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

Proof. cut-and-paste: $w(T) = w(u, v) + w(T_1) + w(T_2)$

if $w(T_1') < w(T_1)$ for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G.

$\Rightarrow$ contradiction, T is the MST

- **Prim's algorithm** - at each step, add the least-weight edge from the tree to some vertex outside the tree
- **Kruskal's algorithm** - at each step, add the least-weight edge that does *not* cause a cycle to form

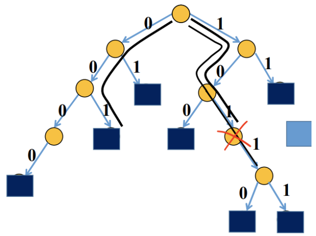
Binary Coding

Given an alphabet set $A : \{a_1, a_2, \dots, a_n\}$ and a text file F (sequence of alphabets), how many bits are needed to encode a text file with m characters?

- **fixed length encoding**: $m \cdot \lceil \log_2 n \rceil$
  - encode each alphabet to unique binary string of length $\lceil \log_2 n \rceil$
  - total bits needed for m characters $= m \cdot \lceil \log_2 n \rceil$
- **variable length encoding**
  - different characters occur with different frequency $\Rightarrow$ use fewer bits for *more frequent* alphabets
  - average bit length, $ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$
  - BUT overlapping prefixes cause indistinguishable characters

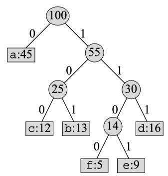
Prefix coding

- a coding $\gamma(A)$ is a **prefix coding** if $\nexists x, y \in A$ such that $\gamma(x)$ is a prefix of $\gamma(y)$.
- **labelled binary tree**: $\gamma(A)$ = label of path from root



- for each prefix code A of n alphabets, there exists a binary tree T on n leaves such that there is a **bijective mapping** between the alphabets and the leaves
- $ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)| = \sum_{x \in A} f(x) \cdot |depth_T(x)|$
- the binary tree corresponding to an *optimal* prefix coding must be a **full binary tree**.
  - every internal node has degree exactly 2
  - multiple possible optimal trees - most optimal depends on alphabet frequencies
- accounting for alphabet **frequencies**:
  - let $a_1, a_2, \dots, a_n$ be the alphabets of A in non-decreasing order of thier frequencies.
  - a_1 must be a leaf node; a_2 *can* be a sibling of a_1.
  - there exists an optimal prefix coding in which a_1 and a_2 are siblings
- derivation of optimal prefix coding: **Huffman's algorithm**
  - keep merging the two least frequent items

```
Huffman(C):
  Q = new PriorityQueue(C)
  while Q:
    allocate a new node z
    z.left = x = extractMin(Q)
    z.right = y = extractMin(Q)
    z.val = x.val + y.val
    Q.add(z)
  return extractMin(Q) // root
```



helpful approximations

stirling's approximation: $T(n) = \sum_{i=0}^{n} \log(n-i) = \log \prod_{i=0}^{n}(n-i) = \Theta(n \log n)$

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

basel problem: $\sum_{n=1}^{N} \frac{1}{n^2} \leq 2 - \frac{1}{N} \xrightarrow{N\to\infty} 2$

because $\sum_{n=1}^{N} \frac{1}{N^2} \leq 1 + \sum_{x=2}^{\log_3 n} \frac{1}{(x-1)x} = 1 + \sum_{n=2}^{N}(\frac{1}{n-1} - \frac{1}{n}) = 1 + 1 - \frac{1}{N} = 2 - \frac{1}{N}$

number of primes in range $\{1, \dots, K\}$ is $> \frac{K}{\ln K}$

asymptotic bounds

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < 2^{2n}$
$\log_a n < n^a < a^n < n! < n^n$
for any $a, b > 0, \quad \log_a n < n^b$

multiple parameters

for two functions $f(m,n)$ and $g(m,n)$, we say that $f(m,n) = O(g(m,n))$ if there exists constants c, m_0, n_0 such that $0 \leq f(m,n) \leq c \cdot g(m,n)$ for all $m \geq m_0$ or $n \geq n_0$.

set notation

- $O(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$
- $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$
- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \mid \forall n \geq n_0, \quad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} = O(g(n)) \cap \Omega(g(n))$
- $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, \quad 0 \leq f(n) < cg(n)\}$
- $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, \quad 0 \leq cg(n) < f(n)\}$

example proofs

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 $n \geq n_0$. □

Proof. $n = o(n^2)$

For any $c > 0$, use $n_0 = 2/c$.

Proof. $n^2 - n = \omega(n)$

For any $c > 0$, use $n_0 = 2(c+1)$.

Example. let $f(n) = n$ and $g(n) = n^{1+\sin(n)}$.

Because of the oscillating behaviour of the sine function, there is no n_0 for which f dominates g or vice versa.

Hence, we cannot compare f and g using asymptotic notation.

Example. let $f(n) = n$ and $g(n) = n(2 + \sin(n))$.

Since $\frac{1}{3}g(n) \leq f(n) \leq g(n)$ for all $n \geq 0$, then $f(n) = \Theta(g(n))$. (note that limit rules will not work here)

mentioned algorithms

- ch.3 - **Misra Gries** - space-efficient computation of the majority bit in array A
- ch.3 - **Euclidean** - efficient computation of GCD of two integers
- ch.3 - **Tower of Hanoi** - $T(n) = 2^n - 1$
  1. move the top $n-1$ discs from the first to the second peg using the third as temporary storage.
  2. move the biggest disc directly to the empty third peg.
  3. move the $n-1$ discs from the second peg to the third using the first peg for temporary storage.
- ch.3 - **MergeSort** - $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$
- ch.3 - **Karatsuba Multiplication** - multiply two n-digit numbers x and y in $O(n^{\log_2 3})$
  - worst-case runtime: $T(n) = 3T(\lceil n/2 \rceil) + \Theta(n)$

uncommon notations

- $\perp$ - false