CS3230

01. ASYMPTOTIC ANALYSIS

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

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 (use for θ,o,ω)

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 $f(n) = O(g(n))$

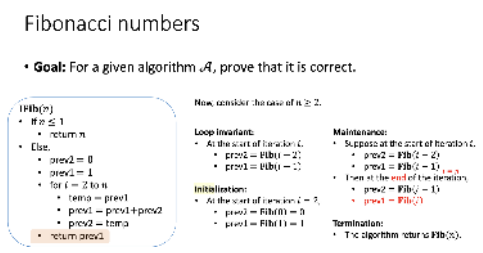
$$\boxed{\log\log n < \log n < (\log n)^k < n^k < k^n}$$

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

03a. Proof of Correctness

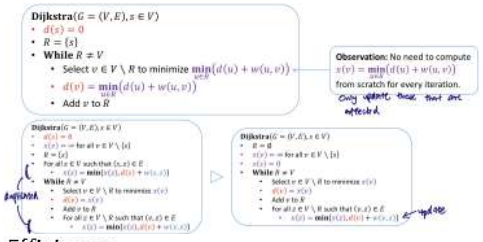
Iterative Algorithms

- **iterative** → loop(s), sequentially processing input elements
- **loop invariant** implies correctness if (Use induction hypo)
  - definition: Some desirable conditions that should be satisfied at the start of each iteration.
  - *initialisation* - true at the start of the first iteration of the loop (Base case)
  - *maintenance* - if the loop invariant is satisfied at the start of the current iteration, then the loop invariant must be satisfied at the start of the next iteration (Inductive step)
  - *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.
- **Fibonacci numbers**: At the start of iteration i, prev2 = Fib(i−2), prev1 =Fib(i1)
- **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.
- **Dijkstra's algorithm**



Efficiency

- The algorithm can be implemented using a priority queue.

	Insert	Extract-min	Decrease-key	Time complexity of Dijkstra's algorithm
Binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O((\lvert E\rvert + \lvert V\rvert)\log\lvert V\rvert)$
Fibonacci heap	$O(1)$	$O(\log n)$ amortized	$O(1)$ amortized	$O(\lvert E\rvert + \lvert V\rvert \log\lvert V\rvert)$

https://en.wikipedia.org/wiki/Priority_queue
Amort acc = average over n operations.

Recursive Algorithms

- Analysis of a recursive algorithm:
- Base case: Show that the algorithm is correct for the base case. (without recursion)
- Inductive step:
  - Assume that the algorithm is correct for any input of size smaller than n
  - Show that the algorithm is correct for any input of size n.

Divide-and-Conquer

1. Divide the problem into smaller subproblems.
2. Solve the subproblems recursively.
3. Combine the subproblem solutions to get the solution of the full problem.

- Eg: Mergesort

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

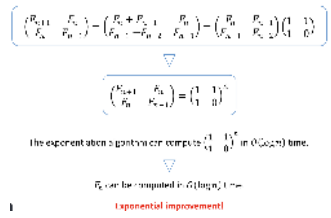
Exponentiation

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

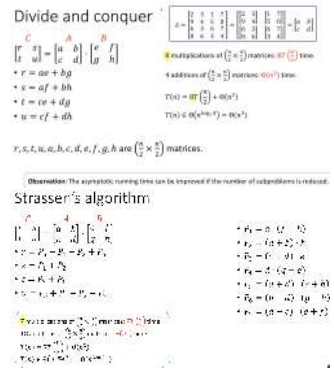
- observation: $f(x+y,m) = f(x,m) * f(y,m) \pmod m$
- **naive solution**: recursively compute and combine $f(n-1,m) * f(1,m) \pmod m$
  - $T(n) = T(n-1) + T(1) + \Theta(1) \Rightarrow T(n) = \Theta(n)$
- **better solution**: divide and conquer
  - divide: trivial
  - conquer: recursively compute $f(\lfloor n/2 \rfloor, m)$
  - 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
  - $T(n) = T(n/2) + \Theta(1) \Rightarrow \Theta(\log n)$

This document is available on studocu
messages.downloaded_by

Fibonacci numbers



Matrix multiplication



Solving Recurrences

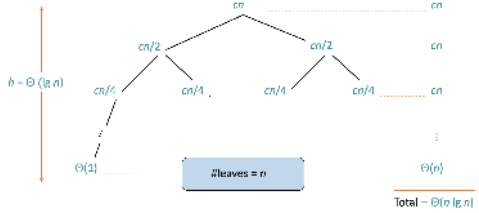
for a sub-problems of size $\frac{n}{b}$ where $f(n)$ is the time to divide and combine,

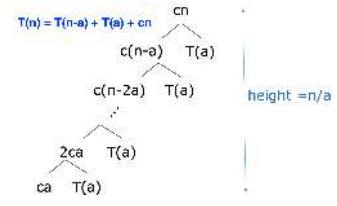
$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

Recursion tree

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

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

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 $n > k \geq 1$
- $T(n) = 4T(n/2) + n$
  $= 4(c_1(n/2)^2 - c_2(n/2)) + n$
  $= c_1 n^2 - 2c_2 n + n$
  $= c_1 n^2 - c_2 n + (1 - c_2)n$
  $= c_1 n^2 - c_2 n$ since $c_2 = 1 \Rightarrow 1 - c_2 = 0$ □

04a. Lower Bound for Comparison-Based Sorting

> **adversary argument** →
> inputs are decided such that they have different solutions

Comparison-based sorting

- **Theorem**: The worst-case time complexity of any comparison-based sorting algorithm is $\Omega(nlogn)$
- 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

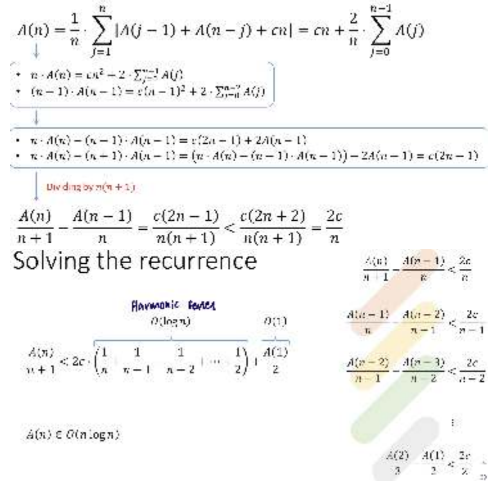
- each node is a comparison
- each branch is an outcome of the comparison
- each leaf is a class label (decision after *all* comparisons)
- **worst-case** runtime = height of tree
- # of leaves = # of permutations $\Rightarrow n!$
- height of tree is at least $log(n!) = \Omega(nlogn)$ (by stirling's approx)
- any decision tree that can sort n elements must have height $\Omega(n \lg n)$.
- Theorem proven

04b. AVERAGE-CASE ANALYSIS & RANDOMISED ALGORITHMS

- **average case** $A(n)$ → 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.
  - $A(n) = \mathop{\mathbb{E}}_{x \sim \mathcal{D}_n}$ [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
- $T(n) = T(j-1) + T(n-j) + \Theta(n)$
  - if the pivot produces subarrays of size j and $(n-j-1)$
- **worst-case**: $T(n) = T(0) + T(n-1) + \Theta(n) \Rightarrow \Theta(n^2)$
- Avg time analysis:
  - Unifomity
    - Observation 1: The pivot is selected uniformly at random. $Pr[pivot = a_j] = \frac{i}{n}$
    - Observation 2: The permutations for both recursive calls are also uniformly random.
  - $A(n) = \frac{1}{n}\sum_{i=1}^{n} A(i-1) + A(n-i) + cn$
    $= cn \frac{2}{n} \sum_{j=0}^{n-1} A(i)$



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

let $P(i)$ be the set of all those permutations of elements $\{e_1, e_2, \ldots, e_n\}$ that begins with e_i.
Let $G(n, i)$ be the average running time of quicksort over $P(i)$. Then
$G(n) = A(i-1) + A(n-i) + (n-1)$.
$A(n) = \frac{1}{n}\sum_{i=1}^{n} G(n, i)$
$= \frac{1}{n}\sum_{i=1}^{n}(A(i-1) + A(n-i) + (n-1))$
$= \frac{2}{n}\sum_{i=1}^{n} A(i-1) + n - 1$
$= O(n \log n)$ 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	$n \lg n$	$n \lg n$	$n \lg n$

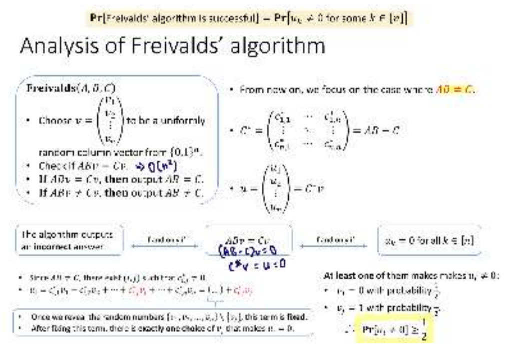
- disadvantages of mergesort:
  - overhead of temporary storage
  - cache misses
- advantages of quicksort
  - in place
  - reliable (as $n \uparrow$, chances of deviation from avg case $\downarrow$)
- issues with quicksort
  - **distribution-sensitive** → time taken depends on the initial (input) permutation

Randomised Algorithms

- **randomised algorithms** → 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 algo: correct with prob 0.99 for every input vs. Avg case analysis: Correct for 99% of input
- expected running time = worst-case running time = $E(n) = \max_{\text{input } x \text{ of size } n} \mathbb{E}$[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

messages.downloaded_by

Matrix Multiplication: Freidvald's algorithm



> **Principle of deferred decision** →
> If we can show that $Pr[\epsilon | X = x] \geq p$ for every x where X is the revealed randomness, then $Pr[\epsilon] \geq p$

- Freidvald's algo is incorrect with a probability of at most $\frac{1}{2}$.
- Do better?

> **Success Probability Amplification** →
> The error probability can be reduced to at most f by repeating the algo for $t = \lceil log\frac{1}{f} \rceil$ times.

- If $AB = C$, Freivalds' algorithm always answers $AB = C$ correctly.
- If $AB! = C$, the probability that Freivalds' algorithm answers $AB = C$ for all t iterations is at most $\frac{1}{2^t} \leq f$

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, $E[X] = \frac{1}{p}$
$$Pr[X = k] = q^{k-1}p$$

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?

- Similar to **Balls and Bins problem**
- Throw m balls into n bins randomly and independently. What is the probability that every bin contains at least one ball?
  - First consider 1 bin: The probability that the bin contains zero balls is $(1 - \frac{1}{n})^m \leq e^{-\frac{m}{n}}$
  - By **Union Bound**, The probability that at least one bin contains zero balls is at most $n(1 - \frac{1}{n})^m \leq ne^{-\frac{m}{n}}$
  - So the probability is at most $\frac{1}{n}$ if $m \geq 2n\lceil lnn \rceil$
- **Answer**: Drawing $m = 2n\lceil lnn \rceil \in \theta(nlogn)$ times guarantees a success probaility of at least $1 - \frac{1}{n}$
- 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 $= \sum_{i=1}^{n} \frac{n}{n-(i-1)} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = \Theta(n \lg n)$

Techniques

Union Bound →
$Pr[\epsilon] = Pr[\epsilon_1 \bigvee \epsilon_2 \bigvee ... \bigvee \epsilon_n] \leq Pr[\epsilon_1] + ... Pr[\epsilon_n]$
To make sure that $Pr[\epsilon] \leq f$, it suffices that $Pr[\epsilon_i] \leq \frac{f}{n}$ for each $i \in [n]$

Expectation →
$\mathbb{E}[X] = \sum_x x Pr[X = x]$, where the sum ranges over all possible outcomes x of the random variable X

Markov inequality →
If X is a non-negative random variable and $a > 0$ then $Pr[X \leq a\mathbb{E}[X]] \leq \frac{1}{a}$

- Application
  - If the expected runtime of A is at most t
  - Then the runtime of A is at most 100t with probability at least 0.99
  - $Pr[runtime \geq 100t] \leq Pr[runtime \leq 100 expected runtime] \leq \frac{1}{100}$
  - If the expected time complexity of A is $O(nlogn)$
  - Then the time complexity of A is $O(nlogn)$ with probability at least 0.99

Linearity of expectation →
For any two events X, Y and a constant a,
$E[X + Y] = E[X] + E[Y]$
$E[aX] = aE[X]$

Indicator random variables →
Let ϵ be an event. The indicator random variable 1_ϵ for ϵ is defined as
$$1_\epsilon = \begin{cases} 1, & \text{if } \epsilon \text{ occurs,} \\ 0, & \text{otherwise} \end{cases}$$
$\mathbb{E}[1_\epsilon] = Pr[\epsilon]$

Randomised Quicksort Analysis

- Choose pivot at random
- **Observation**: Running time of quick sort $\in \theta$(Number of comparisons made)
- For any a_i, a_j within A[], the number of comparisons made $X_{i,j}$ between them is either 0 or 1.
  - When $X_{i,j}$ is 1, either one of them was chosen to be pivot.
  - When $X_{i,j}$ is 0, neither was chosen to be pivot.
- For any $1 \leq i < j \leq n, Pr[X_{i,j} = 1] = \frac{2}{j-i+1}$
- Using indicator random variables, $\mathbb{E}[X_{i,j}]$
  = $Pr[X_{i,j} = 1]$
  = $\frac{2}{j-i+1}$
- So $\mathbb{E}$[number of comparisons]
  =$\sum_{i \leq i < j \leq n} \mathbb{E}[X_{i,j}]$
  = $\sum_{i \leq i < j \leq n} \frac{2}{j-i+1} \in O(nlogn)$
- **Theorem**: The expected running time of randomized quick sort is $O(logn)$
- By Markov inequality, Randomized quick sort finishes in $O(logn)$ time with probability at least 0.99.

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*
  - eg: Freidvald's algorithm
- Las Vegas is stronger as we can turn it into a Monte Carlo using markov inequality

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

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**, $h : U \to \{1, \ldots, M\}$ gives the location of where to store in the hash table
  - notation: $[M] = \{1, \ldots, M\}$
  - storing N items in hash table of size M
- **collision** → for two different keys x and y, $h(x) = h(y)$

This document is available on
messages downloaded by

- 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 $[M]$.

$\mathcal{H}$ is **universal** if $\forall x \neq y$, $\frac{|h \in \mathcal{H}: h(x)=h(y)|}{|\mathcal{H}|} \leq \frac{1}{M}$
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 $x_1, \ldots, x_N \in \mathcal{U}$, 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 $x_1, \ldots, x_{N-1} \in \mathcal{U}$ 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 $x_1, \ldots, x_N$ 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 ... % 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 $z = [z_1\ z_2\ \ldots\ z_u]^T$. $A = [A_1\ A_2\ \ldots\ A_u]$ 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 $x_1, x_2, \ldots, x_N$
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[# 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 $h_1, \ldots, h_N$ **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 → 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 $\sum_k L_k^2 = \sum_{i,j} A_{ij}$

$$E[\sum_{i,j} A_{ij}] = \sum_i E[A_{ii}] + \sum_{i \neq j} E[A_{ij}]$$
$$\leq N \cdot 1 + N(N-1) \cdot \tfrac{1}{N}$$
$$< 2N$$

Hash Table Resizing

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

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).
- **overlapping subproblems** - recursive solution contains a small number of distinct subproblems repeated many times (Solve with memoization)

Longest Common Subsequence

- for sequence $A : a_1, a_2, \ldots, a_n$ 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: $O(m2^n)$, where n, m is length of A, B respectively
  - checking each subsequence takes $O(m)$
  - 2^n possible subsequences

recursive solution

let $LCS(i, j)$: longest common subsequence of $A[1..i]$ and $B[1..j]$. Compute LCS(i,j) for all $i \in [n]$ and $j[m]$ recursively

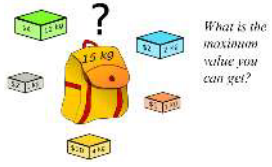
- base case: $LCS(i, 0) = \emptyset$ for all i, $LCS(0, j) = \emptyset$ for all j
- general case:
  - if last characters of A, B are $a_n = b_m$, then $LCS(n, m)$ must terminate with $a_n = b_m$
    - the optimal solution will match a_n with b_m
  - if $a_n \neq b_m$, then either a_n or b_m is not the last symbol
- **optimal substructure**: (general case)
  - Define: An optimal solution to a problem contains optimal solutions to subproblems.
  - if $a_n = b_m$, $LCS(n, m) = LCS(n-1, m-1) :: a_n$
  - if $a_n \neq b_m$, $LCS(n, m) = LCS(n-1, m) \,||\, LCS(n, m-1)$
- **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 $a_n \neq b_m$, 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, row by row
- memoize for DP $\Rightarrow$ makes it $O(mn)$ instead of exponential time

Knapsack Problem

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



- 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 $\{1, 2, \ldots, i\}$ with total weight no more than j.
  - $m[i, j] =$

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

- **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 $d_1, d_2, \ldots, d_n$. Let $M[j]$ be the fewest number of coins needed to change j cents.

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

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

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

- runtime: $O(nk)$ for n cents, k denominations

messages.downloaded_by

$$T(n) = 2T(\tfrac{n}{2}) + O(n) \quad \Rightarrow O(n \log n)$$
$$T(n) = T(\tfrac{n}{2}) + O(n) \quad \Rightarrow O(n)$$
$$T(n) = 2T(\tfrac{n}{2}) + O(1) \quad \Rightarrow O(n)$$
$$T(n) = T(\tfrac{n}{2}) + O(1) \quad \Rightarrow O(\log n)$$
$$T(n) = 2T(n-1) + O(1) \quad \Rightarrow O(2^n)$$
$$T(n) = 2T(\tfrac{n}{2}) + O(n \log n) \quad \Rightarrow O(n(\log n)^2)$$
$$T(n) = 2T(\tfrac{n}{4}) + O(1) \quad \Rightarrow O(\sqrt{n})$$
$$T(n) = T(n-c) + O(n) \quad \Rightarrow O(n^2)$$

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 $\{1, \ldots, K\}$ 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

$O(g(n))$ is actually a *set of functions*. $f(n) = O(g(n))$ means $f(n) \in O(g(n))$
- $O(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$
- $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$
- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \mid \forall n \geq n_0, \quad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} = O(g(n)) \cap \Omega(g(n))$
- $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, \quad 0 \leq f(n) < cg(n)\}$
- $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, \quad 0 \leq cg(n) < f(n)\}$

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

This document is available on studocu
messages.downloaded_by