1 Gale-Shapley Algorithm

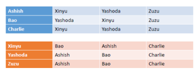
1.1 What problems does it solve?

Gale-Shapley Algorithm mainly solves matching problems between n pairs and produces a *"stable matching"*. e.g.

- There are n med school graduates and n hospital. Each candidate ranks all the hospitals and vice versa. How do we pair them up?
- Matchmaker must match n men and n women. Each man ranks all the women, and each woman ranks all the men. How to pair them up?

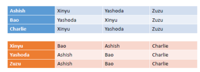
1.2 What is a *stable match*?

- A matching is stable if no unmatched man and woman both prefer each other to their current partner



Is (Ashish, Yashoda), (Bao, Xinyu), (Charlie, Zuzu) stable?

Yes !



Is the matching (Ashish, Zuzu), (Bao, Yashoda), (Charlie, Xinyu) stable?

A prefers Y over Z
Y prefers A over B
Not stable

1.3 Gale-Shapley Algorithm

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}
```

ambiguity here as to which man is chosen

need to ensure this part runs at constant time for algo to be n^2

1.3.1 Analysis of Gale-Shapley Algorithm

Lemma: The `while` loop in Gale-Shapley runs in $\leq n^2$ time.

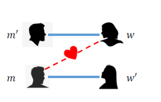
Why?

- A man never proposes to the same women twice when he makes a new proposal.
- There are n^2 possible proposals, $\therefore$ there can only be n^2 loops
- Recurring idea in algorithm analysis: **Find a progress measure that keeps strictly increasing**, in this case its the # of proposals

1.3.2 Other general observation

- Men propose to women in decreasing order of preference.
- Women's partners keep getting better.
- Each man is engaged to a unique woman.
- The matching produced is **stable**. Proof by contradiction:

<u>Lemma</u>: When the algorithm terminates, the matching between men and women is stable.



- m must have proposed to w before w', because he ranks w higher than w'.
- When w rejected m, she must have got engaged to a man that she prefers to m. But because women's partners keep getting better, she must prefer m' to m. Contradiction!

- The above version of Gale-Shapley produces *man-optimal* stable matching where all man are matched with the $best(m)$

1.3.3 Run time of Gale-Shapley Algorithm

Gale-Shapley Algorithm returns a **stable matching** in $O(N^2)$ time.

- Size of all preference profiles is $2n^2$, so cannot expect sub-quadratic running time, i.e. $O(N^2)$ is an optimal running time

2 Asymptotic Analysis

2.1 What is an Algorithm?

- A finite sequence of "well-defined" instructions to solve a given computational problem

2.1.1 Objective

- Design efficient algorithms in terms of:
  1. **Running time**, i.e. smallest $O(x)$
  2. Other matrix such as **space**

2.2 How to analyze running time?

1. **Simulation**: Run the algorithm many times and measure the running time
   - Tends to be **Machine Dependent** (different hardware) and **Input Dependent** (e.g. might work slow for $n = 500$ but fast for $n = 499$)
2. Mathematical Analysis
   - Removes *"external dependencies"* to give us a better idea of how fast algorithms run

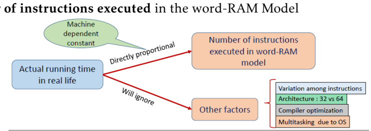
2.3 Word-RAM Model

Assumptions:

- Word is the basic storage of RAM which is basically a collection of bytes
- Any arbitrary location in RAM can be accessed in the same time irrespective of location
- Data as well as program reside fully in RAM
- Each arithmetic operation involves a constant number of words which takes a constant number of CPU cycles to run

2.3.1 How to measure running time?

- **Number of instructions executed** in the word-RAM Model



2.4 Asymptotic Notations

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

$$\boxed{0 \leq f(n) \leq cg(n)}$$

Ω-notation [lower bound ($\geq$)]: $f(n) = \Omega(g(n))$ if $\exists c > 0, n_0 > 0$ such that $\forall n \geq n_0$

$$\boxed{0 \leq cg(n) \leq f(n)}$$

Θ-notation [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 $\forall n \geq n_0$

$$\boxed{0 \leq f(n) \leq cg(n)}$$

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

$$\boxed{0 \leq cg(n) \leq f(n)}$$

Note: $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

2.5 Limits

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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad \Rightarrow f(n) = o(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = O(g(n))$$

$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = \Theta(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \qquad \Rightarrow f(n) = \Omega(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \Rightarrow f(n) = \omega(g(n))$$

$$\lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right) = 0 \rightarrow f(n) = o(g(n))$$

- **Proof:**
  - Since $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$, by definition, we have:
    - For all $\varepsilon>0$, <u>there exists</u> $\delta>0$ such that $\frac{f(n)}{g(n)} < \varepsilon$ for $n > \delta$.
  - Set $c = \varepsilon$ and $n_0 = \delta$. We have:
    - For all $c>0$, there exists $n_0>0$ such that $\frac{f(n)}{g(n)} < c$ for $n > n_0$.
    - Hence, for all $c>0$, there exists $n_0>0$ such that $f(n) < c \cdot g(n)$ for $n > n_0$.
    - By definition, $f(n) = o(g(n))$.

2.6 Properties of Big-O

1. **Transitivity**: applies for $O, \Theta, \Omega, o, \omega$
   $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
2. **Reflexivity**: for O, Ω, Θ
   $f(n) = O(f(n))$
3. **Symmetry**: $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
4. **Complementarity**:
   - $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
   - $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$

2.6.1 Common useful facts

- if $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$
- if $f(n) = o(g(n))$, then $f(n) = O(g(n))$
- Degree-k polynomials are $O(n^k)$.
- Degree-k polynomials are $o(n^{k+1})$ and $\omega(n^{k-1})$.
- Poly dominates logs: $(\log n)^{100} = o(n^{.0001})$, i.e. $n^k > (\log n)^k$
- Exponential dominate polys: $n^{100} = o(2^{.001n})$, i.e. $2^{kn} > n^k$
- $1 < \log\log n < \log n < (\log n)^k < \sqrt{n} < n < n^k < a^n < n! < n^n$
- $2^{2^{\lg\lg n}} < n^2 \lg\lg n < n^3 \leq \sum_{i=2}^{n} \frac{n^3}{i(i-1)} < n^{\lg n} < 2^n < (\lg n)^n < n!$

2.6.2 Common Confusions

- $2^{n+5} = O(2^n)$, because $2^{n+5} = 32 \cdot 2^n$
- $2^{5n} \neq O(2^n)$!!!
- $\max(f(n), g(n)) = \Theta(f(n) + g(n))$, at most $f(n) + g(n)$, at least $1/2 \times f(n) + g(n)$
- $\sin(n) \neq \Omega(\cos n)$

2.6.3 Impossible combinations

- $f(n) = \Omega(g(n))$ and $f(n) = o(g(n))$
- $f(n) = O(g(n))$ and $f(n) = \omega(g(n))$

3 Iteration, Recursion and Divide-and-Conquer

3.1 Iterative Algorithms

- one or multiple loops $\longrightarrow$ sequentially processing input elements
- **Loop invariant** implies correctness if
  - *Initialization*: invariant is true before first iteration of loop
  - *Maintenance*: if invariant is true before an iteration, it remains true at beginning of next iteration
  - *Termination*: invariant provides a useful property for showing correctness when program terminates
- Examples
  - InsertionSort: subarray $A[1 \ldots j-1]$ consists of elements originally in $A[1 \ldots j-1]$ but in sorted order
  - SelectionSort: subarray $A[1 \ldots j-1]$ is sorted and contains the $j-1$ smallest elements of A

3.2 Divide-and-Conquer

Consists of 3 parts:

1. *Divide* problem into smaller subproblems
2. *Conquer* subproblems by solving recursively, small subproblems are trivial to solve
3. *Combine* solutions of subproblems into solution of original problem

3.2.1 Tower of Hanoi

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)    ⟨⟨Recurse!⟩⟩
      move disk n from src to dst
      HANOI(n − 1, tmp, dst, src)    ⟨⟨Recurse!⟩⟩
```

- Runtime: $T(n) = 2 \cdot T(n-1) + 1 = 2^n - 1$

3.2.2 Merge Sort

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & if\ n > 1 \end{cases}$$

Running time: $\Theta(n \lg n)$

3.3 How to solve recurrence?

Recurrence equation typically comes in the form

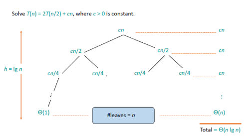
$$T(n) = aT(n/b) + f(n)$$

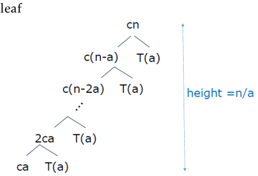
where a is the # of subproblems, n/b is the subproblem size and $f(n)$ is the time to divide and combine

3.3.1 Recursion Tree

Running time = height of tree $\times$ number of leaves

- Each node represents cost of a single subproblem
- Height of tree = longest path from root to leaf





3.3.2 Master Method

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

where $a \geq 1, b > 1$ and f is asymptotically positive **3 common cases of Master Method**

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$,
   - $f(n)$ 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 $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$,
   - 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 $T(n) = \Theta(f(n))$.

Examples of Master Theorem Cases

Ex. $T(n) = 4T(n/2) + n$
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n.$
CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.
$\therefore T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$
CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.
$\therefore T(n) = \Theta(n^2 \lg n)$.

$T(n) = 4T(n/2) + n^3$
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$
CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$
and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.
$\therefore T(n) = \Theta(n^3)$.

Examples where Master Theorem Don't Work

- $T(n) = T(n/2) + n(1 - \cos n)$.
- i.e., $a=1, b=2 \rightarrow n^{\log_2 1} = n^0 = 1$, $f(n)=n(1 - \cos n)$.
- $n(1 - \cos n) \notin O(n^{0-\varepsilon}) \rightarrow$ Not case 1
- $n(1 - \cos n) \notin \Theta(n^0 \log^k n)$ for any $k \geq 0$ $\rightarrow$ Not case 2
- $n(1 - \cos n) \notin \Omega(n^{0+\varepsilon}) \rightarrow$ Not case 3.

$T(n) = 4T(n/2) + n^2/\lg n$
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\lg n.$

$n^2/\lg n \notin O(n^{2-\varepsilon}) \rightarrow$ Not case 1
- *Reason:* for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.

$n^2/\lg n \notin \Theta(n^2 \log^k n)$ for any $k \geq 0$ $\rightarrow$ Not case 2
$n^2/\lg n \notin \Omega(n^{2+\varepsilon}) \rightarrow$ Not case 3
Master method does not apply.

3.3.3 Substitution Method

1. guess that $T(n) = O(f(n))$.
2. verify by induction:
   (a) to show that for $n \geq n_0$, $T(n) \leq c \cdot f(n)$
   (b) set $c = \max\{2, q\}$ and $n_0 = 1$
   (c) verify base case(s): $T(n_0) = q$
   (d) 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> ... $\leq c \cdot f(n)$
   (e) hence $T(n) = O(f(n))$.

Note that this may not be a tight bound!

Example

$T(n) = 4T(n/2) + n$

1. Assume $T(1) = q$, where q is a constant
2. Show that for $n \geq n_0$, $T(n) \leq c_1 n^2 - c_2 n$
3. Set $c_1 = q + 1$ and $c_2 = 1$ and $n_0 = 1$
4. <u>Base case</u> ($n = 1$): $T(1) = q \leq (q+1)(1)^2 - (1)(1)$
5. <u>Recursive case</u> ($n > 1$):
   - By strong induction: assume $T(k) \leq c_1 \cdot k^2 - c_2 \cdot k$ for $n > k \geq 1$
   - $T(n) = 4T(n/2) + n = 4(c_1(n/2)^2 - c_2(n/2)) + n = c_1 n^2 - 2c_2 n + n$
     $= c_1 n^2 - c_2 n + (1 - c_2)n$
   - Since $(1 - c_2) = 0$, $T(n) \leq c_1 n^2 - c_2 n$

4 Randomized Algorithm

4.1 Random Variables and Expectation

- A discrete random variable is a function from a sample space to the integers
- Expectation: $\mathbb{E}[X] = \sum_i i \cdot \Pr[X = i]$
- Linearity of Expectations: $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ for any 2 random variable X and Y and $\mathbb{E}[aX] = a\mathbb{E}[X]$

4.2 Quicksort

- Divide-and-conquer Algorithm with linear time $\Theta(n)$ partitioning subroutine
- Suppose the pivot produces subarrays of size j and $(n-j-1) \rightarrow T(n) = T(j) + T(n-j-1) + \Theta(n)$
- Worst case will occur when we select the first element of a sorted array as pivot

$$T(n) = T(0) + T(n-1) + \Theta(n) \Rightarrow \Theta(n^2)$$

4.2.1 Average Case Analysis of Quicksort

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

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

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

$= O(n \log n)$ by taking it as area under integration $\square$

4.2.2 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

4.3 Randomized Algorithm

4.3.1 Randomized vs Non-Randomized

- Randomized: output and running time are **functions of the input and random bits chosen**
- Non-Randomized: output & running time are functions of the input only

4.3.2 Types of Randomized Algorithms

1. Randomized Las Vegas Algorithm:
   - Output is **always correct** (e.g. Randomized QuickSort)
   - Running time is a random variable
2. Randomized Monte Carlo Algorithm
   - **Output may be incorrect** with small probability
   - Running time is deterministic

4.3.3 Randomized Quicksort

- Choose random pivot
- Probability that the run time of Randomized Quick Sort exceeds average by $x\% = n^{-\frac{x}{100}} \ln \ln n$
- P(run time of randomized quicksort is double average) $= 10^{-15}$ for $n \geq 10^6$

4.3.4 Analysis of Randomized Quicksort

1. Let T_π be the number of comparisons made by the algorithm when the input permutation is π. T_π is a random variable.
   $T_\pi = n - 1 + T_{\pi_l} + T_{\pi_r}$ where $n-1$ is the partition algorithm, π_l and π_r are the left and right parts of the partition
   - Note that the left and right partition are of random length due to the random choice of the pivot
2. $\mathbb{E}[T_{\pi_l} + T_{\pi_r}] = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}[T_{\pi_l} | q = i] + \mathbb{E}[T_{\pi_r} | q = i] \leq \frac{1}{n}\sum_{i=1}^{n} E(i-1) + E(n-i)$
3. $E(n) \leq n - 1 + \frac{1}{n}\sum_{i=1}^{n} E(i-1) + E(n-i) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} E(i) = O(n \lg n)$
4. Very high probability that randomized quick sort will run in $n \lg n$ time which is the case for the average-case quicksort!

5 Hashing

5.1 Dictionary Data Structure

- 3 different types:
  - **Static**: fixed set of inserted items fixed; only care about queries
  - **Insertion-only**: only insertions and queries
  - **Dynamic**: insertions, deletion and queries
- Implementations
  - **sorted list** (static) - Query takes $O(\log n)$ using binary search
  - **balanced search Trees** (dynamic) - $O(\log n)$ for all operations
  - **Direct Access Table**
    - Needs items to be represented as non-negative numbers (i.e. prehashing required)
    - Huge space requirement (same as universe size)
    - Operations are all $O(1)$

5.2 Hashing

- **Hash function**: $h : U \to \{1, ..., M\}$ gives the location of where to store in hash table. Basically maps the universe into 1 of M buckets
- **Collision**: for 2 different keys x, y, $h(x) = h(y)$
  - Resolve collisions by using **chaining, open addressing** etc.
- **Desired Properties**:
  - ✓ minimise collisions - `query(x)` and `delete(x)` take $\Theta(|h(x)|)$. Worst case is when all keys hash to the same location in which case operations will take $\Theta(N)$ time
  - ✓ minimuse storage space - aim to have $M = O(N)$, where N is the # of stored items
  - ✓ function h is easy to compute - assume $h(x)$ runs at 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*): If every slot in hashtable had $< N$ elements, then $|U| \leq (N-1)M$ which is a contradiction.

5.3 Randomization

To prevent adversary, we randomize the hash function!

5.3.1 Universal Hashing

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

$$\mathcal{H} \text{ is universal if } \forall x \neq y\text{: } \frac{|h \in \mathcal{H} : h(x) = h(y)|}{|\mathcal{H}|} \leq \frac{1}{M}$$

$$\text{OR } \Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{M}$$

Examples:

* For each of the Pr(collision), we are looking at the Pr(collision) between a and b for each of the hash function

	a	b
h_1	0	0
h_2	0	1

Pr (collision) = 1/2

	a	b
h_1	0	1
h_2	1	0

Pr (collision) = 0

	a	b
h_1	0	0
h_2	1	0
h_3	0	1

Pr (collision) = 1/3 ≤ 1/2

Universal

	a	b
h_1	0	0
h_3	1	1

Pr (collision) = 1 ≥ 1/2

	a	b	c
h_1	0	0	1
h_2	1	1	0
h_3	1	0	1

Pr (collision) = 2/3 ≥ 1/2

Not Universal

- For a single, *deterministic*, hash function from a universal family $\mathcal{H}$, if x, y are chosen uniformly from universe U, the probability of $h(x) = h(y) = 1$. e.g. In all-zero function, everything will map to 0
- It is possible for a uniform family of hash function to NOT be universal. e.g. Let h_i be the hash function that maps all of U to i. Then $h_1, ..., h_i$ is uniform but not universal

5.3.2 Collision Analysis for Universal family of hash

Claim: For any N elements $x_1, ..., x_N$, the expected number of collisions between x_N and the other elements is $< \frac{N}{M}$

Proof

1. For $i < N$, let $A_i = 1$ if $h(x_i) = h(x_n)$ and 0 otherwise
2. $\mathbb{E}[A_i] = 1 \cdot \Pr[A_i = 1] + 0 \cdot \Pr[A_i = 1] \leq \frac{1}{M}$
3. # of collisions with x_N is $\sum_{i<N} A_i$
4. $\mathbb{E}[\sum_{i<N} A_i] = \sum_{i<N} \mathbb{E}[A_i] \leq (N-1)/M$

- From the claim above, each operation will hence cost $O(1)$ in expectation and by linearity of expectations, total cost of N inserts will be $O(N)$

5.3.3 Construction of universal family

- Supposed U is indexed by u-bit strings, and $M = 2^m$. For any binary matrix A with m rows and u columns: $h_A(x) = Ax (\text{mod} 2)$
  - x is a $u \times 1$ matrix $\Rightarrow Ax$ is $m \times 1$
- Suppose $U = 00, 01, 10, 11$ and $M = 2$.

	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

- In this case, $u = 2$ since it is a 2-bit string, $m = 1$, $A = \{[0,0], [0,1], [1,0], [1,1]\}$

Proof

- Let $x \neq y$ and $z = x - y$. We know that $z \neq 0$.
- We want to show that the probability of collision, i.e. $\Pr_A[Ax = Ay] = \Pr_A[A(x-y) = 0] = \Pr_A[Az = 0] \leq \frac{1}{M}$
- Suppose z is 1 at the i^{th} coordinate and 0 everywhere else. Then Az equals the i^{th} column of A. Since the columns are uniformly random, $\Pr[Az = 0] = \frac{1}{2^m} = \frac{1}{M}$ (probability that every element in the i^{th} column is 0)

Additional Notes:

- In addition to storing the Hash table, using the matrix method will need to store the matrix A.
  - Addtional storage overhead of $\Theta(\log N \cdot \log U)$ bits, if $M = \Theta(N)$

5.4 Perfect Hashing

- **Static Case**: N fixed items in dictionary, $x_1, x_2, ..., x_N$. Want to perform `Query` in $O(1)$ time
- Possible to do it by using **Quadratic Space**, i.e. $M = N^2$

Claim: If $\mathcal{H}$ is universal and $M = N^2$, then if h is sampled uniformly from $\mathcal{H}$, expected number of collision is < 1

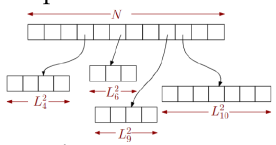
Proof:

- For $i \neq j$, let $A_{ij} = 1$ if $h(x_i) = h(x_j)$ and 0 otherwise
- By universality, $\mathbb{E}[A_{ij}] = \Pr[A_{ij} = 1] \leq \frac{1}{M} = \frac{1}{N^2}$
- $\mathbb{E}[\#c\text{collisions}] = \sum_{i \neq j} \mathbb{E}[A_{ij}] \leq \binom{N}{2} \frac{1}{N^2} < 1$

5.4.1 2-level Scheme

- Choose $h : U \to [N]$ from a universal hash family
- Let L_k be the number of x_i's for which $h(x_i) = k$, i.e. L_k is the number of elements from U that all maps to a bucket $k \in N$
- Choose $h_1, ..., h_N$ second-level hash functions $h_k : [N] \to [L_k^2]$ such that there are no collisions among the L_k elements mapped to k by h

Example of 2-Level Scheme



Claim: If $\mathcal{H}$ is universal, then if h is sampled uniformly:

$$\mathbb{E}\left[\sum_k L_k^2\right] < 2N$$

Proof:

- For $1 \leq i, j \leq N$, define $A_{ij} = 1$ if $h(x_i) = h(x_j)$ and $A_{ij} = 0$ otherwise
- **Crucial Observation**: $\sum_k L_k^2 = \sum_{i,j} A_{ij}$
- $\mathbb{E}[\sum_{ij} A_{ij}] = \sum_i [A_{ii}] + \sum_{i \neq j} \mathbb{E}[A_{ij}] \leq N \cdot 1 + N(N-1) \cdot (\frac{1}{N}) < 2N$

6 Amortized Analysis

- **Amortized analysis** guarantees the *average* performance of each operation in the *worst case*
- It is a strategy for analysing a sequence of operations to show the average cost of operation is small even tho a single operation within the sequence might be expensive
  - Note that in Amortized Analysis, there are **no randomness involved** and is **only used for deterministic algorithm**
  - DO NOT get confused with average-case analysis which is used for random input
  - DO NOT get confused with expected run time which is used for probabilistic algorithms
- Total amortized cost provides an *upper bound* on the total true cost

6.1 Types of Amortized Analysis

6.1.1 Aggregate Method

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

6.1.2 Accounting Method

- Charge i^{th} operation a fictitious amortized cost $c(i)$
- Idea is that amortized cost $c(i)$ is a fixed cost for each operation, whereas true cost $t(i)$ varies depending on when operation is called
- Amortized cost $c(i)$ must satisfy:

$$\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i) \text{ for all } n$$

- Typically fixed amortized cost $c(i)$ will be more than true cost $t(i)$.
- Extra amount we pay for cheap operation can be use to pay for the expensive operation
  - **Invariant**: Bank account is always > 0
- NOTE: Different operations can have different amortized costs.

Example: Queues

- `INSERT` have amortized cost of 2 (true cost of 1)
- `EMPTY` have amortized 0 (true cost is size of queue)
- Whenever an element is inserted, we pay 1 extra to be used for deleting it later
- Total cost is at most 2*#INSERT $\leq 2n$

6.1.3 Potential Method

- ϕ: Potential function associated with algorithm/data structure
- $\phi(i)$: Potential at the i^{th} operation
- Important condition to be fulfilled by ϕ: $\phi(i) \geq 0$ for all i
- Amortized cost of ith operation $\overset{\text{def}}{=}$ True cost of ith operation $+ \Delta\phi_i$
- Typically, $\phi(0) = 0$
- To find for a suitable ϕ, try to find something that decrease during the costly operation

Example: Binary Counter

- Actual cost of ith operation = 1 + Length of longest suffix with all 1's
- $\phi(i)$: Number of 1's in the counter after the ith increment

True ith increment	$\Delta\phi_i$	Amortized cost of ith increment
$l_i + 1$	$-l_i + 1$	2

- Amortized cost of n increments $= 2n = O(n)$ **if starting from all 0s**
- Amortized cost of n increments $\leq O(n+t)$ **if starting from t 1s**

7 Miscellaneous

7.1 Useful Facts

- $e^x \geq 1 + x$
- $(1 - \frac{1}{n})^n \approx \frac{1}{e}$
- $\sum_{j=0}^{\lg(n-1)} 2^j = 2^{\lg(n-1)+1} - 1 \leq 2(n-1)$
- $a = b^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c b$
- $\sum_{i=0}^{\lg n} \frac{1}{\lg n / a^i} = \lg\lg n$
- $(\lg n)! \leq \lg n^{\lg n} = 2^{\lg n \lg\lg n}$
- $\sum_{i=1}^{\lg n} \lg\lg \frac{n}{i} = \lg n \lg\lg n$
- $\log_b a^n = n \log_b a$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $\log_b(1/a) = -\log_b a$
- $\log_b a = \frac{1}{\log_a b}$
- $a^{\log_b c} = c^{\log_b a}$
- $\frac{d}{dn} \lg\lg n = \frac{1}{n \lg n}$
- $n^2 \log_n n! = n^2 \times \frac{\lg n!}{\lg n} = n^3$

7.2 Approximations and Series

- Stirling's Approximation:
  - $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$
  - $\log(n!) = \Theta(n \log n)$
- Arithmetic Series: $\sum_{k=1}^{n} = 1 + 2 + 3 + \cdots + n = \frac{1}{2}n(n+1) = \Theta(n^2)$
- Harmonic Series: $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\lg(n))$
- Geometric Series: $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$
- Geometric Series: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ when $|x| < 1$
- L'Hopital's Rule: $\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$

7.2.1 Arithmetic and Geometric Series

	Arithmetic	Geometric
a_n	$a_n = a_1 + d(n-1)$	$a_n = a_1 \cdot r^{n-1}$
S_n	$S_n = \frac{n}{2}(a_1 + a_n)$	$S_n = a_1\left(\frac{1-r^n}{1-r}\right)$
Infinite Series		$S_\infty = \frac{a_1}{1-r}, \lvert r \rvert < 1$

7.3 Permutations and Combinations

- nPr = n! / (n-r)!
- nCr = n! / (n-r)!r!
- $\binom{2n}{n} \approx 2^{2n}/\sqrt{n}$

7.4 Coupon Collector Problem

- There are n types of coupon given out by the store
- Each time you visit the store you get 1 ticket at random
- What is the expected number of visits before you collect at least one of each type of coupon
- Pr(new coupon collected) $= \left(\frac{n - (i-1)}{n}\right)$
- Let T_i be the # of trials before success
- $\mathbb{E}[T_1 + T_2 + ... + T_n] = \mathbb{E}[T_1] + \mathbb{E}[T_2] + ... + \mathbb{E}[T_n] = n/n + n/(n-1) + ... + n/(n-(n-1)) = n(1 + 1/2 + ... + 1/n) = O(n \lg n)$

7.5 n bins n balls problem

Suppose that you throw n balls uniformly and independently at random into n bins. What does the expected fraction of bins with exactly 3 balls converge to as $n \to \infty$

- Pr[ball go into bin k] $= \frac{1}{n}$, Pr[ball goes into any other bin] $= 1 - \frac{1}{n}$

  Pr[3 balls go into 1 bin] $= \binom{n}{3}(\frac{1}{n^3})(1 - \frac{1}{n})^{n-3}$

  $= \frac{n(n-2)}{6(n-1)^2}(1 - (1/n))^n$
- $(1 - \frac{1}{n})^n \approx \frac{1}{e}$, $\lim_{n \to \infty} \frac{n(n-2)}{6(n-1)^2} = 1/6$, limit $= \frac{1}{6e}$

7.6 Potentially useful algos

7.6.1 Karp Rabin (find substrings)

- Given a text `txt[0...n-1]` and a pattern `pat[0...m-1]`, write a function `search` that prints all occurrences of pat[] in txt[]. You may assume that n > m.
- Idea is that you can hash the patterns as well as substrings in original array so that you can efficiently find the patterns in the array without having to manually loop through each element and then checking whether it matches the pattern
- Let the length of each pattern be k. Runtime will be $O(nk + mk)$.

7.7 Adversarial Argument

7.7.1 Merge needs $2n - 1$ comparisons

Question: Give an adversarial argument to show that at least $2n - 1$ comparisons are needed to merge two sorted arrays $A = [A1, A2, ..., An]$ and $B = [B1, B2, ..., Bn]$ into one sorted array by any comparison-based algorithm.

1. Take $A = [1, 3, 5, ..., 2n-1]$, $B = [2, 4, 6, ..., 2n]$
2. Suppose $\mathcal{M}$ outputs the correct sorted array $[1, 2, ..., 2n-1, 2n]$ where $\mathcal{M}$ makes $< 2n - 1$ comparisons
3. There must exist 2 consecutive elements in the error that were never compared, 1 element from A and 1 element from B
4. Suppose "3" and "4" were the ones not compared
5. Set $A' = A, B' = [2, 2.99, 6, ..., 2n]$, where 4 is replaced by 2.99.
6. We know that "3" in A and 2.99 in B will not be compared and thus $\mathcal{M}$ will output $[1, 2, 3, 2.99, 5, ..., 2n-1, 2n]$
7. Therefore must have $2n - 1$ comparisons