**Algorithm Analysis**

GP to infinity: **|**

AP:

nCr =

Sum of squares:   
Harmonic Series:

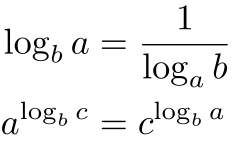
n! = O(nn)

< < n0.25 < < < < < < < < < <

Poly dominates logs:

Exp dominates polys:

,



**Big O: f(n) = O(g(n))** if there exist constants c > 0, n0 > 0 such that 0 <= f(n) <= cg(n) for all n >= n0.

* Define c and n0 such that for all n >= n0, eqn holds

**Big Omega: f(n) = 𝛀(g(n))** if there exists constants c > 0, n0 > 0 such that 0<=cg(n)<=f(n) for all n >= n0.

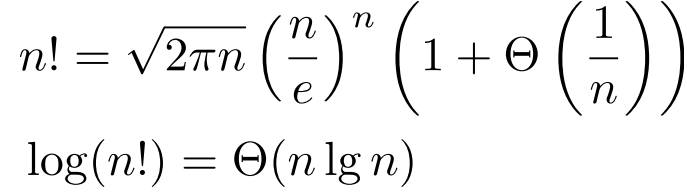
**Big Theta: f(n) = Θ(g(n))** if there exists constants c1, c2 and n0 such that 0 <= c1g(n) <= f(n) <= c2g(n) for all n >= n0

**Little o: f(n) = o(g(n)))** if there exists a constant n0> 0 such that for *any constant c > 0*, 0 <= f(n) < cg(n) for all n >= n0

* Define n0 such that for any c > 0 and for all n >= n0, eqn holds

**Little-Omega: f(n) = 𝛚(g(n))** if there exists a constant n0 such that for any constant c > 0, 0<=cg(n)<f(n) for all n >= n0

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

Stirling’s Approx.: 

**Iteration, Recursion, and Divide-and-Conquer**

T(n) = 2T() + 1 = O(log n)

T(n) = 2T() + log n = O(log n log log n)

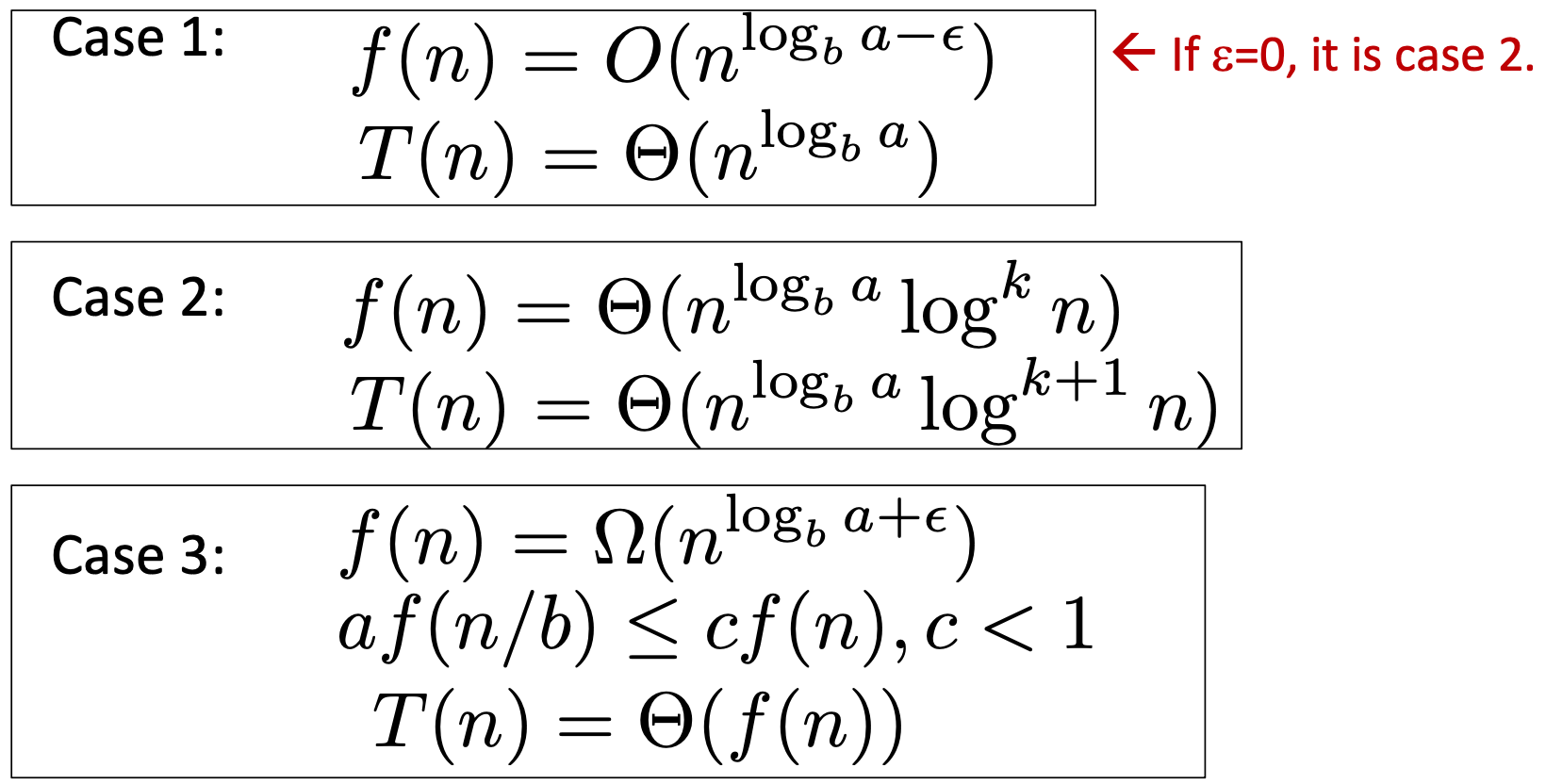
T(n) = 2T() + n = O(n) + O( log log n) = O(n)

T(n) = T() + 1 = O(log log n) ⇒ log log n levels

T(n) = 2T(n-1) + 1 = O(2^n)

**Master method:** T(n) = a T(n/b) + f(n) where a >= 1, b > 1 and f is asymptotically positive.

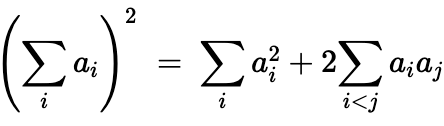
Compare f(n) with n:



**Randomization**

Bernoulli trials before the first success: expected no. = 1/p

Pr[X = k] =



**Dynamic Programming**

**Optimal Substructure:** An optimal solution to a subproblem contains optimal solutions to subproblems

* Cut-and-paste: optimal = S. Suppose the subproblem T has better solution, then by adding back, the original problem will have better solution than the optimal ⇒ contradict S is optimal
* e.g. Assume vk is a vertice on the shortest path p(v0, vn). So p(v0, vk) + p(vk, vn) = p(v0, vn).
* Ass8Q2 price increase/day (optimal substructure): Suppose S is an optimal solution, and item i is the last item in S. Then, S[1, . . . , n − 1] is an optimal solution for the subproblem where the shop initially does not contain item i.
* Ass8Q2 optimal substructure proof: Using the cut-and-paste argument, if T is an optimal solution for the subproblem and has a total cost less than S[1,...,n−1], then T +[i] will have a total cost less than S, as the cost of buying item i is the same for S and T + [i]. ⇒ contradiction to S being an optimal solution

**Overlapping Subproblems:** A recursive solution contains a “small” number of distinct subproblems repeated many times

* E.g. Show which solution of a subproblem is calculated multiple times
* Ass7Q1 Hamming Distance: Observe that for any vertices U and V with binary representations that differ at exactly 2 positions i and j, if Ui = 1, Vi = 0 and Uj = 0, Vj = 1, then there is a vertex W that differs from U only at i and differs from V only at j with Wi = Wj = 0, such that there are edges from W to U and W to V . Then, the computation of T(U) and T(V ) both require T(W).

**Greedy Algorithms**

1. Cast the problem such that we have to make a choice and are then left with 1 subproblem to solve
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so greedy choice is safe.
3. Use optimal substructure to show that we can combine greedy choice to get an optimal solution to the original problem.

E.g. MST

Optimal Substructure: Any subtree of the MST is an MST of the graph induced by its vertices.

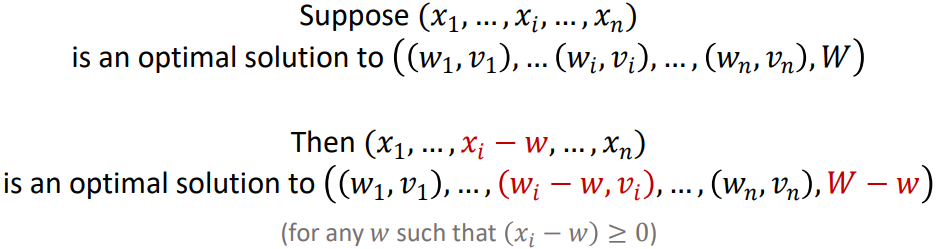
* Shows problem can be broken down

Greedy choice property: The least weight edge connecting any set of vertices to its complement is contained in the MST

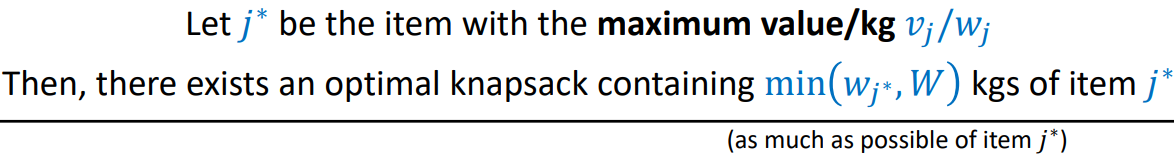
* Shows local optimal = global optimal

E.g. Fractional Knapsack

Optimal Substructure:



Greedy Choice Property:



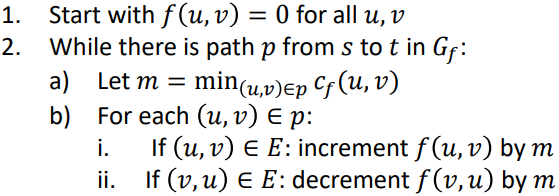
**Max Flow**

Given flow network 𝐺 = (𝑉, 𝐸) and capacities 𝑐, find a flow 𝑓 that maximizes the value |𝑓|

Applications

* Matching Problems: Add dummy nodes ⇒ source, sink which connect to starting/ending nodes with capacity 1
* Binary Search with Flows: Connect a supernode to original start nodes and find optimal capacity using binary search and check if input flow = output flow

Ford-Fulkerson Algorithm



Given flow 𝑓, define residual capacities 𝑐𝑓:

* If 𝑢, 𝑣 ∈ 𝐸: 𝑐𝑓(𝑢, 𝑣) = 𝑐𝑓(𝑢, 𝑣) − 𝑓(𝑢, 𝑣)
* If 𝑣, 𝑢 ∈ 𝐸: 𝑐𝑓(𝑢, 𝑣)=𝑓(𝑣, 𝑢)
* If neither, 𝑐𝑓(𝑢, 𝑣) = 0

T(n) = O(|E|\*|fmax|) where fmax is the maximal flow

Min Cut

1. Partition of 𝑉 into any 𝑆 and 𝑇 such that 𝑠 ∈ 𝑆 and 𝑡 ∈ 𝑇 is called a cut
2. Capacity of a cut is the total capacity of all edges going from 𝑆 to 𝑇
3. Any cut is a bottleneck for any flow 𝑓, 𝑓 ≤ capacity of min cut
4. For 𝑢 ∈ 𝑆,𝑣 ∈ 𝑇
   * If (𝑢,𝑣) ∈ 𝐸:𝑓(𝑢, 𝑣) = 𝑐(𝑢,𝑣);
   * If (𝑣, 𝑢) ∈ 𝐸:𝑓(𝑢, 𝑣) = 0
5. |fFF| = total capacity of edges going from 𝑆 to 𝑇
6. |𝑓FF| ≥ capacity of this cut → combine with (3): 𝑓 = capacity of min cut = value of max flow

**Linear Programming**

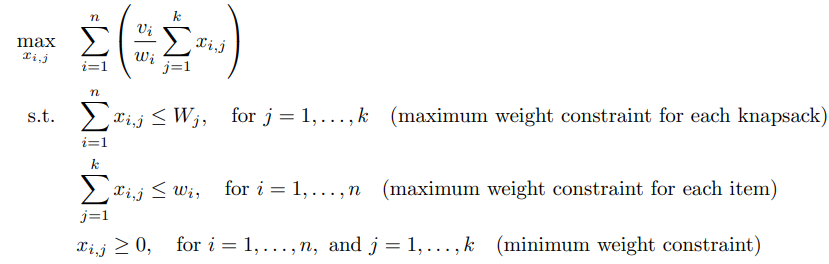
Standard Form of LP:

Subject to:

[inequalities must be <=]

* Except for x >= 0, other inequalities all <=
* E.g. x + y = 20 → x + y <= 20 & -(x+y) <= -20
* If qns is to minimise, convert to maximum and revert all inequalities
* Optimal solution (if one exists) can be found in 𝑝𝑜𝑙𝑦(𝑛, 𝑚) time

Fractional Knapsack for k knapsacks example:



* After solving the LP and obtaining the best weights x∗i,j, directly computing the objective function with those weights gives the max total value V .
* In general, trying to solve the LP problem (using simplex algorithm or otherwise) with nk variables might take polynomial or even exponential time in nk, so it would not be more efficient than the greedy method

**Reductions**

**Pseudo-polynomial:**

1. Polynomial in numerical input length
2. Exponential in binary input length

* If array, num input = O(mn) where m is max elem

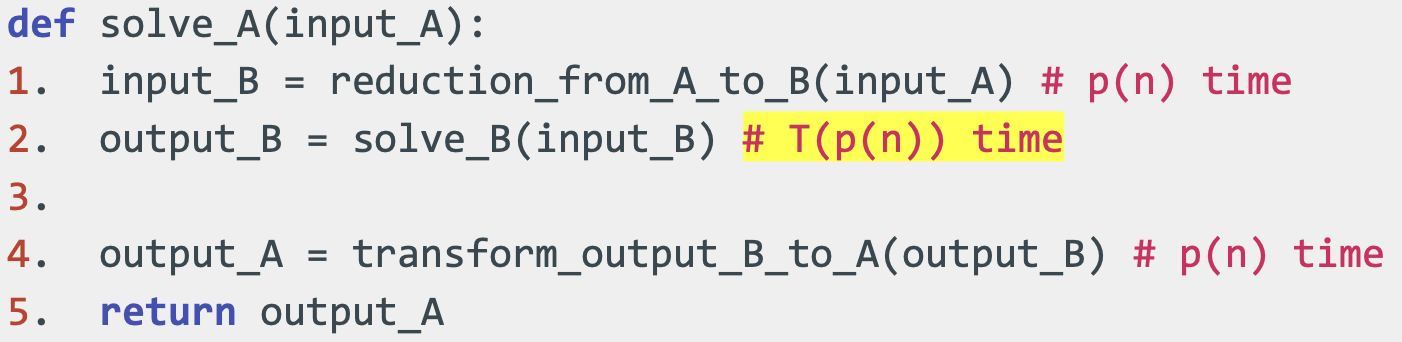
Ass11Q1a

For m = max{x, y}, the length of the input is ℓ = Θ(log m).

* Since the largest possible value for z is m, T(n) = O(m).
* The runtime is polynomial in the input values, and is exponential in ℓ as O(m) = O(cℓ) for some constant c.

**Polynomial-Time Reduction:**

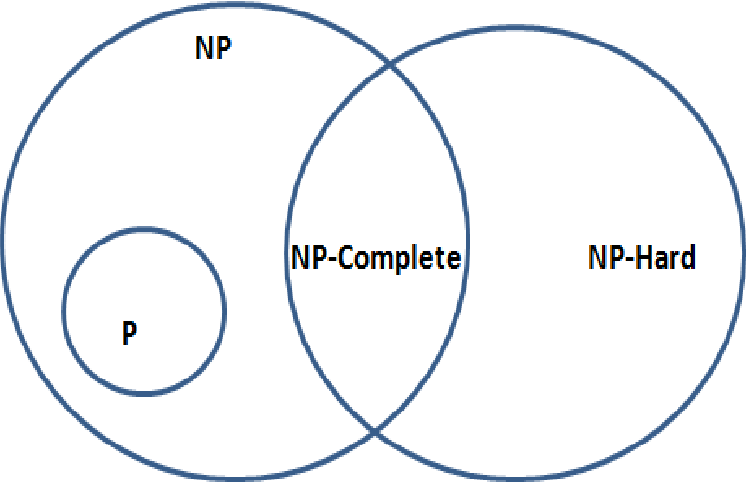
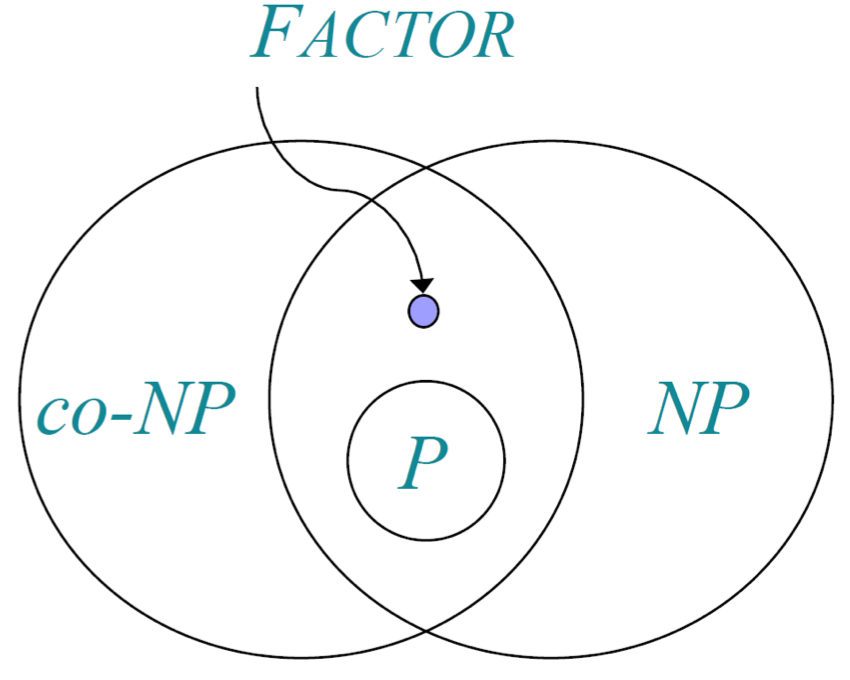
* : A can be reduced to B in polynomial time
* If A is “hard”, then B is “at least as hard” as A
* If A is “easy”, cannot conclude anything about B
* Assume B takes T(n), time to solve A = T(p(n)) + 2p(n)



Suffices to show:

* Reduction from A to B runs in polynomial time
* If 𝛼 is a YES-instance of 𝐴, 𝛽 is a YES-instance of 𝐵
* If 𝛽 is a YES-instance of 𝐵, 𝛼 is a YES-instance of 𝐴

**NP-Completeness**

| NP | * Verifiable certificates of YES-instance exist in poly time i.e. can check that solution is correct in polynomial time * Certificate y with |y| = poly(|x|) |
| --- | --- |
| co-NP | * polynomial time verifiable certificates (“counterexamples”) of NO instances exist. |
| NP-Complete | * In both NP and NP-Hard * Proof: Can be reduced **from** another known NP-Complete problem * E.g. CNF-SAT, 3-SAT, Vertex Cover, Independent Set, Hamiltonian Cycle, TSP, SubsetSum |
| NP-Hard | * A is NP-Hard if for any B in NP: |

True-False

1. If A is NP-Complete and B is NP-Complete then A ≤p B and B ≤p A → True
2. If A is NP-Hard and A ∊ P, then A is NP-Complete → True since P ⊆ NP
3. If A in NP-Complete and A can be reduced to a problem B ∈ NP, then B is also NPComplete → False. Must ensure that reduction is also poly time
4. P ⋃ NP-Complete = NP → Undetermined. True if P = NP, false if P =/= NP e.g. Factor
5. Suppose that P = NP. Then, there exists a polynomial time algorithm for IndependentSet. → True. Every NP problem would have a poly time algo.

Independent Set [NP-Complete]

Given a graph G = (V, E), and int k, is there a subset of >= k vertices such that no two are adjacent

Vertex Cover [NP-Complete]

Given a graph G = (V, E), and int k, is there a subset of <= k vertices such that each edge is incident to at least one vertex in the subset (vertices can cover all edges)

* G has indep set of size k G has a vertex cover of size (n - k)

Set Cover [NP-Complete]

Given int k and n, and a collection S of subsets {1, …, n}, are there <= k of these subsets whose union equals {1, …, n}

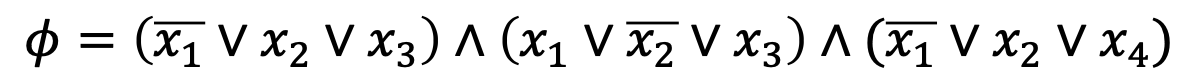
Satisfiability [NP-Complete]

* Conjunctive Normal Form (CNF) formula: a “formula” 𝜙 that is a conjunction (AND) of clauses e.g. 𝝓 = 𝑪𝟏 ∧ 𝑪𝟐 ∧ 𝑪𝟑 ∧ 𝑪𝟒
* Satisfying assignment: given a formula 𝜙 over variables 𝑥1, ... , 𝑥𝑛, an assignment of values 𝑇, 𝐹 that makes the entire formula evaluate to 𝑇

SAT: Given a CNF formula 𝜙 over 𝑛 variables, does it have a satisfying assignment?

3-SAT [NP-Complete]

SAT where each clause in the given formula contains exactly 3 literals corresponding to different variables



**Approximation**

C\*: Cost of optimal solution

C: cost of solution found by your algo

Minimization approximation ratio: C\*/C; always >= 1

Maximisation approximation ratio: C/C\* [inverse of min problem]

Extra

* Ass7Q1 Hamming dist: For any vertex V != O, T(V) = minU∈S{T(U)+C(U,V)} = smallest travelling cost, where S = set of vertices with binary rep differing from V at exactly one position s.t. num at that position of V is 1.
* Ass7 Q1 proof: ​​For any vertex U , for there to be an edge from U to V , their Hamming distance is 1, so their binary representations differ at exactly one position. Moreover, the direction of the edge is to a vertex of higher value, so the number at that position must be 0 for U and 1 for V .
* Then, any path from O to V must be a path from O to some U∈S, and then adding the edge from U to V. Thus, T(V) = min among total costs of paths from O to U to V for all U∈S.

**Hashing**

Desired properties: (N is number of stored items)

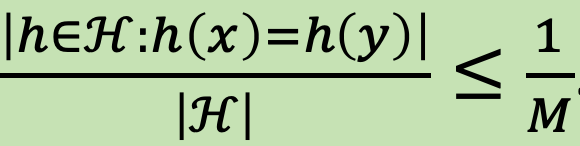
* Minimise collisions: Worst case = θ(N)
* Intuition: h(x) should be a “random” value
* Minimise storage: M = O(N)
* h(x) should be easy to compute

Randomization:

* Select a random hash function from hash family
* Use the same hash function for all elements

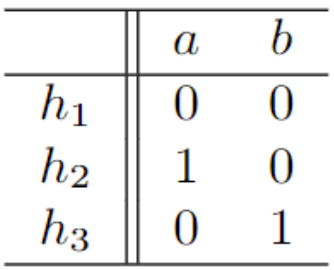
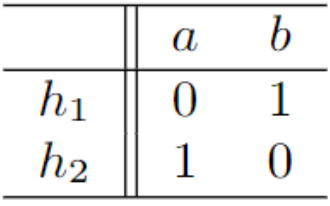
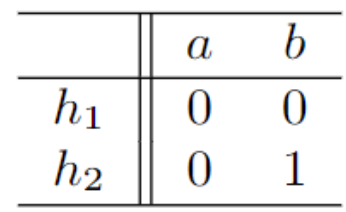
Universal Hashing:

Suppose H is a set of hash functions mapping 𝑈 to [𝑀]. We say H is universal if for all 𝑥 ≠ 𝑦: (Fraction of hash functions for which x and y collide <= 1/M)

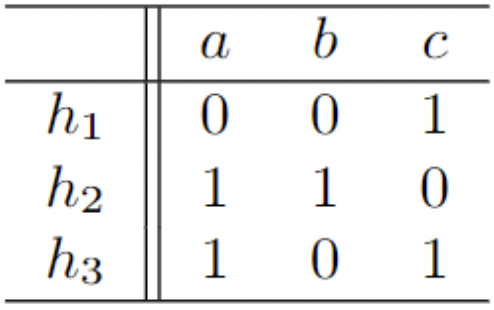
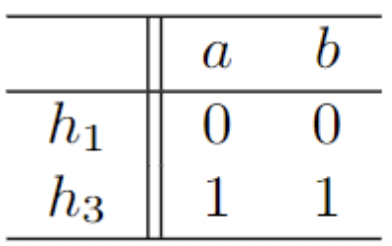


* For any 𝑥 ≠ 𝑦, if **h is chosen uniformly at random** from a universal H, there’s at most 1/M probability that h(𝑥) = h(𝑦).
* If 𝑥, 𝑦 are chosen uniformly from the universe 𝑈, the probability that h(𝑥)=h(𝑦) <= 1/𝑀 ⇒ FALSE: h can be all zero function

Universal hashing e.g.

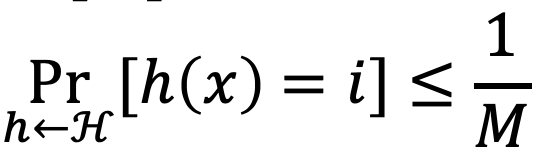


Not Universal



Uniform family of hash functions:

H is said to be a uniform family of hash functions if for every key 𝑥 ∈ 𝑈 and every hash value 𝑖 ∈ [𝑀], it holds that:



* for all 𝑀, there exists a family H that is uniform but not universal.
  + E.g. For every 𝑖 = 1, ... , 𝑀, let h1 be the constant function mapping all of 𝑈 to 𝑖.
  + {h1, ... , hM} is uniform but not universal.

Collision Analysis

Suppose H is a universal family of hash functions mapping 𝑈 to [𝑀]. For any 𝑁 elements 𝑥1,..., 𝑥N , the expected number of collisions between 𝑥1 and the other elements is < .

Expected Cost

Suppose H is a universal family of hash functions mapping 𝑈 to [𝑀]. For any sequence of 𝑁 insertions, deletions and queries, if 𝑀 ≥ 𝑁, then the expected total cost for a random h ∈ H is 𝑂(𝑁).  
Construction of universal family

Suppose 𝑈 is indexed by 𝑢-bit strings, and 𝑀 = 2m. For any binary matrix 𝐴 with 𝑚 rows and 𝑢 columns: hA(𝑥) =𝐴𝑥(mod2)

* {h(x): 𝐴 ∈ {0,1}m✕u} is universal.

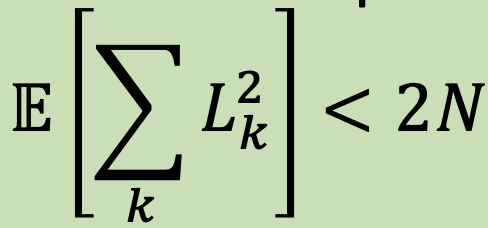
Perfect Hashing: Quadratic Space

If H is universal and 𝑀 = 𝑁2, then if h is sampled uniformly from H, the expected number of collisions is < 1

Perfect Hashing: 2-Level Scheme

* h: 𝑈 → [𝑁] from a universal hash family.
* Let 𝐿k be the number of 𝑥i’s for which h(𝑥i) = 𝑘
* Choose h1,...,hN second-level hash functions hk: [𝑁] → [𝐿k2] such that there are no collisions among the 𝐿k elements mapped to 𝑘 by h.

If H is universal, then if h is sampled uniformly from H:



Pairwise Independent

if, for any 2 distinct universe elements x,y, and for any 2 hash values i1, i2: (probability of a random hash function in H)

Pr[h(x) = i1,h(y)=i2] = 1/M2

**Amortised Analysis**

Analysing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Aggregate Method:

Counting each operation's complexity, then find a pattern and come up with an upper bound.

e.g. k-bit Binary counter: No. of bit flips

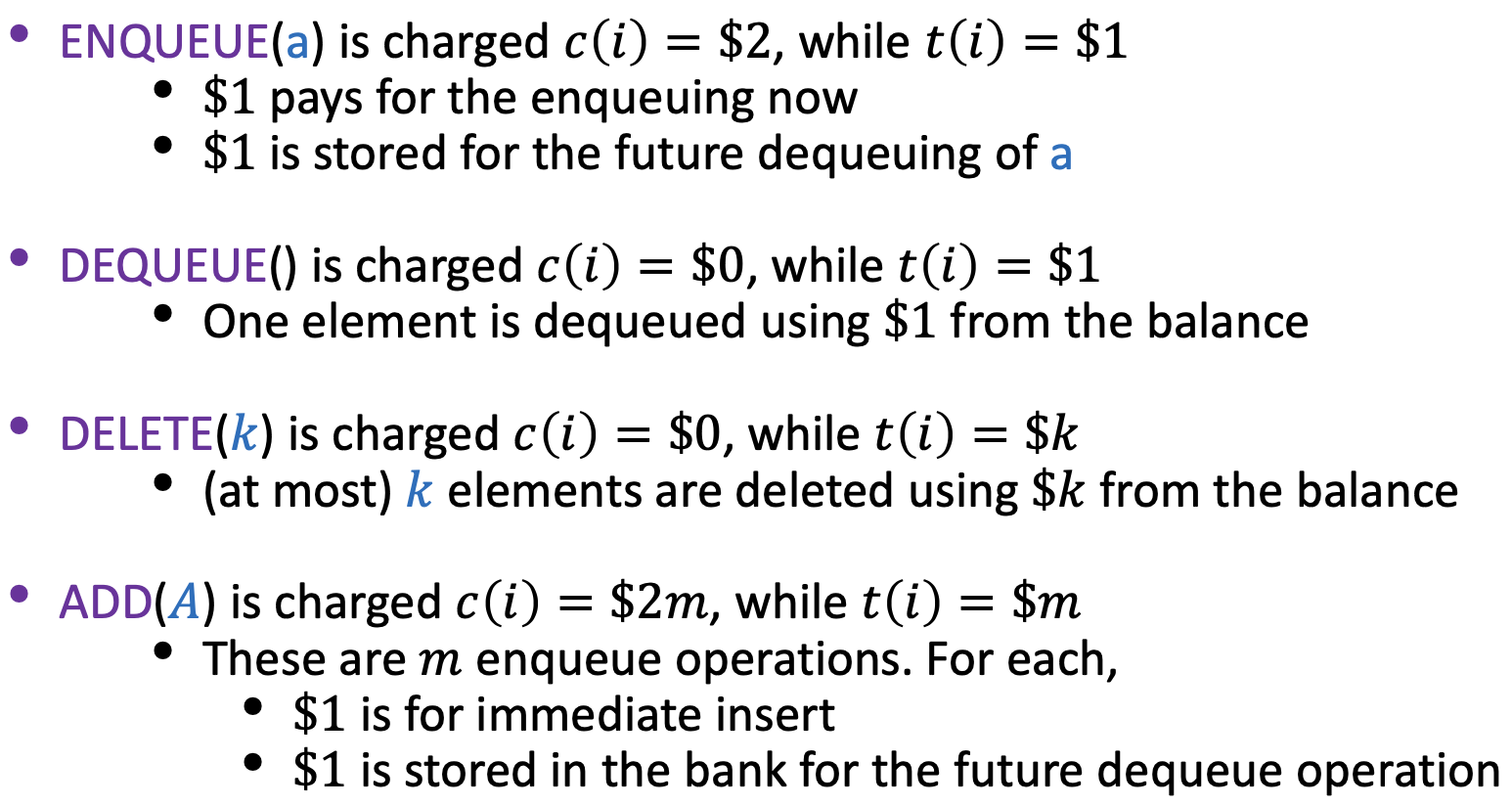
Let f(i) = no. of times ith bit flips

f(0) = n; f(1) = n/2; f(2) = n/4; f(i)=n/2i

Accounting Method:

1. Pay more (“amortised costs”) for inexpensive operations
2. Use balance to pay for the more exp operations later
3. Show that balance never < 0

E.g. n1 ENQUEUE, n2 DEQUEUE, n3 DELETE(k), n4 ADD(A) where |A|=m



Potential Method:

𝜙(𝑖): Potential at the end of the 𝑖-th operation

Conditions to be fulfilled by 𝜙:

* 𝜙(0)=0
* 𝜙(𝑖) ≥ 0, for all 𝑖

c(i) ≝ Actual cost of 𝑖-th operation + (𝜙(𝑖) − 𝜙(𝑖 − 1))

c(i) = Actual cost of 𝑛 operations + (𝜙(𝑛) − 𝜙(0))

* observe the costly operation and see if there is some quantity that is “decreasing” during the operation

E.g. cost of deleting all of the elements in dynamic tables

* Insert: 𝜙(𝑖) = 2i - size(𝑇)

→ table size halved when num elements after del = ½ original

𝜙(𝑖) = size(𝑇) − 𝑛