Graphs

Adjacency List: V = set of vertices; Adj is dictionary s.t. $Adj[u]$ is the list of vertices adjacent to u. Good for sparse graphs where $|E| << |V|^2$ since **space** requirement is $\Theta(V + E)$.

Add/Remove an edge: O(1), Check if (u,v) is an edge: O(# neighbors) = O(E), Neighbors of u: O(E)

Adjacency Matrix: Let $A = (a_{ij})$ be a $|V| \times |V|$ matrix where $a_{ij} = 1$ if $(i, j) \in E$, 0 otherwise. Good for dense graphs where $|E| \approx |V|^2$. **Space required = $\Theta(V^2)$**. Note: To find nodes k edges away: A^k can use this to quickly test if $(i, j) \in E$.

Add/Remove an edge: O(1), Check if (u,v) is an edge: O(1), Visit All Neighbors of u: O(#neighbors) = O(V)

UNWEIGHTED GRAPH EXPLORATION ALGORITHMS

1) Breadth-First Search **Space & Time: $\Theta(V + E)$.**

Explore level by level. Mark vertices once discovered (need space |V|). Given level k, mark all unmarked vertices adjacent to level k; at this point level k + 1 consists of all vertices marked. This finds shortest paths for unweighted graphs.

```
def bfs(top_node, visit):
    """Breadth-first search on a graph, starting at top_node."""
    visited = set()
    queue = deque([top_node])
    while queue:
        curr_node = queue.popleft()          # Dequeue
        if curr_node in visited: continue    # Skip visited nodes
        visit(curr_node)                     # Visit the node
        visited.add(curr_node)
        # Enqueue the children
        queue.extend(curr_node.children)
```

2) Depth-First Search **Space & Time: $\Theta(V + E)$.**

Like exploring a maze. Backtrack when necessary to find unexplored edges. Mark vertices to avoid getting stuck in a loop.

```
DFS(G)
1  for each vertex u in V [G]
2       do color[u] = WHITE
3          p[u] = NIL
4  time = 0
5  for each vertex u in V [G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)

DFS-VISIT(u)
1  color[u] = GRAY     #White vertex u has just been discovered.
2  time = time +1
3  d[u] = time
4  for each v in Adj[u]  #Explore edge(u, v).
5       do if color[v] = WHITE
6             then p[v] = u
7                  DFS-VISIT(v)
8  color[u] BLACK      # Blacken u; it is finished.
9  f [u] = time = time +1
```

As in breadth-first search, vertices are colored during the search to indicate their state. Initially white, grayed when it is discovered in the search, and blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint. **Timestamps**: d[v] records when v is first discovered (and grayed), and the second timestamp f[v] records when the search finishes examining v's adjacency list (and blackens v).

The loops on lines 1-3 and lines 5-7 of DFS take time Θ(V), exclusive of the time to execute the calls to DFS-VISIT. The total cost of executing lines 4-7 of DFS-VISIT is Θ(E). Therefore total runtime = Θ(V + E).

Edge Classification

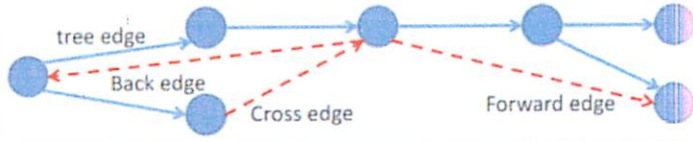
Each edge can be classified by the color of the vertex v that is reached when the edge is first explored:

Tree edge: an edge in a DFS-tree.

Back edge: connects a vertex to an ancestor in a DFS-tree. Note that a self-loop is a back edge.

Forward edge: a non-tree edge that connects a vertex to a descendent in a DFS-tree.

Cross edge: all others. It connects vertices in two different DFS-tree or two vertices in the same DFS-tree neither of which is the ancestor of the other.



Cycles

Theorem: G is a cycle ↔ DFS of G produces no back edges.

Lemma: In DFS, all edges (u, v) that are not back edges have property: v finishes before u.

Detecting cycles: We can tell in Θ(V + E) time whether a directed graph is acyclic. To do this, run DFS, see if any back edges produced.

Topological sort: Given an acyclic graph G, we want to produce a list of vertices s.t. all edges go left to right. To do this, on input G:

1. call DFS(G) to get finishing times f[v] for each vertex v
2. as each vertex is finished, insert it in front of a linked list
3. return the linked list of vertices

Parallel processing: If we can do many tasks at once (as long as prereqs done) how long does it take to do all tasks? (Identify critical path.) To do this, do topological sort, then iterate through sorted list: take elements one at a time, put in earliest time slot possible.

Strongly Connected Components: 1) DFS(G) with finishing-time; 2) get G^T ((u,v) in G → (u,v) in G^T; 3) DFS(G^T) explore nodes in decreasing finish-time order; 4) All trees are SCC.

The graph replacing SCCs of the original directed graph with vertices must be a DAG because if there were cycles in this graph, then all the components of the cycle would be one SCC.

Properties of Shortest Paths

Triangle inequality: For any edge (u, v) ∈ E, we have δ(s, v) = δ(s, u) + w(u, v).

Upper-bound property: We always have d[v] = δ(s, v) for all vertices v ∈ V , and once d[v] achieves the value δ(s, v), it never changes.

No-path property: If there is no path from s to v, then we always have d[v] = δ(s, v) = ∞.

Convergence property: If s → ... → u → v is a shortest path in G for some u, v ∈ V , and if d[u] = δ(s, u) at any time prior to relaxing edge (u, v), then d[v] = δ(s, v) at all times afterward.

Path-relaxation property: If p =< v0 , v1 , . . . , vk > is a shortest path from s = v0 to vk, and the edges of p are relaxed in the order (v0, v1), (v1, v2), ..., (vk−1, vk), then d[vk] = δ(s, vk). This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property: Once d[v] = δ(s, v) for all v ∈ V , the predecessor subgraph is a shortest-paths tree rooted at s. Stated more simply: subpaths of shortest paths are also shortest paths.

Finding Shortest Paths for Weighted Graphs

Problem: On input G = (V, E), weight function w, and a source vertex s, the goal is to find δ(s, v) for every v ∈ V and the best path from s to v.

Strategy: We note that each edge (u, v) ∈ E supplies a constraint: δ(s, v) ≤ δ(s, u) + w(u, v). So we start with some d[v] that satisfies these constraints [illegible] is not minimal, then relax the constraints until we are done. **Relaxing** the [illegible]straint for edge (u, v) means we test to see whether we can improve the shortest path to v found so far by going through u; if so, update d[v] and π[v].

```
for v in V:
d[v] = infinity
pi[v] = None
d[s] = 0
while d[v] > d[u] + w(u,v) for some v:
d[v] = d[u] + w(u,v)  #{relaxation
pi[v] = u #{relaxation
```

Variation: The difference in algorithms is in how they choose the order to relax the vertices. If we just do it randomly, the worst case is exponential: $T(n) = 3 + 2T(n-2) = \Theta(2^{n/2})$, and it may not terminate if there is a negative weight cycle.

Variables we'll use: d[v] = length of best path to v so far. Initialize as d[v] = 0 if v = s, ∞ otherwise. During the algorithm, d[v] ≥ δ(s, v) and when the algorithm terminates, d[v] = δ(s, v). π[v] = predecessor of v on best path. Initialize π[v] = None.

SINGLE SOURCE SINGLE TARGET ALGORITHMS

1) Bidirectional Search **Runtime: O(2T(n/2))**

Alternatingly searching a graph forward from s and backwards from t with Dijkstra's Algorithm allows us to effectively solve a problem with half as many nodes, twice.

Can be used with Dijkstra's Algorithm and other BFS-style algorithms. This can be used to solve the Single-Source Single-Target problem in a faster way. Not asymptotically faster, but still faster. If the SSSP problem is T(n) time, then the SSST problem can be solved in O(2T(n/2)) time. Use two instances of Dijkstra's algorithm.

2) A* Search (Goal-Directed Search)

[illegible]cts search from s toward t with "heuristic function" $\lambda(v)$

[illegible]) is a lower bound on $\delta(v,t)$

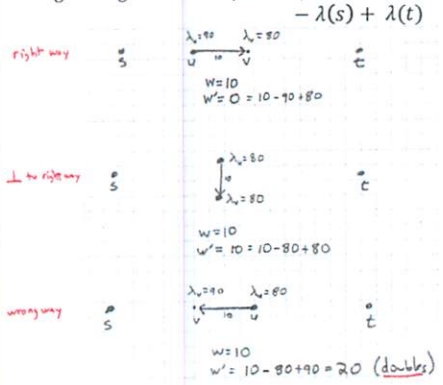
Modifies edge weights:

Uses $w'(u,v) = w(u,v) - \lambda(u) + \lambda(v)$, instead of $w(u,v)$

($w'(u,v)$ *must be* ≥ 0 in order to use Dijkstra)

Changes length of all s-t paths by the same amount:

$$-\lambda(s) + \lambda(t)$$



SINGLE SOURCE SHORTEST PATH ALGORITHMS

Helper Methods

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v in V
2    d[v] = infinity
3    pi[v] = None
4  d[s] = 0
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v):
2     d[v] = d[u] + w(u, v)
3     pi[v] = u
```

Setting	Weights	Principle	Algorithm
Single source	=1	Greedy	BFS: $O(V+E)$
Single source	≥0	Greedy	Dijkstra: $O(E+V\lg$
Single source	General	$\lvert V\rvert$-1 passes	Bellman-Ford: $O($
All pairs	General	DP-length	Matrix Mult: $O(V^$
All pairs	General	DP-vertices	Floyd-Warshall: $O($
All pairs	General	Reweigh	Johnson: $O(VE+V^2\lg V)$

1) GRAPHS W/ NO CYCLES

Topological-Sort **Runtime: Θ(V + E)**

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       do for each vertex v in Adj[u]
5             do RELAX(u, v, w)
```

2) GRAPHS W/ POSITIVE WEIGHT EDGES

Dijkstra Algorithm Using Fibonacci Heap **Θ(Vlog V + E)**

Decreasing keys: Θ(1) amortized

Dijkstra Using Min Heap **Θ(VlogV + ElogV) ~ Θ(ElogV)**

Uses a heap. Need functions *{initialize, extract min, and decrease key}*.

Initialization/build heap: Θ(V).

Extract minimums: Θ(V) calls of Θ(log V). Total: Θ(V log V).

Relaxations: Θ(E).

Decreasing keys: O(ElogV).

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  d[s] = 0
3  Q = build_heap(d)
4  for i in range(n-1):
5    u = extract_min(Q) #index, not value
6    for(u,v) in E:
7      if d[v] > d[u] + w(u,v)
8        d[v] = d[u] + w(u,v)
9        decrease_key(Q,v,d[v])
```

3) GRAPHS W/ NEGATIVE WEIGHT EDGES

The shortest path cannot contain negative-weight cycle or positive weight cycle.

Bellman-Ford **Runtime: O(VE)**

Initialization: Θ(V) time

each of the|V | − 1 passes over the edges: Θ(E) time (Θ(E) edges and Θ(1) time per edge)

negative weight cycle detection takes O(E) time.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  do n-1 times:
3       for each edge (u, v) in E:
4             do RELAX(u, v, w)
5  for each edge (u, v) in E[G]
6       do if d[v] > d[u] + w(u, v)
7             then negative weight cycle exists
8  return TRUE
```

Dynamic Programming

Overlapping subproblems are encountered when a problem can be broken down into several subproblems that are reused. Examples: Fibonacci computation, shortest-path algorithms.

roblem exhibits **Optimal substructure** if an optimal solution to the problem tains within it optimal solutions to subproblems.

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. However, unlike the divide-and-conquer method, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

Can carry out dynamic programming through two strategies: **top-down + memorization** (recursive implementation, requires test to see if we have stored memoized solution and avoid recomputing results) or **bottom-up** (order sub-problems in a way that allows answering bigger sub-problems using already computed solutions to smaller sub-problems).

Subproblem dependence graph: vertices are subproblems to solve, edge from A to B represents dependence of A on B (solving A requires solving B first). If the graph is a DAG, we can use DP. **Top-down + memorization** is DFS, **bottom-up** is solving problems in order determined by reverse of topological sort. (Both are **Θ(V + E)**)

Memoization is the technique where a function "remembers" the results corresponding to a set of specific inputs, and when called again on the same inputs, returns the remembered result rather than recomputing.

5 Steps to Dynamic Programming

1. Define subproblems (count # of subproblems)
2. Guess (part of the solution) (count # of choices)
3. Relate subproblem solutions (compute time/subproblem)
4. Recurse + memoize OR build DP table bottom up (time = time/subproblem × # subproblems)
5. Solve original problem: a subproblem OR by combining subproblem solutions

2 kinds of Guessing

A. In Step 3, guess which other subproblems to use
B. In Step 1, create more subproblems to guess/remember more structure of solutions
   a. Effectively report many solutions to subproblems
   b. Lets parent subproblem know features of solutions

Useful subproblems for strings/sequences X:

- Suffixes X[i:] – Θ(|X|)
- Prefixes X[:i] – Θ(|X|)
- Substrings X[i:j] – Θ($|X|^2$)

All pairs shortest paths Runtime: O(n^3)

- Problem: Given directed graph $G = (V, E)$, where $V = \{1,\ldots, n\}$, with edge-weight function $w : E \rightarrow R$. Return An $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.
- Setup: Consider the $n \times n$ matrix $A = (a_{ij})$, $a_{ij}=w(i,j)$, if $(i, j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- Base Case: $d_{ij}^{(0)} = a_{ij}$;
- Recurrence:
  $d_{ij}^{(m)}$ =weight of a shortest path from i to j that only uses intermediate vertices from set $\{1,\ldots,m\}$
  For $m = 1, 2, \ldots, n$, $d_{ij}^{(m)} = \min\{d_{ij}^{(m-1)}, d_{im}^{(m-1)} + d_{mj}^{(m-1)}\}$.
- Return: $d_{ij}^{(n)}$

Longest Common Subsequence Runtime: $O(n \times m)$

- Problem: given two sequences x[1..m] and y[1..n], find a longest subsequence LCS(x,y) common to both
- Recurrence:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1, & if\ x_i = y_i \\ \max\{c[i-1,j], c[i,j-1]\}, & otherwise \end{cases}$$

- Bottom up

x: A B C B
y: B D C

	y_j	B	D	C
x_i	0	0	0	0
A	0	↑0	↑0	↑0
B	0	↖1	← 1	← 1
C	0	↑1	↑1	↖2
B	0	↖1	↑1	↑2

Knapsack Runtime: O(nS), pseudo-polynomial time on S

- Problem: Given knapsack of size S, a collection of n items, each item i has size s_i and value v_i Choose subset with $\sum_i s_i < S$ (feasible, i.e. fits in knapsack) and maximize $\sum_i v_i$
- Setup: Consider the $n \times n$ matrix $A = (a_{ij})$, $a_{ij}=w(i,j)$, if $(i, j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- Base Case: $d_{ij}^{(0)} = a_{ij}$;
- Recurrence:
  Val[i, X] = max value if one can choose from items[$i : n$] and available size is X

$$Val[i,X] = \begin{cases} Val[i+1,X], & if\ s_i > X \\ \max\{Val[i+1,X], v_i + Val[i+1, X-s_i]\}, & otherwise \end{cases}$$

- Return: Val[1, S]

Text Justification/Word Processing Runtime: O(n^2)

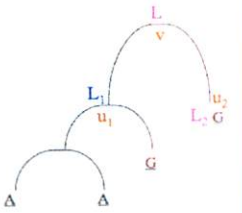
- Problem: input: array of word lengths w[1..n], split into lines L_1, L_2 …
- Approach: 1) formalize layout as an optimization problem, 2) define a scoring rule, 3) takes as input partition of words into lines, 4) measures how good the layout is, 5) find the layout with best score
- Badness of a line: badness(L) = (page width – total length(L))^3 OR ∞ if total length of line > page width
- Objective: break into lines L_1, L_2… minimizing $\sum_i badness(L_i)$
- Recurrence:
  - DP[i] = min badness for words w[i:n], n subproblems where n is # words
  - DP[i] = $\min_{j \text{ in range}(i+1,n)}$ (badness(w[i:j]) + DP[j+1])
  - DP[n+1]=0
- Return: DP[1]

Vertex Cover Runtime: O(n)

- Problem: Find a minimum set of vertices that contains at least one endpoint of each edge (like placing guards in a house to guard all corridors)
- Base case: cost(leaf, YES) = 1, cost(leaf, NO) = 0
- Recurrence:
  - Let cost(v,b) be the min-cost solution of the sub-tree rooted at v, assuming v's status is b ∈ {YES,NO}, where YES corresponds to "v included in the vertex cover" and NO to "not included in the vertex cover" (a term per child of v)
  - cost(v, YES) = 1+$\min_{b1}$ cost(u1,b1)+ $\min_{b2}$cost(u2,b2)+…
  - cost(v, NO) = cost(u1,YES)+cost(u2,YES)+….
- Return: min{cost(root, YES), cost(root, NO)

Parsimony (aka evolutionary tree)
Runtime: O(n k)×O(k) = O(nk^2), k = alphabet size

- Problem: Given: n "leaf strings" of length m each, with letters from {A, C, G, T} and a tree. Goal: find **"inner node" sequences** that minimize the sum of all mutations along all edges



- Recurrence:
  - For any node v of the tree and label L in {A, C, G, T}, define cost(v,L). This is the minimum cost for the subtree rooted at v, if v is labeled L.
  - cost(v,L)=$\min_{L1, L2}$ (D(L,L_1)+D(L,L_2)+cost(u_1,L_1)+ cost(u_2,L_2))
- Return: $\min_l$ cost(root,L)

Hash Tables

insert(k): O(1) ↑ first search **search(k):** O(1) **remove(k):** O(1)

Hash Functions h(k): maps the universe U of keys to some index 0 to m.

Good Hash Function:

- satisfies **simple uniform hashing assumption (SUHA):** each key is equally likely to hash to any of the m slots.
- does not hash similar keys to the same slot
- is quick to calculate, should have O(1) runtime
- is deterministic. h(k) should always return the same value for a given k

Divisor: $h(k) = k \bmod m$ (m is NOT a power of 2)

Multiplication: $h(k) = \lfloor m(kA \bmod 1) \rfloor$

(0<A<1, non-rational, $\frac{\sqrt{5}-1}{2}$) Works with any size m.

COLLISION HANDING 1: Chaining: if a key collides, it is inserted into the same linked list in the hashed slot

search: time to hash + time to go through the chain = T(h(k)) + T(α) = **O(1+α) = O(1),** assuming m= Ω(n), α = n/m

COLLISION HANDING 2: Open Addressing: Hash function maps keys "randomly" to permutations of {0, 1, …, m - 1}. During insertion, systematically probe table in order specified by permutation until empty slot is found. (Use same function for search and deletion - although with deletion, should use "deleted" label.)

1) Linear probing: resolves collisions by simply checking the next slot, i.e. if a collision occurred in slot j, the next slot to check would be slot j + 1:

$h(k,i) = (h'(k) + i) \bmod m$

Drawback: if keys hash to slots close to each other, a cluster of adjacent slots gets filled up. When trying to insert future keys into this cluster, we must traverse through the entire cluster in order to find an empty slot to insert into.

2) Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

Drawback: Introduces more spacing between the slots we try in case of a collision, which reduces the clustering effect seen in linear probing. However, a milder form of clustering can still occur, since keys that hash to the same initial value will probe the exact same sequence of slots to find an empty slot.

3) Double Hashing: resolves collisions by using another hash function to determine which slot to try next

$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$ (choose prime m so $gcd(m, h_2(h)) = 1$)

Both the initial probe slot and the method to try other slots depend on the key k, which further reduces the clustering effect.

4) Cuckoo Hashing: uses two hash functions. If A[h_1(k)] is empty, store key there, if A[h_2(k)] is empty, store there. Otherwise, push key into A[h_1(k)] and insert the key that was there into A[h_2(k)] using the same procedure, only with this location and that kicked-out key's alternative location. Insertion takes **O(1)** on average. If it loops, rehash table using new hash functions.

Searching for a key in a hash table using open addressing involves probing through slots until we find the key we want to find or NIL. If we encounter a slot with a NIL value before finding the key itself, it means that the key in question is not in the hash table.

Deleting for a key involves searching for the key first. Once the key is found, we remove it by replacing the key in that slot with a dummy DELETED value to indicate that a key was once present in this slot, but not anymore. Note that we cannot replace the key with a NIL value, or else searching for keys further down in the probe sequence will falsely return NIL.

Performance: all operations (**search, insert, delete**) require a probe sequence (Linear, Quadratic, Double Hashing). We make the **uniform hashing assumption (UHA),** which assumes that the probe sequence is a random permutation of the slots 0 to m-1. Using principles of probability, if there is exactly a probability of p that we will find an empty slot at each probe, then we expect to probe 1/p times before we succeed. Since p = 1 − α, we expect to probe at most **1/ (1 − α)** times. The performance is fairly good until α approaches too close to 1. **Runtime: Unsuccessful Search:** $\frac{1}{1-\alpha}$, **Successful Search:** $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

iversal Hashing: *O(1+ α) = O(1).* With a fixed hashing function, an adversary could select a series of keys to insert into the hash table that all collide, giving the n table worst-case performance. Universal hashing is the idea that we select the hash function randomly from a group of hash functions. The group of hash functions H must be universal. For each pair of keys *k, l*, the number of hash functions in the group for which *h(k) = h(l)* is at most *|H| / m.* -> the chances m of picking a hash function in which they collide is at most *1/m.*

Newton's Method – Find Roots of Functions

The approximate the zero of a function $f(x)$, take the following approximation with initial guess x_0.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

This has quadratic convergence, so the number of correct digits doubles after each iteration.

Karatsuba's Method – High Precision Multiplication

To multiply two large numbers x and y, split them into $x = x_1 B^m + x_o$, and $y = y_1 B^m + y_o$.

Then $xy = z_2 B^{2m} + z_1 B^m + z_o$,

($z_2 = x_1 y_1, z_0 = x_0 y_0$, and $z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$) So we only need 3 multiplications instead of 4.

If $t(n)$ denotes the total number of elementary operations to multiply two n-digit numbers, then

$$t(n) = 3t\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + cn + d \rightarrow t(n) = \Theta(n^{\frac{\log(3)}{\log(2)}})$$

Maths:

- logs:
  - $a^{\log_a c} = c$
  - $\log_b a = 1/\log_a b$
- Summation:
  - $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
  - $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
  - $\sum_{i=0}^{n} i^3 = \frac{n^3(n+1)^2}{4}$
  - $\sum_{n=0}^{n} x^n = \frac{x^{n+1} - 1}{x - 1}$
  - if $|x| < 1$, $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$
  - $\sum_{n=1}^{n} \frac{1}{n} = \ln(n) + O(1)$

Karp-Rabin: O(n)

- Uses if string is big, s = string
- Hash s and window of t, compare if equal, compare real value
- If not, no need to recompute the whole window, subtract value of first char, add value of next, then hash

$P(h(K_i) = h(K_j)) = \frac{1}{m}$

$E[n_{bucket}] = \frac{n}{m} = \alpha$