1 Java

ArrayList
A resizable array.

LinkedList
A list of elements where ea. element points to the next.

ArrayDeque
A resizable array that acts like a double ended queue, which means that you can enqueue and dequeue elements from both ends of the queue. This can be used as either a stack or a normal queue.

TreeSet
An ordered set of unique values.

TreeMap
Ordered keys mapped to values.

HashSet
Unordered keys mapped to values.

HashMap
An unordered set of unique values.

Priority Queue
min or max, not both.

```
// Enqueue/Dequeue: O(log(n))
pQueue.remove();
pQueue.add();
pQueue.poll();

// Retrieval: O(1)
pQueue.peek();
pQueue.element();
pQueue.size();
```

Custom Sorting

```
import java.util.Arrays;
import java.util.Comparator;

Arrays.sort(ToSort,
    new Comparator<ToSortClass>() {
        @Override
        public int compare(
            ToSortClass o1, ToSortClass o2
        ) {
            return Integer.compare(o1.value,
                                   o2.value);
        }
    }
);
```

2 Backtracking

Recursion, try ea. possibility and go next, if no work go back

```
// ex: Sudoku
for (int i=1;i<=9;i++) { // place 1-9 for sudoku
  if (check(r,c,i)) { // if can place value i
    grid[r][c]=i; // place it
    if (solve(r,c+1)) { // solve w/ board next pos
      return true; // it's solved
    }
    // backtrack, rm val for later calls
    grid[r][c]=0;
  }
}
```

3 Data Structures

Disjoint Sets
A set of sets. Each set has a marked "leader" element. Two sets A and B are **disjoint** if $A \cap B = \emptyset$

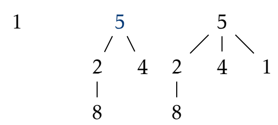
Array Representation:

Value in any given index corresponds to its direct parent. Value will be the same as the index if it is the "root" of a tree set or is just an individual value.
Find: Returns the marked "leader" element of a set.

Union Operator:

(prioritize smaller tree height)
Merges two disjoint sets together. If visualizing as a tree set, take the tree with the smaller height, and merge it into the taller tree.

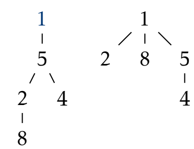
Before After



Path Compression:

(findset on 8)

Before After



1	5	7	5	1	7	7	2
1	2	3	4	5	6	7	8

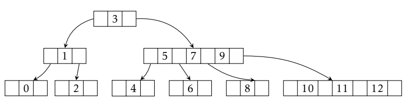
First, you find the root of this tree which is 1. Then you go through the path again, starting at 8, changing the parent of each of the nodes on that path to 1.

1	5	7	5	1	7	7	1
1	2	3	4	5	6	7	8

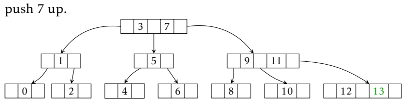
Then, you take the 2 that was previously stored in index 8, and then change the value in that index to 1.

2-4 Trees
num of children is equal to entries + 1 || 0

Insertion:



Insert 13: 4 node. Split it. Push 11 up to parent. Parent is a 4 node, push 7 up.



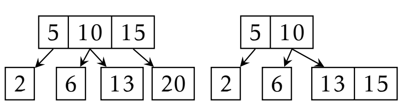
can push up 2nd or 3rd value (3rd is more common)

Deletion:

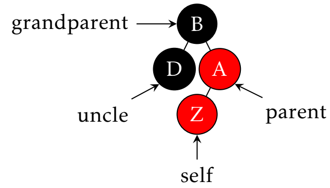
1. Find key to remove and replace w/ next higher key.
2. If sibling > 1 key, steal an adjacent key, make taht the parent and bring down the current parent.
3. If no adjacent sibling has greater than one key, steal a key from a parent.
4. If parent is the root and contains only one key and sibling has only one key, fuse it into a key node and make it the new root.

Delete 20 from the following 2-4 tree.

Before After



Binary Tree Relationships



Red-Black Trees

1. A node is either red or black.
2. The root and leaves (NIL) are black.
3. If a node is red, then its children are black.
4. All paths from a node to its NIL descendants contain the same number of black nodes.
5. The longest path (root to farthest NIL) is no more than twice the length of the shortest path (root to nearest NIL).
   - Shortest path: all black nodes
   - Longest path: alternating red and black

Insertion:

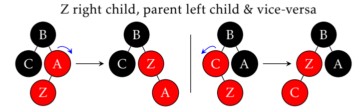
Strategy:

1. Insert Z and color it red
2. Recolor and rotate nodes to fix violation

Z is illegal scenarios

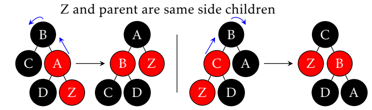
0. Z = root → color black
1. Z.uncle = red → recolor
2. Z.uncle = **black**(triangle) → rotate Z.parent

Z right child, parent left child & vice-versa



3. Z.uncle = **black**(line) → rotate Z.grandparent & recolor

Z and parent are same side children



Deletion:

DB node has...

1. **black sibling with at least one red child.** This fixes the problem structurally. No extra work is required after this case completes. This corresponds to a transfer operation in a 2-4 Tree.
2. **black sibling with two black children.** This uses recoloring and no structural change. It may solve the problem, but may ALSO propagate the DB node to the parent of the current DB node. This corresponds to a fusion and drop operation in a 2-4 Tree.
3. **red sibling.** A structural change here puts you in case 1 or case 2. At this point, a single application of either case is sufficient. This corresponds to a fusion where you have enough values in the parent node to drop one into the fused child.

Skip Lists
Stacks of linked lists.

Insertion:

Insert at bottom level. 50% chance it's added up, again & again. Stop if it is only one at top level.

Deletion:

Delete value and update all pointers.

4 Algorithm Analysis

$\lg n = \log_2 n$

Order Notation

Big-Oh (O):

Big-Oh is an upper bound. It simply guarantees that a function is no larger than a constant times a function $g(n)$, for $O(g(n))$.
$f(n) = O(g(n))$ iff $\forall n \geq n_0$ (const n_0)
$f(n) \leq cg(n)$ for some const c

Big-Omega (Ω):

$f(n) = \Omega(g(n))$ iff $\forall n \geq n_0$ (const n_0)
$f(n) \geq cg(n)$ for some const c

Big-Theta (Θ):

$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
$f(n) = \Theta(g(n))$
iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, where const c and $c > 0$

Master Theorem

$$T(n) = \overbrace{\ A\ }^{\text{\# of sub probs in recursion}}\ T\underbrace{\left(\frac{n}{B}\right)}_{\text{size of ea. sub problem}} + \overbrace{O(n^k)}^{\text{cost of work done out of recursive call}}$$

$$T(n) = \begin{cases} B^k < A \Rightarrow O(n^{\log B^A}) \\ B^k = A \Rightarrow O(n^k \lg n) \\ B^k > A \Rightarrow O(n^k) \end{cases}$$

Expectation Definition
$E(x) = \sum_{x \in X} x \cdot p(x)$

Binomial Theorem
$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$

$$\overbrace{\binom{n}{k}}^{\text{\# of ords of } k \text{ suc } n-k \text{ fail}}\ \underbrace{p^k}_{\text{prob of k suc in their slots}}\ \overbrace{(1-p)^{n-k}}^{\text{same for fails}}$$

Binary Search Average Case Run Time
$O(\log n)$

Make Heap Worst Case Run Time
$O(n \log n)$

Quick Select Average Case Run Time
$O(n)$

5 Sorting

Lower Bounds

Comparison Sorts:

Given an input of n numbers to sort, they can be arranged in $n!$ different orders. With k cols, each with 2 possible answers, there are 2^k possible distinct rows. $\Omega(n \log n)$

Input	$a[1] > a[2]$	$a[1] > a[3]$	$a[2] > a[3], \ldots$	$a[n-2] > a[n-1]$
2,1,4,3	T	F	F	T
2,3,1,4	F	T	T	F
4,3,2,1	T	T	T	T

Adjacent Element Swap Sorts:

An inversion in a list of numbers is a pair of numbers that are out of order relative to each other.
The average number of inversions in a random list of distinct numbers is:
$\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{2} = \Omega(n^2)$
The average case run-time of all of these algorithms is $\Omega(n^2)$.

Bucket Sort
Inputs randomly distributed in range $[x, N)$, for n amount of values, create different buckets to hold the values. $\frac{N}{n}$ will give the new ranges. $O(n)$
Consider sorting a list of 10 numbers known to be in between 0 and 2, not including 2 itself. Thus, each bucket will store values in a range of $\frac{2}{10} = .2$ In particular, we have the following list:

Bucket	Range of Values
0	$[0, .2)$
1	$[.2, .4)$
2	$[.4, .6)$
3	$[.6, .8)$
4	$[.8, 1)$
5	$[1, 1.2)$
6	$[1.2, 1.4)$
7	$[1.4, 1.6)$
8	$[1.6, 1.8)$
9	$[1.8, 2)$

Counting Sort
In counting sort, each of the values being sorted are in the range from 0 to m, inclusive. Here is the algorithm for sorting an array a[0],...,a[n-1]:

1. Create an aux c, indexed from $c[0]$ to $c[m]$ and init each value in the array to 0.
2. Run through the input array a, tabulating the number of occurrences of each value 0 through m by adding 1 to the value stored in the appropriate index in c. (Thus, c is a freq array.)
3. Run through the array c, a 2^{nd} time so that the value stored in each array slot represents the number of elements ≤ the index value in the original array a.
4. Now, run through the original input array a, and for each value in a, use the aux array c to tell you the proper placement of that value in the sorted input, which will be stored in a new array $b[0]..b[n-1]$.
5. Copy the sorted array from b to a.

Radix Sort
input: non-neg ints, k digits long, $O(nk)$.

1. Sort the values using a $O(n)$ stable sort on the kth most sig. digit.
2. Decrement k by 1
3. Repeat step 1. (Unless $k = 0$, then you're done.)

unsorted	v_1	v_2	v_3
235	162	628	162
162	734	734	175
734	674	235	235
175	235	237	237
237	175	162	628
674	237	674	674
628	628	175	734

v_1: sorted by units digit
v_2: sorted by tens digit
v_3: sorted by hundreds digit

6 Greedy Algorithms

Fractional Knapsack

Goal is to maximize the value of a knapsack that can hold at most W units worth of goods from a list of items $I_1, I_2, \dots I_n$.
Each item has 2 attrs:

1. A value/weight; let this be v_i for item I_i.
2. Weight available; let this be w_i for item I_i.

The algorithm is as follows:

1. Sort the items by value/unit.
2. Take as much as you can of the most expensive item left, moving down the sorted list. You may end up taking a fractional portion of the "last" item you take.

Single Room Scheduling

Given a single room to schedule, and a list of requests, the goal of this problem is to maximize the total number of events scheduled. Each request simply consists of the group, a start time and an end time during the day.
Here's the greedy solution:

1. Sort the requests by finish time.
2. Go through the requests in order of finish time, scheduling them in the room if the room is unoccupied at its start time.

Multiple Room Scheduling

Given a set of requests with start and end times, the goal here is to schedule all events using the minimal number of rooms. Once again, a greedy algorithm will suffice:

1. Sort all the requests by start time.
2. Schedule each event in any available empty room. If no room is available, schedule the event in a new room.

Change

The goal here is to give change with the minimal number of coins possible for a certain number of cents using 1 cent, 5 cent, 10 cent, and 25 cent coins.
The greedy algorithm is to keep on giving as many coins of the largest denomination until you the value that remains to be given is less than the value of that denomination. Then you continue to the lower denomination and repeat until you've given out the correct change.

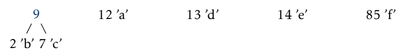
Huffman Coding

For the following character frequencies:

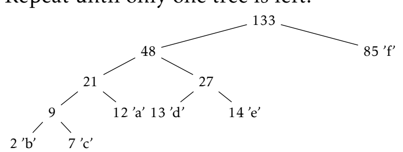
Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

Create a binary tree for each char that also stores the frequency w/ which it occurs.
The algorithm is as follows:

1. Find the two bin trees in the list that store min freqs at their nodes.
2. Connect these two nodes at a newly created common node that will store NO character but will store the sum of the freqs of all nodes connected below it.



Repeat until only one tree is left:



One the tree is built, each leaf node corresponds to a letter w/ a code. To determine the code for a node, walk a std search path from the root to the leaf node.
For every step to the left, append a 0 to the code and for every step right, append a 1.
For the ex. tree we get the codes:

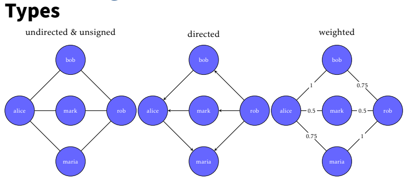
Character	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

Calculating Bits Saved:

total bits $= \sum \text{char}_{\text{freq}} \cdot \text{char}_{\#\text{ bits}} = (12 \cdot 3)_a + \cdots + (85 \cdot 1)_f = 238$
Assuming the original file is storing each of the 6 chars with a 3-bit code. Since there are 133 such characters, the total num of bits used before huffman is $3 \cdot 133 = 399$.
$\therefore$ we saved $399 - 238 = 161$ bits.

7 Unweighted Graphs

Types



Depth First Search

Search down a path from a vertex as far as you can go. Then backtrack to the last vertex from which a different path could have been taken. $O(V+E)$

Breadth First Search

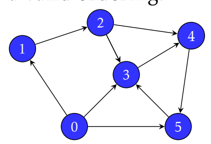
Search all the paths at a uniform depth from the source before moving into deeper paths.

Topological Sort

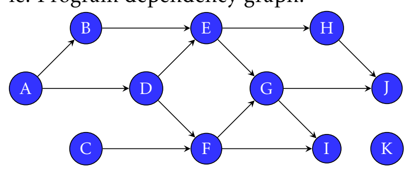
Can find a top. ord. in $O(V+E)$ time.

Topological Ordering:

An ordering of the nodes in a directed graph where for each directed edge from node A to node B, node A appears before node B in the ordering.
Top. ords. are NOT unique.
A graph which contains a cycle cannot have a valid ordering:



The only type of graph which has a valid top. ordering is a **Directed Acyclic Graph (DAG)**. These are graphs with directed edges and no cycles.
ie: Program dependency graph.



Topological Algorithm:

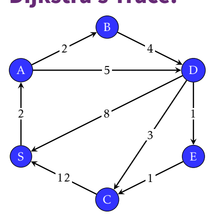
1. Pick an unvisited node
2. Beginning w/ the selected node, do a DFS exploring only unvisited nodes.
3. On the recursive callback of the DFS, add the current node to the top. order- ing in rev. order.

8 Weighted Graphs

Dijkstra's

Finds the shortest path from a src vertex to all other vertices in a weighted directed graph w/out negative edge weights. (uses BFS)

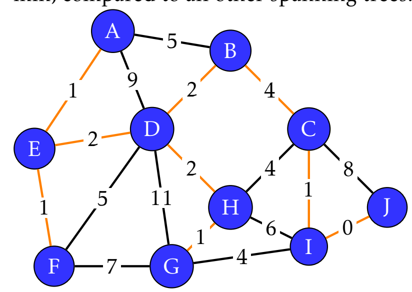
Dijkstra's Trace:



	Distances					
processed	S	A	B	C	D	E
	0	∞	∞	∞	∞	∞
S (0)	0	2	∞	12	8	∞
A (2)	0	2	4	12	7	∞
B (4)	0	2	4	12	7	∞
D (7)	0	2	4	10	7	8
E (8)	0	2	4	9	7	8
C (9)	0	2	4	9	7	8

Minimum Spanning Trees

tree: A connected graph w/out cycles.
Spanning Tree: A subtree of a graph that includes each vertex of the graph. A subtree of a given graph as a subset of the components of that given graph.
Minimum Spanning Tree: Only def for weighted graphs. This is the spanning tree of a given graph whose $\sum$edge weights is min, compared to all other spanning trees.



(orange) Minimum spanning tree with weight 14

Kruskal's

1. Sort edges by ascending edge weight.
2. Walk through the sorted edges and look at the two nodes the edge belongs to, if the nodes are already unified we don't include this edge, otherwise we include it and unify the nodes.
3. The algorithm terminates when every edge has been processe or all the vertices have been unified.

Prim's

A greedy MST algorithm that works well on dense graphs. However, when finding the minimum spanning forest on a disconnected graph, Prim's must be run on each connected component individually.
The lazy version of Prim's has a runtime of $O(E \log E)$, and the eager version has a better runtime of $O(E \log V)$.

Lazy Prim's:

1. Maintain a min Priority Queue that sorts edged based on min edge cost.
2. Start the algorithm on any node **s**. Mark **s** as visited and iterate over all edges of **s**, adding them to the PQ.
3. While the PQ is not empty and a MST has not been formed, dequeue the next cheapest edge from the PQ. If the dequeued edge has already been visited, skip it and poll again. Otherwise, marked the current node as visited and add the selected edge to the MST.
4. Iterate over the new current node's edges and all all its edges to the PQ. Do not add edges to the PQ which point to already visited nodes.

```
// edge implements Comparable
public ArrayList<edge> prims(
  ArrayList<edge>[] graph
) {
  int n = graph.length;
  // shortened to PQ for spacing
  PQ<edge> pq = new PriorityQueue<edge>();
  boolean[] used = new boolean[n];
  used[0] = true;
  for (edge e: graph[0])
    pq.offer(e);

  ArrayList<edge> mst = new ArrayList<edge>();
  while (pq.size() > 0 && mst.size() < n-1) {
    edge cur = pq.poll();
    if (used[cur.u] && used[cur.v]) continue;
    int newV = !used[cur.u] ? cur.u : cur.v;
    mst.add(cur);
    used[newV] = true;
    for (edge e: graph[newV])
      pq.offer(e);
  }
  if (mst.size() < n-1) return null;
  return mst;
}
```

Network Flow

Max-Flow, Min-Cut Theorem: The value of the maximal flow in a flow network equals the value of the minimum cut.

Ford Fulkerson:

While there exists an augmenting path: Add the appropriate flow to that augmenting path.
Typically, DFS is used to check for the existence of an augmenting path.
worse-case, the algorithm takes $O(|f|E)$ time, where $|f|$ is the maximal flow of the network.

Edmonds Karp Algorithm:

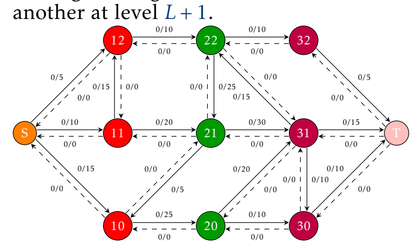
A variation on the Ford-Fulkerson method. Idea is to stry to choose good augmenting paths. In this algorithm, the augmenting path suggested is the augmenting path with the mininal number of edges. (Can be found using BFS) The total number of iters is $O(VE)$. Thus, total run time with the graph stored as an adj. matrix is $O(V^3E)$.

Dinic's Algorithm:

A strongly polynomial max flow algorithm with a runtime of $O(V^2E)$.
The strongly polynomial means that the runtime does not depend on the capacity values of the flow graph.
Extremely fast and works even better on bipartite graphs, with a time complexity of $O(\sqrt{V}E)$ due to the algorithm's reduction to Hopcroft-Karp.
The main idea is to guide augmenting paths from $s \to t$ using a level graph.

Level Graph:

An edge is only part of the level graph if it makes progress towards the sink. That is, the edge must go from a node at Level L to another at level $L+1$.



Algorithm Steps:

1. Construct a level graph by doing a BFS from the src to label all the levels of the current flow graph.
2. If the sink was never reached while building the level graph, then stop and return the max flow.
3. Using only valid edges in the level graph, do multiple DFSs from $s \to t$ until a blocking flow is reached, and sum over the bottleneck values of all the augmenting paths found to calculate the max flow.
4. Repeat Steps $1 \to 3$

9 Divide and Conquer

Integer Multiplication

Imagine multiplying an n-bit number by another n-bit number, where n is a perfect power of 2. (This will make the analysis easier.) We can split up each of these numbers into two halves.

$I \times J = [(I_h \times 2^{\frac{n}{2}} + I_l)] \times [(J_h \times 2^{\frac{n}{2}} + J_l)] = I_h \times J_h \times 2^n + (I_l \times J_h + I_h \times J_l) \times 2^{\frac{n}{2}} + I_l \times J_l$

This way, we have broken down the problem of 2 n-bit nums into 4 mults of n/2-bit nums plus 3 addtions. Thus the run-time $T(n) = 4T(n/2) + \theta(n)$

This has the solution of $T(n) = \theta(n^2)$ by the Master Theorem.

To optimize this:

$P_1 = (I_h + I_l) \times (J_h + J_l) = I_h \times J_h + I_x \times J_l + I_l \times J_h + I_l \times J_l$

$P_2 = I_h \times J_h$

$P_3 = I_l \times J_l$

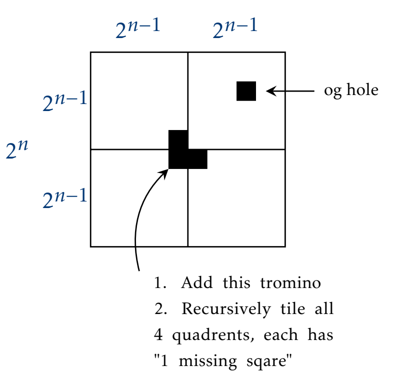
$P_1 - P_2 - P_3 = I_h \times J_1 + I_l \times J_h$

$I \times J = P_2 \times 2^n + [P_1 - P_2 - P_3] \times 2^{\frac{n}{2}} + P_3$

Tromino "Tiling"

A tromino is a figure composed of three 1x1 squares in the shape of an L. Given a $2^n \times 2^n$ checkerboard with 1 missing square, we can recursively tile that square with trominoes.

1. Split the board into four equal sized squares.
2. The missing square is in one of these four squares. Recursively tile this square since it is a proper recursive case.
3. Although the three other squares aren't missing squares, we can "create" these recursive cases by tiling one tronimo in the center of the board, where appropriate:



Skyline Problem

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Closest Pair of Points

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Strassen's Algorithm

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

10 Dynamic Programming

Fibonacci

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Combinations

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Longest Common Subsequence (LCS DP)

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Number of Ways to Make Change

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Fewest Number of Coins to Make Change

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

0-1 Knapsack Problem

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Floyd-Warshall's Algorithm and path reconstruction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Matrix Chain Multiplication

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Edit Distance

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Road Optimization Problem Idea

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

11 Probabilistic Algorithms

Fermat's Theorem

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Miller-Rabin Primality Test

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Rolling Hash Function and String Matching

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.