Hack Pack

Matthew Garrison

2017-10-31

Contents

Approaches

Brute force

Greedy

Dynamic programming

Feasibility

1 second: $\leq$ 100 million operations; 5 seconds: $\leq$ billion; 30 seconds: $\leq$ 10 billion.

Approximate size-to-runtime relationships:

- n = 1,000,000: *O(n log(n))*
- n = 10,000: $O(n^2)$

Secondly, you will get stack overflow if your stack is too large. It varies competition to competition, but you should aim for less than 5000 method calls.

Thirdly, your memory (eg. size of your array) should not exceed 50 million.

Permutations: You should only do permutations if n $<$ 13.

Graphs

Dense graph: E $\approx$ V^2
Sparse graph: E $\approx$ V

General matching works on sparse graphs, V $\leq$ 500.
Bipartite matching works on sparse graphs, V $\leq$ 5000.
Floyd-Warshal works on sparse and dense graphs, V $\leq$ 500.
Dijkstras works on sparse and dense graphs, V $\leq$ 5000.
Toposort works on sparse graphs, V $\leq$ 100,000.

Dynamic programming

You can find the max amount of memory youll use by multiplying the bounds of each component of your state. (eg. For the knapsack problem, this is number of items * the weight limit). If it exceeds 50 million, you can try to save space by removing one item from your state. (eg. You can iterate over the same 1D array in the knapsack problem, instead of using an array that has a row for every item.)

2^n memory

Bitmask DP uses 2^n memory. For these and other, similar problems, n $\leq$ 25 (usually). If n is greater than that, try cutting the problem in half, and approaching the "center" of the problem from both ends.

(For bitmask DP, the array is declared with size 2^n, so that 2^{n-1} can fit into the array.)

Ways to speed up a program

Use a FastScanner and a PrintWriter.

Add all output to a StringBuilder and print it all at once, at the end of the program.

Instead of sorting an ArrayList of custom objects, you can sort by a unique attribute (eg. a String), and hold the rest of the data in a HashMap. (Or use a TreeMap.)

Binary and ternary search

Binary search

Runtime: O(log(n))

A binary search will search through an increasing or decreasing function or sorted array for a desired value.

```
1 int lo = 0, hi = arr.length-1, ans = -1;
2 while (lo <= hi) {
3     int mid = lo + (hi - lo)/2;
4     if (arr[mid] == key) { ans = mid; break; }
5     else if (arr[mid] > key) hi = mid - 1;
6     else lo = mid + 1;
7 }
```

If the array contains multiple occurrences of the value, this will find the first one. To find the last occurrence, change line #6 to be lo = mid + 1.

```
1 int lo = 0, hi = arr.length-1, firstOccurrence = -1;
2 while (lo <= hi) {
3     int mid = (lo + hi) >> 1;
4     if (arr[mid] == key) {
5         firstOccurrence = mid;
6         hi = mid - 1;
7     } else if (arr[mid] > key) hi = mid - 1;
8     else lo = mid + 1;
9 }
```

Binary search for the last `true` value. The answer is at `lo-1`. To search for the first `true`, negate the if-statement, and the answer will be at `lo`.

```
1 int lo = 0, hi;
2 while (lo <= hi) {
3     int mid = (lo + hi) >> 1;
4     if (!isValid(mid)) hi = mid - 1;
5     else lo = mid + 1;
6 }
```

You can also do a binary search on floating point numbers. Basically, you go for repetition, instead of checking if lo exceeds hi. This is most often used when binary searching a function. The number of repetitions you need varies depending on the problem, but 100 is often enough. Make sure to stay within the time limit.

```
1 double lo = 0, hi = Math.PI, ans = -1;
2 for (int i = 0; i < 100; i++) {
3     double mid = lo + (hi - lo)/2.0;
4     double val = formula(mid);
5     if (equals(val, key)) { ans = mid; break; }
6     else if (val > key) hi = mid;
7     else lo = mid;
8 }
```

Alternatively (answer at `lo`):

```
1 double lo = 0, hi = Math.PI;
2 for (int i = 0; i < 100; i++) {
```

```
3       double mid = lo + (hi - lo)/2.0;
4       double ans = formula(mid);
5       if (ans > key) hi = mid;
6       else lo = mid;
7   }
```

Ternary search

A ternary search will find the lowest or highest value in a parabolic function. Like when using a binary search on floating points numbers, we go for repetition. Answer is at `low`.

```
1   double low = 0, high = Math.PI;
2   for (int i = 0; i < 100; i++) {
3       double lowThird = ((2 * low) + high) / 3;
4       double highThird = (low + (2 * high)) / 3;
5       if (calc(lowThird) > calc(highThird)) low = lowThird;
6       else high = highThird;
7   }
```

Ternary search on integers:

```
1   while(lo < hi) {
2       int mid = (lo + hi) >> 1;
3       if (calc(mid) > calc(mid+1)) hi = mid;
4       else lo = mid+1;
5   }
```

Bits

NOT: Because Java uses two's complement, `~n` returns `abs(n) - 1`

Left shift: left shift of x by y is equivalent to multiplying x by 2^y

Right shift: right shift of x by y is equivalent to dividing x by 2^y

Modding by a power of two: If $y = 2^n$, then `x % y` is equivalent to `x & (y - 1)`

Check if n is a power of two: `n != 0 && (n & (n - 1)) == 0`

Alternatively: `Integer.bitCount(n) == 1`

Check if x and y have opposite signs: `(x ^ y) < 0`

Determine powers of 2: `Integer.highestOneBit(n)` returns the largest power of 2 less than or equal to n (ie. a number with a single one bit in the same position as the highest one bit in n). For example, `Integer.highestOneBit(10) == 8` and `Integer.highestOneBit(16) == 16`.

Next power of two: `Integer.highestOneBit(n) << 1;`

`Integer.numberOfTrailingZeros(Integer.highestOneBit(n))` will return the position of that one bit. This is also equivalent to $floor(log_2(n))$

Lowest one bit (there's also an `Integer` method): `n & (-n)`

Undoing XOR: if `c = a ^ b`, then `a = c ^ b` and `b = c ^ a`

Toggle bitmasks: If you have a bitmask and you want to toggle the n^{th} bit, `mask ^= 1 << n;`

Perfect square check:

```
1  static boolean isPerfectSquare(long n) {
2      if (0xC840C04048404040L << n >= 0) return false;
3      int q = Long.numberOfTrailingZeros(n);
4      if ((q & 1) != 0) return false;
5      n >>= q;
6      if ((n & (7|Long.MIN_VALUE)) != 1) return n == 0;
7      long t = (long) Math.sqrt(n);
8      return t*t==n;
9  }
```

Formatting strings

All of this is contained in the `Formatter` Javadoc. There are also more flags, more conversion characters, and specific details.

Format specifiers follow this format:
`%[argument index][flags][width][.precision]conversion-character`

Flags

`-` : Padding occurs to the right of the output instead of the left (width must be specified)
`0` : Numeric values are zero-padded (width must be specified)
`,` : Numeric values include commas

Width

The minimum number of characters to be written to the output. Includes commas and decimal points if the output is numeric. (Note: if the width exceeds the length of the argument, and the '0' flag is not used, spaces will be used to pad.)

Precision

For general argument types, the precision is the maximum number of characters to be written to the output. The precision is applied before the width; thus the output will be truncated to precision characters even if the width is greater than the precision. If the precision is not specified, then there is no explicit limit on the number of characters.

For the floating-point conversions 'e', 'E', and 'f' the precision is the number of digits after the decimal separator (the default is 6 if not specified). If the conversion is 'g' or 'G', then the precision is the total number of digits in the resulting magnitude after rounding. If the conversion is 'a' or 'A', then the precision must not be specified.

For character, integral, and date/time argument, the precision is not applicable. If a precision is provided, an exception will be thrown.

Misc

You can use `printf("%.0f", num)` to round to the nearest integer.

A newline is written as `%n` or `\n`. A percent sign is written as `%%` or `\%`.

Pad a String with characters other than space: `String.format("%10s", str).replace(' ', '*');`.

String of n repetitions of a given character or String (currently n asterisks): `String.format("%" + n + "s", "").replace(' ', '*');`.

Pad base conversions: `String.format("%" + len + "s", Integer.toString(n, r)).replace(' ', '0');`

If you want to print an entire decimal, regardless of its length (ie. print 1.2, 4.578, and 0.13456), convert it to a String before printing.

Uses HALF_UP rounding by default, and uses leading zeroes.

Formatting decimals with DecimalFormat

Check the DecimalFormat Javadoc. Uses HALF_EVEN rounding by default.

Miscellaneous

Anonymous classes and lambdas

Anonymous classes are classes that are declared inline. For example, this is a PriorityQueue with a Comparator that was declared inline.

```
PriorityQueue<Integer> pq = new PriorityQueue<Integer>(new Comparator<Integer>() {
    public int compare(Integer one, Integer two) {
        return two - one;
    }
});
```

You can also use lambdas in a similar way.

```
PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> b - a);
```

Converting 2D coordinates to 1D coordinates

If you have `x` and `y` and `cols`, then you can set `newCoord = y * cols + x;`. From that, you can then work backwards: `x = newCoord % cols;` and `y = newCord / cols;`.

This can also be extended to 3 dimensions: `newCoord = z * rows * cols + y * cols + x;`

Finding k^{th} largest number in unsorted array

*Runtime (**expected**): O(n)*

This is zero-indexed (ie. the smallest number is k=0 and the largest is k=n-1).

```
static int quickSelect(int[] arr, int left, int right, int k) {
    if (left == right) return arr[left];
    int pivotIdx = left + (int)(Math.random() % (right - left + 1));
    pivotIdx = partition(arr, left, right, pivotIdx);
    if (k == pivotIdx) return arr[k];
    else if (k < pivotIdx) return quickSelect(arr, left, pivotIdx-1, k);
    else return quickSelect(arr, pivotIdx+1, right, k);
}
static int partition(int[] arr, int left, int right, int pivotIdx) {
    int pivotVal = arr[pivotIdx];
    swap(arr, pivotIdx, right);
    int storeIdx = left;
    for (int i = left; i < right; i++) {
        if (arr[i] < pivotVal) {
            swap(arr, storeIdx++, i);
        }
    }
    swap(arr, right, storeIdx);
    return storeIdx;
}
```

Frame problems

A frame problem is one where you have to find the consecutive section with the largest value or largest length, while remaining within a constraint (usually a maximum value that the section can be). Also known as a two pointer problem.

To solve one of these problems, you need a startIndex and endIndex. Set them all to zero, and use this for-loop: `for (; endIndex < numValues; endIndex++)`

Within the for-loop, you check if the constraint is still true. If it isn't, you increment startIndex until the constraint is true again.

Hashcodes

When writing custom classes, you may need to create a `hashCode()` method for your class. Eclipse can help you do this: while your cursor is inside the class you want to create the method for, press Alt+S, then click "Generate hashCode() and equals()".

Infinity

For a lot of problems, you may wish to define `INFINITY`. A good value for that is `(Integer.MAX_VALUE / 2) - 5`. This value is sufficiently large that any actual values will not be larger, while still being small enough to not overflow if it is added to itself.

Input and output

Input

A Scanner that reads from a file: `new Scanner(new File(String filename));`

A BufferedReader that reads from Standard In: `new BufferedReader(new InputStreamReader(System.in));`

A BufferedReader that reads from a file: `new BufferedReader(new FileReader(String fileName));`

Output

A faster way to print to Standard Out: `new PrintWriter(new BufferedWriter(new OutputStreamWriter(System.out)));`

To a file: `new PrintWriter(new BufferedWriter(new FileWriter(String fileName)));`

Note: you NEED to call either out.flush() or out.close() (which in turn calls out.flush()) when you're done printing. If you don't, the print statements will not work.

Notes

Reads until end of file (Scanner): `while (scan.hasNext()) {}`

Reads until end of file (BufferedReader): `String input; while ((input = br.readLine()) != null) {}`

Inputting lines in clockwise order

If you want to input lines in order, so each one is at a 90-degree angle to the previous, do it like so:

```
long[] dx = {1, 0, -1, 0};
long[] dy = {0, 1, 0, -1};

String[] distances = br.readLine().split(" ");
Line2D.Double[] paths = new Line2D.Double[distances.length];

long currX = 0;
long currY = 0;
for (int j = 0; j < distances.length; j++) {
    long prevX = currX;
    long prevY = currY;
    currX += dx[j%4] * Long.parseLong(distances[j]);
    currY += dy[j%4] * Long.parseLong(distances[j]);
    paths[j] = new Line2D.Double(prevX, prevY, currX, currY);
}
```

Knight's Shortest Path

To find the shortest path a knight can take from (x_1, y_1) to (x_2, y_2) on an infinite chessboard:

```
int dx = Math.abs(x1 - x2), dy = Math.abs(y1 - y2);
if (dx < dy) { int t = dx; dx = dy; dy = t; }
if (dx == 1 && dy == 0) return 3;
if (dx == 2 && dy == 2) return 4;
double delta = dx - dy, divisor = (dy > delta ? 3 : 4);
return (int)(delta - 2 * Math.floor((delta - dy) / divisor));
```

If the chessboard is not infinite, you will need to check if the knight is moving from a corner of the board to the square that is one square away diagonally (the formula produces 2, but the answer is 4).

If `numRows == 4`, then when `dy == 3 && (x == 0 || x == numCols-1)`, the formula produces 3 instead of 5 (this applies if `numCols == 4` or both are 4).

If `numRows == 3`, then when `dx == 2 && y == 1`, the formula produces 2 instead of 4 (this applies if `numCols == 3`, but doesn't if both are 3; you start getting impossible cases).

Max Contiguous Subsequent Sum (MCSS)

This problem asks you to find the maximum sum of some contiguous sequence of numbers, whose length is non-zero. If all the numbers are negative, then the answer is zero.

If you just want the answer, use this version:

```
int max = Integer.MIN_VALUE, sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
    max = Math.max(max, sum);
    if (sum < 0) sum = 0;
}
```

If you also want the start and end indexes of the subsequence, use this version:

```
int max = Integer.MIN_VALUE, sum = 0, start = 0, end = 0, i = 0;
for (int j = 0; j < arr.length; j++) {
    sum += arr[j];
    if (sum > max) {
        max = sum;
        start = i;
        end = j;
    } else if (sum < 0) {
        i = j + 1;
        sum = 0;
    }
}
```

Magic squares

To solve a 3x3 magic square, add up all the known numbers and divide by two. This is your target number. Then, for every blank square, subtract the sum of the non-blank squares in that row/column/diagonal from the target number to get your answer.

Range sums

To quickly compute the sum of the values of an array over a range, you can use a running sum array. For each index, the value in the array at the index will be the sum of all the values before that index plus the value in the original array at that index. Using 1-based indexes for the running sum array is better, so that you dont need a special case for when a = 0.

```
int[] arr = new int[length];
int[] runSum = new int[length+1];
for (int i = 0; i < length; i++) {
    arr[i] = scan.nextInt();
    runSum[i+1] = runSum[i] + arr[i];
}
```

Then, to find the sum of the values on the range `[a, b]`:

```
int ans = runSum[b] - runSum[a-1];
```

If you want to do this for 2D arrays, use this code to create the running sum array:

```
int[][] arr = new int[height][width];
runSum = new int[height+1][width+1];
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        arr[y][x] = scan.nextInt();
        runSum[y+1][x+1] = runSum[y+1][x] + arr[y][x];
    }
}
// Add the columns.
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        runSum[y+1][x+1] += runSum[y][x+1];
    }
}
```

Then, to find the sum of the values in the rectangle, where the top left corner is (lowX, lowY) and the bottom right corner is (highX, highY):

```
int ans = runSum[highY][highX] - runSum[lowY-1][highX] -
          runSum[highY][lowX-1] + runSum[lowY-1][lowX-1];
```

Specific dx and dy arrays

Starting above the current space and going clockwise, including diagonals.

```
static int[] dx = { 0,  1, 1, 1, 0, -1, -1, -1};
static int[] dy = {-1, -1, 0, 1, 1,  1,  0, -1};
```

The movements for a knight in chess, following this clockwise pattern:

```
/*
 |8| |1|
7| | | |2
 | |S| |
6| | | |3
 |5| |4|
*/
static int[] dx = {+1, +2, +2, +1, -1, -2, -2, -1};
static int[] dy = {-2, -1, +1, +2, +2, +1, -1, -2};
```

Solve a problem with time intervals

If youre given a series of overlapping times (ie. start and end times), and have to find something (eg. location of the most overlaps), make a Time class, with an `isStart` boolean, a `time` int, and one or more values for that time interval, depending on the problem. Implement Comparable and sort it according to `time`, with starts coming before ends in the event of a tie.

Fill an ArrayList with the Time objects, sort it, and run through it.

Stack trick

If you're getting stack overflow:

- Implement Runnable
- Move everything from `main()` to `run()`
- In `main()`, the only thing should be `new Thread(null, new ClassName(), "", stackSize).start();`

Tower of Hanoi

The minimum number of moves required to solve a Tower of Hanoi with n disks is $2^n - 1$.

To move n disks from peg A to peg C:

- Move n-1 discs from A to B. This leaves disc n alone on peg A.
- Move disk n from A to C

- Move n-1 discs from B to C so they sit on disc n.

To print out the moves of the 3 disk variant:

```
static void solve(int n, String A, String B, String C) {
    if (n == 1) {
        println(A + " -> " + pole3Name);
    } else {
        solve(n - 1, A, C, B);
        println(A + " -> " + C);
        solve(n - 1, B, A, C);
    }
}
```

Trailing zeroes of a factorial

Finds the number of trailing zeroes of $n!$.

```
long findTrailingZeros(long  n){
    long count = 0;
    for (int i = 5; n/i >= 1; i *= 5) count += n/i;
    return count;
}
```

Useful classes

ArrayDeque: stacks and queues

BitSet: useful for problems with large bitmasks

Formatter: the JavaDoc lists printf syntax

Line2D: see if lines intersect

LocalDate, LocalTime, LocalDateAndTime: time problems

MathContext and RoundingMode

Pattern: the JavaDoc lists regex syntax

Rectangle2D

Change making problem

There are two similar problems called the change making problem. NOTE: for both problems, if the coins aren't guaranteed to be given in sorted order, make sure to sort them.

The first

The first asks how many different ways an amount of change can be made using a set of coins.

Iterative (space-saving) solution, where `coins` is an array containing all the values of the coins, in ascending order.

```
1  int[] memo = new int[numCents+1];
2  memo[0] = 1;
3  for (int c = 0; c < numCoins; c++)
4      for (int v = 0; v <= numCents; v++) {
5          int tempIndex = v - coins[c];
6          if (tempIndex >= 0) memo[v] += memo[tempIndex];
7      }
```

Recursive version:

```
1  static int recur(int cents, int index) {
2      if (index >= coins.length) return 0;
3      if (cents < 0) return 0;
4      if (cents == 0) return 1;
5      if (memo[index][cents] != -1) return memo[index][cents];
6
7      int val = recur(cents - coins[index], index) + recur(cents, index+1);
8
9      return memo[index][cents] = val;
10 }
```

The second

The second asks how to make that amount of change using the fewest coins.

This problem can sometimes be solved using a greedy algorithm. If `coins` is a descending-sorted array of the values of each of the coins, this will work for some combinations of coin values, including U.S. coins. However, for example, if the coins are $\{25, 21, 10, 5, 1\}$ and `numCents` $= 63$, this will not work.

The DP solution that will always work is as follows. Note that for this program, `coins` is in ascending order.

```
1  int[] memo = new int[numCents+1];
2  for (int c = 1; c <= numCoins; c++)
3      for (int v = 0; v <= numCents; v++) {
4          int tempIndex = v - coins[c-1];
5          if (tempIndex >= 0) {
6              int take = 1 + memo[tempIndex];
7              if (take < memo[v] || memo[v] == 0)
8                  memo[v] = take;
9          }
10     }
```

Dijkstra's algorithm

Runtime: O(E + V log(V))

Dijkstras algorithm finds the shortest path from a single index to every other index. This algorithm will fail if any of the edges have a negative weight.

```
1  int sourceIndex = scan.nextInt();
2  vertexes[sourceIndex].distanceFromSource = 0;
3  PriorityQueue<Vertex> pq = new PriorityQueue<Vertex>();
4  pq.add(vertexes[sourceIndex]);
5
6  while (!pq.isEmpty()) {
7      Vertex curr = pq.poll();
8      if (curr.visited) continue;
9
10     // If there's a single known destination index, you can use a break
11     // statement here.
12
13     curr.visited = true;
14     for (int i : curr.neighbors) {
15         if (vertexes[i].visited) continue;
16
17         int newDistance = curr.distanceFromSource + distances[curr.index][i];
18         if (newDistance < vertexes[i].distanceFromSource) {
19             vertexes[i].distanceFromSource = newDistance;
20             pq.add(vertexes[i]);
21             vertexes[i].prev = curr.index;
22         }
23     }
24 }
```

Floyd's Cycle Finding Algorithm

Note: this finds cycles in a sequence of iterated function values, not in graphs.

Runtime: O(mu + lambda)

Let `mu` be the smallest index i where x_i is part of the cycle. Let `lambda` be the length of the cycle. $foo(x_i)$ returns the x_{i+1} element.

Step 1

Both `tortoise` and `hare` start at the first element, x_0. `tortoise` advances by 1 every iteration and `hare` advances by 2. When `tortoise` and `hare` equal each other, a cycle of `k*lambda` length has been found.

```
1 int tortoise = foo(x0);
2 int hare = foo(foo(x0));
3 while (tortoise != hare) {
4     tortoise = foo(tortoise);
5     hare = foo(foo(hare));
6 }
```

Step 2

We reset `hare` to x_0 and advance both pointers by 1 every iteration. When they equal the same value, we've found `mu`.

```
1 int mu = 0;
2 hare = x0;
3 while (tortoise != hare) {
4     tortoise = foo(tortoise);
5     hare = foo(hare);
6     mu++;
7 }
```

Step 3

We then set `hare` to `tortoise` and advance it by 1 each iteration. When `hare` equals `tortoise`, we've found the length of the cycle.

```
1 int lambda = 1;
2 hare = foo(tortoise);
3 while (tortoise != hare) {
4     hare = foo(hare);
5     lambda++;
6 }
```

Floyd-Warshall's algorithm

Runtime: $O(V^3)$

This algorithm finds the shortest path between all indexes of a weighted or unweighted graph. Negative edge weights are allowed, but negative cycles are not (ie. the sum of the weights of the edges of the cycle is negative).

The path array is only used if you need to find the actual path from i to j. Once the algorithm has been run, the value at (i, j) is the index of the last vertex reached before j when travelling on the shortest path from i to j. For example, if the path from i to j is [i -¿ 5 -¿ 3 -¿ 2 -¿ j], the value at (i, j) will be 2. You construct the array with the previous vertex for each edge (ie. i) and use -1 to indicate no edge.

```
1  int[][] path = new int[n][n];
2  for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++) {
4          if (adj[i][j] == INFINITY) path[i][j] = -1;
5          else path[i][j] = i;
6      }
```

The algorithm:

```
1  for (int k = 0; k < n; k++)
2      for (int i = 0; i < n; i++)
3          for (int j = 0; j < n; j++)
4              if (adj[i][j] > adj[i][k] + adj[k][j]){
5                  adj[i][j] = adj[i][k] + adj[k][j];
6                  path[i][j] = path[k][j];
7              }
```

To find the actual path from index i to index j using the path array, you need to work backwards. You can add them to an ArrayList, and then print it out backwards.

```
1  ArrayList<Integer> indexes = new ArrayList<Integer>();
2  indexes.add(end);
3  // Loop through each previous vertex until you get back to start.
4  while (path[start][end] != start) {
5      indexes.add(path[start][end]);
6      end = path[start][end];
7  }
8  indexes.add(start);
```

Graphs

General graph algorithms.

BFS and DFS

DFS-Ordering

The DFS-Ordering of a tree orders the vertexes such that all of a vertex's children are on `[startIdx, endIdx]` (and the vertex itself is at `startIdx`). This version (with initial `idx=-1`) will produce indexes on `[0, numVertexes-1]`. You can start with `idx=0` to produce indexes on `[1, numVertexes]`.

```
1 static int idx = -1;
2 static void dfs_order(Vertex curr) {
3     curr.startIdx = ++idx;
4     for (Vertex v : curr.children) dfs_order(v);
5     curr.endIdx = idx;
6 }
```

Eulerian paths/cycles

Eulerian path: a path that visits every vertex exactly once

Eulerian cycle (or circuit): a Eulerian path that is a cycle (ie. starts and ends at the same vertex)

Semi-Eulerian graph: a graph containing a Eulerian path

Eulerian graph: a graph containing a Eulerian cycle

Undirected:

- Eulerian cycle: iff there are no vertexes of odd degree
- Eulerian path: iff there are zero or two vertexes of odd degree (if two, those are the starting and ending vertexes)

Directed:

- Eulerian cycle: iff every vertex has equal in-degree and out-degree and all those vertexes belong to a single connected component
- Eulerian path: iff at most one vertex has $in - out = 1$, at most one vertex has $out - in = 1$, every other vertex has equal in-degree and out-degree, and all vertexes belong to a single component in the equivalent undirected graph

Fluery's algorithm

Runtime: $O(E^2)$

The algorithm is *O(E)*, however it must run Tarjan's bridge-finding algorithm (which is itself *O(E)* every iteration.

1. Start at a vertex of odd degree, or if there isn't any, an arbitrary vertex
2. Choose a non-bridge edge of the current vertex, or if there is no non-bridge edge, choose the remaining edge
3. Move to the other endpoint of the edge and delete the edge

At the end of the algorithm there are no edges left, and the sequence from which the edges were chosen forms an Eulerian cycle if the graph has no vertices of odd degree, or an Eulerian trail if there are exactly two vertices of odd degree.

Hierholzer's algorithm

Runtime: O(E)

- Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
- As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

By using a data structure such as a doubly linked list to maintain the set of unused edges incident to each vertex, to maintain the list of vertices on the current tour that have unused edges, and to maintain the tour itself, the individual operations of the algorithm (finding unused edges exiting each vertex, finding a new starting vertex for a tour, and connecting two tours that share a vertex) may be performed in constant time each

Hamiltonian paths/cycles

Hamiltonian path: a path that visits every vertex exactly once

Hamiltonian cycle: a Hamiltonian path that is a cycle (ie. starts and ends at the same vertex)

Hamiltonian graph: a graph containing a Hamiltonian cycle
Both complete graphs (with more than 2 vertexes) and cycle graphs are Hamiltonian graphs.
All Hamiltonian graphs are biconnected, but not all biconnected graphs are Hamiltonian.

Determining if a graph contains a Hamiltonian path or cycle is NP-complete.

Misc terminology

Articulation point (AKA cut vertex): removing this vertex disconnects the graph

Biconnected: removing any single vertex does not disconnect the graph (ie. it doesn't contain any articulation points)

Bridge: removing this edge disconnects the graph

Cycle graph (C_n): the graph with n vertexes containing a single cycle (AKA just a circle). C_0, C_1, and C_2 are not defined.

Complete graph (K_n): the graph with n vertexes and every possible edge (not a multigraph). K_n has $\binom{n}{2}$ or $\frac{n(n-1)}{2}$ edges.

Empty graph ($\overline{K_n}$): the graph with n vertexes and no edges

k-connected: removing any k vertexes does not disconnect the graph

Toposort

Khan's algorithm:

- Take all the vertexes with indegree 0 and put them into a queue.

- Remove a vertex and subtract the indegree of all its neighbors by 1. If any of those neighbors now have indegree 0, add them to the queue.
- Repeat until the queue is empty.

You can use a PriorityQueue (sorting on index) to get the lexographically first toposort.

Checking if a TopoSort is valid

```
ArrayList<Vertex> inputTopoSort;
for (Vertex v : inputTopoSort) {
    v.visited = true;
    for (Vertex next : v.children) {
        if (next.visited) // INVALID TOPO SORT
    }
}
```

Trees

Diameter: longest path in the tree. Procedure to find it: run a BFS starting at an arbitrary node. Then, run a second BFS starting at the last node you visited in the first BFS. The largest distance found in the second BFS is the diameter. This finds the diameter in terms of edges - to find it in terms of nodes, add 1.

Center: the node that minimizes remoteness from all other nodes (ie. it minimizes the maximum distance to any node). If the diameter is odd (when edge-counting; if node-counting, when it's even), there will be two centers. The center will be the node(s) with distance `diameter/2` after the second BFS.

Centroid: if you remove this node, the maximum size components remaining are minimized

Two-coloring

Runtime: O(n)

Works in Bipartite graphs.

```
outerLoop : for (int i = 0; i < numVertexes; i++) {
    if (vertexes[i].color != NONE) continue;

    ArrayDeque<Vertex> queue = new ArrayDeque<>();
    queue.add(vertexes[i]);
    vertexes[i].color = RED;
    while (!queue.isEmpty()) {
        Vertex curr = queue.poll();
        int otherColor = otherColor(curr.color);

        for (Vertex neighbor : curr.neighbors) {
            if (curr.color == neighbor.color) {
                // We're trying to color two neighbors the same color.
                isPossible = false;
                break outerLoop;
            }

            if (neighbor.color == NONE) {
                // If it's uncolored, color it and add it to the queue.
                neighbor.color = otherColor;
```

```
21                      queue.add(neighbor);
22                  }
23              }
24          }
25      }
```

Held-Karp algorithm

Also called the Bellman-Held-Karp algorithm. Uses dynamic programming to solve the Travelling Salesman problem.

Call `TSP(0, 1)`. `memo` has size `[numCities][1 << numCities]`.

```
static int numCities;
static int[][] dist, memo;
static int TSP(int pos, int bitmask) {
    The round trip has been completed.
    if (bitmask == (1 << numCities) - 1)
        return dist[pos][0];

    if (memo[pos][bitmask] != -1) return memo[pos][bitmask];

    int answer = Integer.MAX_VALUE;
    for (int i = 0; i <= numCities; i++) {
        int pow = (1 << i);
        if ((bitmask & pow) == 0) {
            answer = Math.min(answer, dist[pos][i] + TSP(i, bitmask | pow));
        }
    }

    memo[pos][bitmask] = answer;
    return answer;
}
```

If you want to reconstruct the path, add another array, path, of the same size as memo. At the end of the method, add `path[pos][bitmask] = best`, where best is the city that results in the lowest answer.

```
int idx = 0, mask = 1;
while (mask != (1 << numCities) - 1) {
    System.out.println(idx);
    idx = path[idx][mask];
    mask |= 1 << idx;
}
```

Longest common subsequence/substring

Longest common subsequence

This problem finds the longest common subsequence between two strings. To find just the length (stored in `memo[a.length()][b.length()]`, where `a` and `b` are `char[]`):

```
int[][] memo = new int[a.length+1][b.length+1];
for (int i = 1; i <= a.length; i++) {
    for (int j = 1; j <= b.length; j++) {
        if (a[i-1] == b[j-1]) memo[i][j] = memo[i-1][j-1] + 1;
        else memo[i][j] = Math.max(memo[i][j-1], memo[i-1][j]);
    }
}
```

Space-saving version, if all you need is the length (answer in `memo[1][b.length]` and `b` contains the shorter string):

```
int[][] memo = new int[2][b.length+1];
for (int i = 1; i <= a.length; i++) {
    for (int j = 1; j <= b.length; j++) {
        if (a[i-1] == b[j-1] ) memo[1][j] = memo[0][j-1] + 1;
        else memo[1][j] = Math.max(memo[0][j], memo[1][j-1]);
    }
    // Copies the second row to the first, and then overwrites the second.
    if (i < a.length) {
        memo[0] = memo[1];
        memo[1] = new int[b.length+1];
    }
}
```

To find the LCS itself:

```
int i = a.length, j = b.length;
String LCS = "";
while(i > 0 && j > 0) {
    if (a[i-1] == b[j-1]) {
        LCS = a[i-1] + LCS;
        i--; j--;
    }
    else if (memo[i-1][j] >= memo[i][j-1]) i--;
    else j--;
}
```

If that doesn't work, use this recursive method:

```
static String lcs(int[][] memo, char[] a, char[] b, int i, int j){
    if (i == 0 || j == 0) return "";
    else if (a[i-1] == b[j-1]) return lcs(memo, a, b, i-1, j-1) + a[i-1];
    else {
        if (memo[i][j-1] > memo[i-1][j]) return lcs(memo, a, b, i, j-1);
        else return lcs(memo, a, b, i-1, j);
    }
}
```

If theres more than one possible LCS, use this method:

```
static HashSet<String> lcs(int[][] memo, char[] a, char[] b, int i, int j) {
    HashSet<String> LCS = new HashSet<>();

    if (i == 0 || j == 0) {
        LCS.add("");
    } else if (a[i - 1] == b[j - 1]) {
        for(String s : lcs(memo, a, b, i - 1, j - 1)) {
            LCS.add(s + a[i - 1]);
        }
    } else {
        if (memo[i - 1][j] >= memo[i][j - 1]) {
            LCS.addAll(lcs(memo, a, b, i - 1, j));
        }
        if (memo[i][j - 1] >= memo[i - 1][j]) {
            LCS.addAll(lcs(memo, a, b, i, j - 1));
        }
    }
    return LCS;
}
```

To find the LCS of n Strings, you need an n^{th} dimensional array.

Longest common substring

The answer is in `memo[1][b.length]` and `b` contains the shorter string.

```
int[][] memo = new int[2][b.length+1];
int maxLength = 0;
HashSet<Integer> lcsIndexes = new HashSet<Integer>();

for (int i = 1; i <= a.length; i++) {
    for (int j = 1; j <= b.length; j++) {
        if (a[i-1] == b[j-1]) {
            memo[1][j] = memo[0][j-1] + 1;

            if (memo[1][j] > maxLength) {
                maxLength = memo[1][j];
                lcsIndexes.clear();
                lcsIndexes.add(i);
            } else if (memo[1][j] == maxLength) lcsIndexes.add(i);
        }
    }
    // Copies the second row to the first, and then overwrites the second.
    if (i < a.length) {
        memo[0] = memo[1];
        memo[1] = new int[b.length+1];
    }
}
// Prints all the LCS's.
for (int i : lcsIndexes) {
    System.out.println(String.valueOf(a, i-maxLength, maxLength));
}
```

Longest increasing subsequence/subsection

Longest increasing subsequence

This is the $n\log(n)$ version. `indexOfLastElement[j]` stores the index `k` of the smallest value `arr[k]` such that there is an increasing subsequence of length `j` ending at `arr[k]`. `previous[k]` stores the index of the predecessor of `arr[k]` in the longest increasing subsequence ending at `arr[k]`.

```
int maxLength = 0;
int[] indexOfLastElement = new int[arr.length+1];
int[] previous = new int[arr.length];
for (int i = 0; i < arr.length; i++) {
    // Binary search for the largest positive j <= maxLength such that
    // arr[indexOfLastElement[j]] < arr[i]
    int lo = 1, hi = maxLength;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (arr[indexOfLastElement[mid]] < arr[i]) lo = mid + 1;
        else hi = mid - 1;
    }
    // Now, lo is 1 greater than the length of the longest prefix of arr[i].
    int newMaxLength = lo;
    // The predecessor of arr[i] is the last index of the subsequence of
    // length newMaxLength-1.
    previous[i] = indexOfLastElement[newMaxLength-1];
    indexOfLastElement[newMaxLength] = i;
    if (newMaxLength > maxLength) maxLength = newMaxLength;
}
// Reconstruct the LIS.
int[] LDS = new int[maxLength];
int k = indexOfLastElement[maxLength];
for (int i = maxLength-1; i >= 0; i--) {
    LDS[i] = arr[k]; k = previous[k];
}
```

Longest increasing substring

Finds the LIS, as a frame problem.

```
int maxLength = 1, maxEnd = 1, start = 0, end = 1;
for (; end < arr.length; end++) {
    // The current substring is no longer increasing.
    if (arr[end] <= arr[end-1]) {
        if (end - start > maxLength) {
            maxLength = end - start;
            maxEnd = end - 1;
        }
        start = end;
    }
}
if (end - start > maxLength) {
    maxLength = end - start; maxEnd = end - 1;
}
```

Max flow and Dinic's

Max flow is...

Dinic's

Runtime: $O(V^2E)$

In graphs where every edge has a capacity of 1, the runtime is $O(min\{V^{2/3}, E^{1/2}\}\ E)$

In graphs where "each vertex, except for source and sink, either has a single entering edge of capacity one, or a single outgoing edge of capacity one, and all other capacities are arbitrary integers", the runtime is $O(\sqrt{V}\ E)$. This includes graphs arising from bipartite matching problems.

```
1  static int numVertexes, source, sink;
2  static Vertex[] vertexes;
3  static int[] dist;
4  static boolean[] blocked;
5  static ArrayDeque<Integer> queue;
6  static void createGraph(int n) {
7      // You can remove or change these two lines if you want source and sink to
8      // be different vertexes.
9      source = n++;
10     sink = n++;
11     numVertexes = n;
12     vertexes = new Vertex[n];
13     for (int i = 0; i < n; i++) vertexes[i] = new Vertex(i);
14     dist = new int[n];
15     blocked = new boolean[n];
16     queue = new ArrayDeque<>();
17 }
18 static void addEdge(int v1, int v2, int cap) {
19     Edge e = new Edge(v1, v2, cap);
20     vertexes[v1].edges.add(e);
21     vertexes[v2].edges.add(e.rev);
22 }
23 // Only necessary if you are running Dinic's on the same graph multiple times.
24 static void clear() {
25     for (Vertex v : vertexes)
26         for (Edge e : v.edges) e.flow = 0;
27 }
28 static int dinics() {
29     clear();
30     int flow = 0;
31     while (bfs()) {
32         Arrays.fill(blocked, false);
33         flow += dfs(source, Integer.MAX_VALUE);
34     }
35     return flow;
36 }
37 static boolean bfs() {
38     Arrays.fill(dist, -1);
39     dist[source] = 0;
40     queue.clear();
41     queue.add(source);
```

```
42      while (!queue.isEmpty()) {
43          int curr = queue.poll();
44          for (Edge e : vertexes[curr].edges) {
45              int v = e.to;
46              if (dist[v] == -1 && e.flow < e.cap) {
47                  dist[v] = dist[curr] + 1;
48                  queue.add(v);
49              }
50          }
51      }
52      // Returns whether we made it to the source from the sink.
53      return (dist[sink] > 0);
54  }
55  static int dfs(int curr, int min) {
56      // The min parameter refers to the minimum residual capacity encountered
57      // further up our call stack.
58      if (curr == sink) return min;
59
60      int flow = 0;
61      for (Edge e : vertexes[curr].edges) {
62          int v = e.to;
63          // Check to make sure the path is not blocked and that this is a valid
64          // path according to our level graph.
65          if (!blocked[v] && dist[v] == dist[curr] + 1 && e.cap > e.flow) {
66              // Assign as much flow as we can still fit along this path.
67              int push = dfs(v, Math.min(min - flow, e.cap - e.flow));
68              flow += push;
69              e.flow += push;
70              e.rev.flow -= push;
71          }
72          // We can stop if we can't push any more flow.
73          if (flow == min) return flow;
74      }
75      /*
76      If what we pushed, that is, what was available further down the path, is
77      not equal to what was available up our call stack, that means we filled up
78      a bottleneck along this path, so we can just mark it as blocked. This
79      means speed-ups in the event that we go down this path again if we end up
80      backtracking and trying a different route.
81      */
82      blocked[curr] = (flow != min);
83      return flow;
84  }
85  static void removeEdge(int v1, int v2) {
86      Edge e;
87      for (int i = 0; i < vertexes[v1].edges.size(); i++) {
88          e = vertexes[v1].edges.get(i);
89          if (e.to == v2 && e.cap != 0) vertexes[v1].edges.remove(i--);
90      }
91      for (int i = 0; i < vertexes[v2].edges.size(); i++) {
92          e = vertexes[v2].edges.get(i);
93          if (e.to == v1 && e.cap == 0) vertexes[v2].edges.remove(i--);
94      }
95  }
```

Minimum spanning trees

Kruskal's algorithm takes the shortest Edge that hasn't been used yet and connects the two vertexes if they aren't already indirectly connected. Prim's algorithm only considers the edges connected to vertexes already in the MST.

If the graph is not connected, then the correct algorithm to use depends on what you want: Kruskal's will make a MST for each component and tell you the weight of all of them combined, while both Kruskal's and Prim's can tell you if a single MST doesn't exist, and allow you to go from there.

Kruskal's

Runtime: O(E log(E))

If `edgeCount != numNodes-1` at the end, then the graph is not connected.

```
1 DisjointSet djset = new DisjointSet(numNodes);
2 int edgeCount = 0, mstWeight = 0;
3 Arrays.sort(edges);
4 for (int i = 0; i < edges.size() && edgeCount < numNodes - 1; i++) {
5     if (djset.union(edges[i].vertex1, edges[i].vertex2)) {
6         edgeCount++;
7         mstWeight += edges.get(i).cost;
8     }
9 }
```

Prim's

Runtime: O(E log(E))

(This algorithm uses two `Edge` objects to represent a single undirected edge, one for each vertex in the edge (ie. v_1 -> v_2 and v_2 -> v_1).)

If `vertexCount != numNodes` at the end, then the graph is not connected.

```
1  PriorityQueue<Edge> pq = new PriorityQueue<Edge>();
2  used[0] = true;
3  pq.addAll(vertexes[0].edges);
4  int mstWeight = 0, vertexCount = 1;
5  while (!pq.isEmpty()) {
6      Edge current = pq.poll();
7      if (used[current.vertex2]) continue;
8      mstWeight += current.cost;
9      used[current.vertex2] = true;
10     vertexCount++;
11     pq.addAll(vertexes[current.vertex2].edges);
12 }
```

Runtime: $O(V^2)$

This version is useful on dense graphs ($E \approx V^2$), because it is actually faster than the above version (no *log*() factor). Also, unlike other `adj` arrays, `adj[i][i]` is NOT set to 0 (rather, it is `INFINITY`).

```
1 int mstWeight = 0, vertexCount = 0;
2 boolean[] used = new boolean[numVertexes];
3 int[] dist = new int[numVertexes];
```

```
4  Arrays.fill(dist, INFINITY);
5  dist[0] = 0; // Start with an arbitrary vertex.
6  for (int i = 0; i < numVertexes; i++) {
7      // Find the edge with the smallest weight and who's vertexes aren't
8      // already used.
9      int curr = 0;
10     for (int j = 1; j < numVertexes; j++)
11         if (!used[j] && dist[j] < dist[curr]) curr = j;
12
13     if (dist[curr] == INFINITY) break; // We're done.
14
15     for (int j = 0; j < numVertexes; j++)
16         if (!used[j]) dist[j] = Math.min(dist[j], adj[curr][j]);
17
18     vertexCount++;
19     mstWeight += dist[curr];
20     dist[curr] = INFINITY;
21     used[curr] = true;
22 }
```

Permutations and combinations

Combinations: to only process combinations with n items, check if `Integer.bitCount(subset) == n`.

Permutations

Note: assumes items can only be chosen once.

Generates all the permutations of length `len` in lexicographical order (assuming `data` is sorted lexicographically). `data` is the input array and `perms` and `used` are empty arrays of the same size.

```
1  static int[] data, perms;
2  static boolean[] used;
3  static void perms(int position, int len) {
4      if (position >= len) {
5          // Process the permutation (it's stored in the perms array).
6      } else {
7          for (int i = 0; i < data.length; i++) {
8              if (!used[i]) {
9                  used[i] = true;
10                 perms[position] = data[i];
11                 perms(position+1, len);
12                 used[i] = false;
13             }
14         }
15     }
16 }
```

You can pre-process permutations by adding conditions to line #8. For example, if you want all the permutations of a string where first character of the original string isn't at an even index:

```
1  if (used[j] == false && (j != 0 || position % 2 != 0)) {}
```

Finding the lexicographically kth permutation

Runtime: $O(n^2)$, where n is the number of elements

`data` contains the input in lexicographically sorted order. `k` is 0 indexed.

```
1  static int[] findKthPerm(int[] data, int n, int k) {
2      int[] indices = new int[n];
3
4      // Find the factoradic representation of k.
5      int divisor = 1;
6      for (int place = 1; place <= n; place++) {
7          if (k / divisor == 0) break;
8          indices[n - place] = (k / divisor) % place;
9          divisor *= place;
10     }
11
12     // Use it to build the permuation.
13     for (int i = 0; i < n; i++) {
14         int index = indices[i] + i;
```

```
15          if(index != i) {
16              int temp = data[index];
17              for (int j = index; j > i; j--) data[j] = data[j-1];
18              data[i] = temp;
19          }
20      }
21
22      return data;
23  }
```

Lexicographically next permutation

Runtime: O(n log(n)), where n is the number of elements

Assumes that the data will compare correctly, ie. `data[i] < data[i+1]`. If this is not the case, you'll need a `indexOf(n)`, which returns the index of `n` in `data`.

To find the next permutation, we compute `first`, the right-most element whose right element is larger than itself, and `second`, the smallest element to the right of `first` that is larger than `first`. We then swap the elements at `first` and `second` and sort the elements to the right of `first`.

```
1   static int[] getNextPerm(int[] data, int[] currPerm) {
2       int[] nextPerm = Arrays.copyOf(currPerm, data.length);
3
4       int firstIdx = -1, secondIdx = -1;
5       for (int i = 0; i < data.length - 1; i++) {
6           if (currPerm[i] < currPerm[i+1]) firstIdx = i;
7           if (currPerm[i+1] > currPerm[firstIdx]) secondIdx = i + 1;
8       }
9       swap(nextPerm, firstIdx, secondIdx);
10      Arrays.sort(nextPerm, firstIdx+1, secondIdx+1);
11
12      return nextPerm;
13  }
```

Subset sum

Runtime: O(sn), where s is the sum we want to find in the set of n numbers

Note: `nums` is 1-indexed in these code snippets. Also, this algorithm is a special case of the 0/1 knapsack, where `weight = value`.

Positive numbers only

```
1 boolean[][] memo = new boolean[numNums+1][target+1];
2 // If the sum is 0, then we know we can make that sum.
3 for(int i = 0; i <= numNums; i++) memo[i][0] = true;
4 for (int n = 1; n <= numNums; n++) {
5     for (int t = 1; t <= target; t++) {
6         memo[n][t] = memo[n-1][t];
7         if (t >= nums[n]) memo[n][t] |= memo[n-1][t - nums[n]];
8     }
9 }
```

If you need to know which numbers were used, change lines 6-7 to:

```
1 if (memo[n-1][t]) {
2     memo[n][t] = true;
3     continue;
4 }
5 boolean take = false;
6 if (t >= nums[n]) take = memo[n-1][t - nums[n]];
7 if (take) isUsed[n][t] = true;
8 memo[n][t] = take;
```

And then you can retrieve which ones were used with this:

```
1 boolean[] willUse = new boolean[numNums+1];
2 for (int n = numNums, t = target; n > 0; n--) {
3     if (isUsed[n][t]) {
4         t -= nums[n];
5         willUse[n] = true;
6     }
7 }
```

Positive numbers and negative numbers

Invert all the negative numbers and add them to `target`, then run the algorithm like normal. For example, {-3, -2, 1, 4 =? 10} becomes {1, 2, 3, 4 =? 15}.

Alternatively, you can sum all the negative numbers and sum all the positive, and run the algorithm on `[negativeSum, positiveSum]` instead of `[0, target]`, with an offset of `negativeSum`.

Counting subsets

Modifying this to count how many ways you can make `target` is very easy. Change memo to an `int` array, line #3 to `memo[i][0] = 1;`, and line #8 to `+=`.

DisjointSet

Used by Kruskal's:

```
1  class DisjointSet {
2      int[] parents, ranks;
3
4      public DisjointSet(int numNodes) {
5          this.parents = new int[numNodes];
6          this.ranks = new int[numNodes];
7          for (int i = 0; i < numNodes; i++) parents[i] = i;
8      }
9
10     int findRoot(int index) {
11         if (parents[index] != index)
12             parents[index] = findRoot(parents[index]);
13         return parents[index];
14     }
15
16     // Union by rank. Returns false if the union did not occur.
17     boolean union(int index1, int index2) {
18         int root1 = findRoot(index1);
19         int root2 = findRoot(index2);
20         if (root1 == root2) return false;
21
22         /*
23         We put the tree with the lower rank directly under the root of the
24         other tree. If they have the same rank, we arbitrarily decide to place
25         the first under the second.
26         */
27         if (ranks[root1] <= ranks[root2]) parents[root1] = root2;
28         else parents[root2] = root1;
29         /*
30         This is the only thing that differs from the case where the first
31         tree's rank is less than the rank of the second tree.
32         */
33         if (ranks[root1] == ranks[root2]) ranks[root2]++;
34
35         return true;
36     }
37 }
```

Fenwick tree

Range update, point query

```
class FenwickTree {
    int size, ft[];
    public FenwickTree(int n) {
        this.ft = new int[n+1];
        this.size = n;
    }
    public FenwickTree(int[] arr) {
        this(arr.length);
        for (int i = 0; i < arr.length; i++) update(i+1, i+1, arr[i]);
    }
    // Updates the values at the point p.
    void update(int p, int val) {
        for (; p <= size; p += p&(-p)) ft[p] += val;
    }
    // Updates the values on the range [a, b].
    void update(int a, int b, int val) {
        update(a, val);
        update(b + 1, -val);
    }
    // Queries the value at the point p.
    int query(int p) {
        int sum = 0;
        for (; p > 0; p -= p&(-p)) sum += ft[p];
        return sum;
    }
}
```

Point update, range query

```
class FenwickTree {
    int size, ft[];
    public FenwickTree(int n) {
        this.ft = new int[n+1];
        this.size = n;
    }
    public FenwickTree(int[] arr) {
        this(arr.length);
        for (int i = 0; i < arr.length; i++) {
            update(i+1, arr[i]);
        }
    }
    // Updates the values at the point p.
    void update(int p, int val) {
        for (; p <= size; p += p&(-p))
            ft[p] += val;
    }
    // Queries the value at the point p.
    int query(int p) {
        int sum = 0;
```

```
        for (; p > 0; p -= p&(-p)) sum += ft[p];
        return sum;
    }
    // Queries the sum of the values on the range [a, b].
    int query(int a, int b) {
        return query(b) - query(a-1);
    }
}
```

Range update, range query

```
class FenwickTree {
    int size, ft1[], ft2[];
    public FenwickTree(int n) {
        this.ft1 = new int[n+1];
        this.ft2 = new int[n+1];
        this.size = n;
    }
    public FenwickTree(int[] arr) {
        this(arr.length);
        for (int i = 0; i < arr.length; i++) update(i+1, i+1, arr[i]);
    }
    // Updates the values at the point p.
    void update(boolean isOne, int p, int val) {
        for (; p <= size; p += p&(-p)) {
            if (isOne) ft1[p] += val;
            else ft2[p] += val;
        }
    }
    // Updates the values on the range [a, b].
    void update(int a, int b, int val) {
        update(true, a, val);
        update(true, b + 1, -val);
        update(false, a, val * (a-1));
        update(false, b + 1, -val * b);
    }
    // Queries the value at the point p.
    int query(boolean isOne, int p) {
        int sum = 0;
        for (; p > 0; p -= p&(-p))
            if (isOne) sum += ft1[p];
            else sum += ft2[p];
        return sum;
    }
    // Queries at point p.
    int query(int p) {
        return query(true, p) * p - query(false, p);
    }
    // Queries the sum of the values on the range [a, b].
    int query(int a, int b) {
        return query(b) - query(a-1);
    }
}
```

Uses

average from the FHSPS final and counting inversions

Miscellaneous

Built-in data structures

Maps

If you wish to use a custom class as Keys, then the class needs to implement `equals()` and `hashCode()`.

A `TreeMap` sorts the Keys according to their natural ordering or a provided Comparator. Operations are mostly *O(log(n))*.

A `LinkedHashMap` orders the Keys according to the order in which they were inserted. "Note that insertion order is not affected if a key is re-inserted into the map. A special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (access-order)." Operations are mostly *O(1)* (slightly slower than `HashMap`, except for iteration, which is *O(n)*).

A `HashMap` orders the Keys randomly. Operations are mostly *O(1)* (worst case *O(n)*), though iteration is *O(n+capacity)*.

Iterates over the values in the map, where K and V are the types of the Keys and Values in the map:

```
1  for (Entry<K, V> i : map.entrySet()) {
2      System.out.println(i.getKey() + " " + i.getValue());
3  }
```

Sets

`TreeSet` and `LinkedHashSet` have the same ordering as `TreeMap` and `LinkedHashMap`.

To iterate over the elements of a Set, you can use the Iterator returned by `set.iterator()` or with a for-each loop.

Edge (Dinic's)

```
1  class Edge {
2      int from, to, cap, flow;
3      Edge rev;
4      public Edge(int from, int to, int cap) {
5          this(from, to, cap, null);
6          // Change 0 to cap for undirected edges.
7          this.rev = new Edge(to, from, 0, this);
8      }
9      private Edge(int from, int to, int cap, Edge rev) {
10         this.from = from;
11         this.to = to;
12         this.cap = cap;
13         this.rev = rev;
14     }
15 }
```

FastScanner

```
1  class FastScanner {
2      BufferedReader br;
3      StringTokenizer st;
```

```
4       public FastScanner(InputStream in) {
5           br = new BufferedReader(new InputStreamReader(in));
6           st = new StringTokenizer("");
7       }
8       String next() throws IOException {
9           while(!st.hasMoreElements()) st = new StringTokenizer(br.readLine());
10          return st.nextToken();
11      }
12      String nextLine() throws IOException {
13          while (st == null || !st.hasMoreTokens())
14              st = new StringTokenizer(br.readLine());
15          return st.nextToken("\n");
16      }
17      int nextInt() throws IOException {
18          return Integer.parseInt(next());
19      }
20      long nextLong() throws Exception {
21          return Long.parseLong(next());
22      }
23      double nextDouble() throws Exception {
24          return Double.parseDouble(next());
25      }
26  }
```

Interval

```
1   // From the Tokyo hack pack.
2   class Intervals {
3       TreeMap<Integer, Integer> map = new TreeMap<Integer, Integer>();
4       public Intervals() {
5           map.put(Integer.MIN_VALUE, -1);
6           map.put(Integer.MAX_VALUE, -1);
7       }
8       void paint(int s, int t, int c) {
9           int p = get(t);
10          map.subMap(s, t).clear();
11          map.put(s, c);
12          map.put(t, p);
13      }
14      int get(int k) {
15          return map.floorEntry(k).getValue();
16      }
17  }
```

Treap

```
1  // From the Tokyo hack pack.
2  class Treap {
3      final int key, val;
4      final double p;
5      final Treap left, right;
6      Treap(int key, int val, double p, Treap left, Treap right) {
7          this.key = key;
8          this.val = val;
9          this.p = p;
10         this.left = left;
11         this.right = right;
12     }
13     Treap change(Treap left, Treap right) {
14         return new Treap(key, val, p, left, right);
15     }
16     Treap normal() {
17         if (left != null && left.p < p && (right == null || left.p < right.p)) {
18             return left.change(left.left, change(left.right, right));
19         } else if (right != null && right.p < p) {
20             return right.change(change(left, right.left), right.right);
21         }
22         return this;
23     }
24 }
25 Treap put(Treap t, int key, int val) {
26     if (t == null) return new Treap(key, val, Math.random(), null, null);
27     if (key < t.key) return t.change(put(t.left, key, val), t.right).normal();
28     if (key > t.key) return t.change(t.left, put(t.right, key, val)).normal();
29     return new Treap(key, val, t.p, t.left, t.right);
30 }
31 Treap remove(Treap t, int key) {
32     if (t == null) return null;
33     if (key < t.key) return t.change(remove(t.left, key), t.right);
34     if (key > t.key) return t.change(t.left, remove(t.right, key));
35     return merge(t.left, t.right);
36 }
37 Treap merge(Treap t1, Treap t2) {
38     if (t1 == null) return t2;
39     if (t2 == null) return t1;
40     if (t1.p < t2.p) return t1.change(t1.left, merge(t1.right, t2));
41     return t2.change(merge(t1, t2.left), t2.right);
42 }
```

Wavelet Tree

```
1  class WaveletTree {
2      int low = Integer.MAX_VALUE, high = Integer.MIN_VALUE;
3      // leftCount[i] is the count of numbers "going left" (ie. less than mid)
4      // at index i in arr.
5      int[] arr, leftCount, rightCount;
6      WaveletTree left = null, right = null;
7      WaveletTree (int[] arr) {
8          this.arr = arr;
9          for (int i : arr) {
10             low = Math.min(low, i); high = Math.max(high, i);
11         }
12         leftCount = new int[arr.length + 1];
13         rightCount = new int[arr.length + 1];
14         int mid = low + (high - low) / 2;
15         for (int i = 0; i < arr.length; i++) {
16             leftCount[i+1] = leftCount[i];
17             rightCount[i+1] = rightCount[i];
18             if (arr[i] > mid) rightCount[i+1]++;
19             else leftCount[i+1]++;
20         }
21         if (low == high) return;
22         int[] leftArr = new int[leftCount[arr.length]];
23         int[] rightArr = new int[rightCount[arr.length]];
24         for (int i = 0; i < arr.length; i++) {
25             if (rightCount[i] == rightCount[i+1]) leftArr[leftCount[i]] = arr[i];
26             else rightArr[rightCount[i]] = arr[i];
27         }
28         left = new WaveletTree(leftArr); right = new WaveletTree(rightArr);
29     }
30     // Finds the k-th lowest number in [leftIdx, rightIdx), where k is 1-indexed.
31     int kthLowest(int k, int leftIdx, int rightIdx){
32         if (low == high) return arr[leftIdx + k - 1];
33         if (leftCount[rightIdx] - leftCount[leftIdx] >= k)
34             return left.kthLowest(k, leftCount[leftIdx], leftCount[rightIdx]);
35         return right.kthLowest(k - leftCount[rightIdx] +
36                 leftCount[leftIdx], rightCount[leftIdx], rightCount[rightIdx]);
37     }
38     // Counts the number of values on [lo, hi] in indexes [leftIdx, rightIdx).
39     int count(int lo, int hi, int leftIdx, int rightIdx){
40         if (lo <= low && high <= hi) return rightIdx - leftIdx;
41         if (low == high) return 0;
42         // All the values are in the left subtree.
43         if (hi < right.low) return left.count(lo, hi, leftCount[leftIdx],
44                 leftCount[rightIdx]);
45         // All the values are in the right subtree.
46         if (left.high < lo) return right.count(lo, hi, rightCount[leftIdx],
47                 rightCount[rightIdx]);
48         return left.count(lo, hi, leftCount[leftIdx], leftCount[rightIdx])
49                 + right.count(lo, hi, rightCount[leftIdx], rightCount[rightIdx]);
50     }
51 }
```

Base conversion

To base 10

`Integer.parseInt(String str, int radix)` will convert a String from a given radix to base 10, where radix ≤ 36.

Manual base conversion is useful if the radix is higher than 36 or the base is non-standard. For example, you have to convert to base 7, but all the odd digits aren't allowed.

```
1  static String baseString = "02468ACE";
2  static int convertToBaseTen(String input, int radix) {
3      int sum = 0, pow = 1;
4      for (int i = input.length() - 1; i >= 0; i--, pow *= radix) {
5          sum += pow * baseString.indexOf(input.charAt(i));
6      }
7      return sum;
8  }
```

From base 10

`Integer.toString(int num, int radix)` will convert a number from base 10 to a given radix, where radix ≤ 36.

Manual base conversion:

```
1  static String baseString = "02468ACE";
2  static String convertFromBaseTen(int input, int radix) {
3      StringBuilder output = new StringBuilder();
4      while (input > 0) {
5          int index = input % radix;
6          output.insert(0, baseString.charAt(index));
7          input /= radix;
8      }
9      return output.toString();
10 }
```

Floating point

To base 10: Do the whole number part like normal. For the fractional part, use inverse powers of radix (eg. $\frac{1}{radix}$, $\frac{1}{radix^2}$, etc.).

From base 10: Do the whole number part like normal. For the fractional part:

1. Remove the whole number part
2. Multiply `input` by `radix`
3. Add the whole number part of `input` to the output
4. Repeat until the fractional part is zero

Euler's totient function

$\phi(n)$, or $\varphi(n)$, counts the positive integers less than or equal to an integer n that are coprime to n.

Properties/formulas:

- $\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right)$, where p is the distinct prime factors of n
- $n^e \pmod m = n^{e \pmod{\phi(m)}} \pmod m$
- $\phi(nm) = \phi(n)\phi(m)\frac{d}{\phi(d)}$, where $d = gcd(n, m)$

  Special cases:

  - If $gcd(n, m) = 1$, then $\phi(nm) = \phi(n)\phi(m)$
  - $\phi(2m) = \begin{cases} 2\phi(m) & \text{if } m \text{ is even} \\ \phi(m) & \text{if } m \text{ is odd} \end{cases}$
  - $\phi(n^m) = n^{m-1}\phi(n)$
- If p is prime, $\phi(p) = p - 1$
- If n is even, $\phi(2n) = 2\phi(n)$
- Euler's theorem: if a and n are coprime, $a^{\phi(n)} \equiv 1 \pmod n$
  - The special case where n is prime is Fermat's Little Theorem:

    $a^p \equiv a \pmod p$
- If p is prime and $k \geq 1$, then $\phi(p^k) = p^k - p^{k-1} = p^{k-1}(p-1) = p^k\left(1 - \frac{1}{p}\right)$
- If $n = \prod p^k$ (ie. the prime factorization of n), then $\phi(n) = \prod \phi\left(p^k\right)$
- Divisor sum: $\sum_{d \mid n} \phi(d) = n$
- $a \mid b \implies \phi(a) \mid \phi(b)$
- $n \mid \phi(a^n - 1)$, for $a, n > 1$
- $\phi(lcm(n, m))\phi(gcd(n, m)) = \phi(n)\phi(m)$
- $\phi(n)$ is even for $n \geq 3$. Moreover, if n has r distinct odd prime factors, $2^r \mid \phi(n)$.
- For any $a > 1$ and $n > 6$ such that 4 doesn't divide n, there exists an $l \geq 2n$ such that $l \mid \phi(a^n - 1)$
- $\frac{\phi(n)}{n} = \frac{\phi(rad(n))}{rad(n)}$, where $rad(n)$ is the radical of n
  - $rad(n) = \prod_{p \mid n} p$, where p is the distinct prime factors of n
- Menon's identity: $\sum_{1 \leq k < n} gcd(k-1, n) = \phi(n)d(n)$, where $d(n)$ is the number of divisors of n and $gcd(k, n) = 1$

There's a sieve in the Sieves document, but if you want to directly calculate $\phi(n)$:

```
1 int result = n;
2 for (int i = 2; i <= n; i++) if (isPrime(i)) result -= result / i;
```

Alternatively (and faster):

```
1  int result = n;
2  for (int i = 2; i * i <= n; i++)
3      if (n % i == 0) {
4          result -= result / i;
5          while (n % i == 0) n /= i;
6      }
```

Fast exponentiation

*Runtime: O(log(n) * operation)*
Since the multiplication of numbers is *O(1)*, fast expo of numbers is *O(log(n))*
Since matrix multiplication is $O(n^3)$, fast expo of matrices is $O(n^3 log(n))$

NOTE: if n isn't guaranteed to be positive, you need to rewrite the method to be `double fastExpo(double x, int n)` and then add `if (n < 0) return fastExpo(1.0/x, -n);` to the beginning of the method.

x^n can be rewritten as $(x^2)^{\frac{n}{2}}$ if n is even, and as $x(x^2)^{\frac{n-1}{2}}$ if n is odd. Therefore, we can use this method to quickly calculate exponents:

```
1  static int fastExpo(int x, int n) {
2      if (n == 0) return 1;
3      else if (n == 1) return x;
4      else if ((n & 1) == 0) return fastExpo(x * x, n >> 1);
5      else return x * fastExpo(x * x, n >> 1);
6  }
```

This is an iterative version thats a little faster:

```
1  int fastExpo(int x, int n) {
2      int result = 1;
3      while (n > 0) {
4          if ((n & 1) != 0){
5              result *= x;
6          }
7          n >>= 1;
8          x *= x;
9      }
10     return result;
11 }
```

If we want to calculate $x^n \mod m$, add `x %= m;` after line 2, change line 5 to `result = (result * x) % m;` and change line 8 to `x = (x * x) % m;`.

Fast Fourier transformation

Runtime: O(n log(n))

`LEN` is the length of the arrays. `LEN` must be at least double the degree of the polynomials being multiplied and a power of 2. You can use `Integer.highestOneBit(n) << 2` to find `LEN`.

NOTE: The polynomial must be *exactly* the one you wish to use. For example, if you are raising a polynomial to the k^{th} power, and you only care about indexes less than 50000, you must zero out all the indexes past 50000, every time you multiply.

If you're squaring a polynomial, you can remove a `fft()` call by multiplying `ar` and `ai` by themselves. If you're raising a polynomial to the k^{th} power, it's faster to have `n==0` return `null` and put a special case into the `multiply()` method.

```
1  // From the Tokyo hack pack.
2  static void fft(int sign, double[] real, double[] imag) {
3      int n = real.length, d = Integer.numberOfLeadingZeros(n) + 1;
4      double theta = sign * 2 * Math.PI / n;
5      for (int m = n; m >= 2; m >>= 1, theta *= 2) {
6          for (int i = 0, mh = m >> 1; i < mh; i++) {
7              double wr = Math.cos(i * theta), wi = Math.sin(i * theta);
8              for (int j = i; j < n; j += m) {
9                  int k = j + mh;
10                 double xr = real[j] - real[k], xi = imag[j] - imag[k];
11                 real[j] += real[k];
12                 imag[j] += imag[k];
13                 real[k] = wr * xr - wi * xi;
14                 imag[k] = wr * xi + wi * xr;
15             }
16         }
17     }
18     for (int i = 0; i < n; i++) {
19         int j = Integer.reverse(i) >>> d;
20         if (j < i) {
21             double tr = real[i]; real[i] = real[j]; real[j] = tr;
22             double ti = imag[i]; imag[i] = imag[j]; imag[j] = ti;
23         }
24     }
25 }
```

To use the above method:

```
1  static double[] multiply(double[] a, double[] b) {
2      double[] ar = a.clone(), br = b.clone();
3      double[] ai = new double[LEN], bi = new double[LEN];
4      fft(1, ar, ai);
5      fft(1, br, bi);
6      double[] cr = new double[LEN], ci = new double[LEN];
7      for (int i = 0; i < LEN; ++i) {
8          cr[i] = ar[i] * br[i] - ai[i] * bi[i];
9          ci[i] = ai[i] * br[i] + ar[i] * bi[i];
10     }
11     ifft(cr, ci);
12     return cr;
13 }
```

```
14 static void ifft(double[] real, double[] imag) {
15     fft(-1, real, imag);
16     for (int i = 0, n = real.length; i < n; ++i) {
17         imag[i] = -imag[i] / n;
18         real[i] /= n;
19     }
20 }
```

GCD, LCM, and EEA

Greatest Common Divisor

```
1  static int gcd(int a, int b) {
2      return ( b == 0 ? a : gcd(b, a%b) );
3  }
```

```
1  static int gcd(int a, int b) {
2      if (a < b) return gcd(b, a);
3      while (b != 0) {
4          int temp = b;
5          b = a % b;
6          a = temp;
7      }
8      return a;
9  }
```

Least Common Multiple

$$LCM(a,b) = \frac{ab}{GCD(a,b)}$$

Extended Euclidean Algorithm

Bezout's identity: $ax + by = d$, where a and b are non-zero integers, d is their GCD, and x and y are Bezout's coefficients.

```
1  static void extendedEuclideanAlgorithm(int a, int b) {
2      if (a < b) extendedEuclideanAlgorithm(b, a);
3      int s = 0, t = 1, r = b;
4      int oldS = 1, oldT = 0, oldR = a;
5      while (r != 0) {
6          int quotient = oldR / r;
7          int temp = r;
8          r = oldR - (quotient * temp);
9          oldR = temp;
10         temp = s;
11         s = oldS - (quotient * temp);
12         oldS = temp;
13         temp = t;
14         t = oldT - (quotient * temp);
15         oldT = temp;
16     }
17     print("Bezout coefficients: " + oldS + ", " + oldT);
18     print("GCD: " + oldR);
19     print("Quotients by the GCD: " + t + ", " + s);
20 }
```

The "quotients by the GCD" are a and b divided by the GCD. They may have an incorrect sign. Similarly, if either a or b is zero and the other is negative, the greatest common divisor that is output is negative, and all the signs of the output must be changed.

Misc

A set of integers S = $\{a_1, a_2, ...a_n\}$ can also be called coprime or setwise coprime if the greatest common divisor of all the elements of the set is 1.

If every pair in a set of integers is coprime, then the set is said to be pairwise coprime (or pairwise relatively prime, mutually coprime, or mutually relatively prime). Pairwise coprimality is a stronger condition than setwise coprimality; every pairwise coprime finite set is also setwise coprime, but the reverse is not true. For example, the integers 4, 5, 6 are setwise coprime, but they are not pairwise coprime (because gcd(4, 6) = 2).

Problem: Given the equation $ax + by = c$ and the values of a, b, and c, determine if there exists a solution for x and y.

Solution: x and y exist if c is divisible by the GCD of a and b, because it means that a, b, and c are multiples of the same number.

Problem: Given the equation $ax - by = 0$, find x and y.

Solution: x = b / GCD and y = a / GCD. The GCD is the largest common factor they share, so A * (B/gcd) = B * (A/gcd).

Matrices

Formatting: a $n \times m$ matrix is a matrix with n rows and m columns.

Terminology:
Square matrix: a $n \times n$ matrix

A matrix *mod* m is just a matrix in which every number has been *mod*ded by m. When doing this with matrix multiplication, add `sum %= m;` after line #6.

The identity matrix is a $n \times n$ matrix where the elements on the main diagonal (top-left to bottom-right) are ones and the rest are zeroes. It is similar to the number 1 in numerical mathematics.

Addition and subtraction

Add/subtract each number to its pair in the second matrix. (Note: the matrices must be the same size.)

$$\begin{bmatrix} 2 & 6 & 12 \\ 7 & -4 & 10 \end{bmatrix} + \begin{bmatrix} 15 & -44 & -3 \\ 8 & 0 & 9 \end{bmatrix} = \begin{bmatrix} 17 & -38 & 9 \\ 15 & -4 & 19 \end{bmatrix}$$

Scalar multiplication

Multiply every number in the matrix by the scale factor.

$$2 * \begin{bmatrix} 4 & 0 \\ 1 & -9 \end{bmatrix} = \begin{bmatrix} 8 & 0 \\ 2 & -18 \end{bmatrix}$$

Multiplication (dot product)

To get the number in row i and column j of the resulting matrix, we multiply each number in row i of the first matrix by each number in column j of the second and add the products.

If C = AB, for an $n \times m$ matrix A and an $m \times p$ matrix B, then C is an $n \times p$ matrix. Note that the number of columns in matrix A and the number of rows in matrix B must be the same.

To find A · B:

```
1  int[][] result = new int[a.length][b[0].length];
2  for (int i = 0; i < result.length; i++) {
3      for (int j = 0; j < result[0].length; j++) {
4          int sum = 0;
5          for (int k = 0; k < b.length; k++) {
6              sum += a[i][k] * b[k][j];
7          }
8          result[i][j] = sum;
9      }
10 }
```

Exponentiation

Matrix exponentiation is essentially the same as numerical exponentiation, and can be used with fast-expo techniques. Note that raising a matrix to the zeroth power returns the identity matrix and that the matrix must be square.

```
1  int[][] expo(int[][] matrix, int n) {
2      // identity() returns an identity matrix of the given size
3      int[][] result = identity(matrix.length);
4      while (n > 0) {
5          if ((n & 1) != 0){
6              result = multiply(result, matrix);
7          }
8          n >>= 1;
9          matrix = multiply(matrix, matrix);
10     }
11     return result;
12 }
```

Miscellaneous

Bell numbers

The number of partitions a set of n elements has. (Partition of a set: grouping of the set's elements into non-empty subsets, in such a way that every element is included in one and only one of the subsets.)

$B = \{1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437, 190899322, 1382958545\}$

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \quad \Big| \quad B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$$

The Bell numbers can be calculated with the Bell triangle (similar to Pascal's triangle), where B_n is the first number in the n^{th} row.

1
1 2
2 3 5
5 7 10 15
15 20 27 37 52

```
1  int[][] bell = new int[n+1][];
2  bell[0] = new int[]{1};
3  for (int i = 1; i <= n; i++) {
4      bell[i] = new int[i+1];
5      bell[i][0] = bell[i-1][bell[i-1].length - 1];
6      for (int j = 1; j <= i; j++) bell[i][j] = bell[i][j-1] + bell[i-1][j-1];
7  }
```

Catalan numbers

The only Catalan numbers C_n that are odd are those for which $n = 2^k - 1$. All others are even.

The only prime Catalan numbers are $C_2 = 2$ and $C_3 = 5$.

$C = \{1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670\}$

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^{n} \frac{n+k}{k} = \frac{1}{n+1}\sum_{i=0}^{n} \binom{n}{i}^2 = \int_0^4 x^n \frac{1}{2\pi}\sqrt{\frac{4-x}{x}}dx$$

$$C_0 = 1 \quad \Big| \quad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} = \frac{2(2n+1)}{n+2} C_n$$

Coupon collector's problem

Suppose there is an urn with n different coupons. What is the probability that more than t sample trials are needed to collect all n coupons? Or, given n coupons, how many coupons do you expect you need to draw with replacement before having drawn each coupon at least once? Alternatively, how many times do you expect to need to roll an n-sided die in order to have every number come up at least once?

The answer: nH_n, where n is the number of coupons and H_n is the n^{th} Harmonic number.

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

Digital root formula

The digital root of a non-negative n is the result of repeatedly summing the digits of n. Eg. the digital root of 65,536 is 7 because $dr(65536) = 6 + 5 + 5 + 3 + 6 = 25$ and $dr(25) = 2 + 5 = 7$.

$dr(n) = 1 + ((n - 1) \bmod 9)$. To generalize to another base b, change 9 to $b - 1$.

Double equality

It can be hard to check equality in doubles due to floating point errors. This method will check for equality if floating point errors are possible:

```
1 static double EPSILON = 1E-9;
2 static boolean equals(double one, double two) {
3     // Absolute equality.
4     if (Math.abs(one - two) < EPSILON) return true;
5     // Relative equality.
6     if (Math.abs(one - two) < EPSILON * Math.max(Math.abs(one),
7             Math.abs(two))) return true;
8     return false;
9 }
```

10^{-9} is generally a good value for `EPSILON`. Also, most problems involving doubles will tell you how much error two values can have and still be considered equal.

Evaluating polynomials at a point

Given an array of the coefficients of a polynomial you can find the value of the polynomial evaluated at `x` (ie. $f(x)$). This works by the remainder theorem.

```
1 int value = coefficients[0];
2 for (int j = 1; j <= polynomialDegree; ++j) {
3     value *= x;
4     value += coefficients[j];
5 }
```

Note that the array of coefficients should be in descending order. If `P(x) = 3x^2 - 2x + 1`, then the array should equal `{3, -2, 1}`.

Fibonacci sequence

$F = \{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610\}$

If $5x^2 + 4$ or $5x^2 - 4$ is a perfect square, then x is part of the Fibonacci sequence.

Every 3^{rd} number of the sequence is even and more generally, every k^{th} number of the sequence is a multiple of F_k. In fact, it satisfies the stronger divisibility property $GCD(F_m, F_n) = F_{GCD(m,n)}$.

Any three consecutive Fibonacci numbers are pairwise coprime, ie. $GCD(F_n, F_{n+1}) = GCD(F_n, F_{n+2}) = GCD(F_{n+1}, F_{n+2}) = 1$.

Every prime number p divides a Fibonacci number that can be determined by the value of p (mod 5). If p is congruent to 1 or 4 (mod 5), then p divides F_{p-1}, and if p is congruent to 2 or 3 (mod 5), then, p divides F_{p+1}. The remaining case is that $p = 5$, and in this case p divides F_p.

You can use matrices to calculate the Fibonacci sequence. This matrix representation gives the following closed expression for the Fibonacci numbers:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \quad \Big| \quad \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

Generalization

If $F_0 = a$ and $F_1 = b$, then:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} b \\ a \end{bmatrix}$$

If $a = b = 1$, then the Fibonacci sequence will be "shifted" to the left by one. For example, 8 is the 6^{th} Fibonacci number, but would be the 5^{th} in the generalized Fibonacci sequence.

(If $a = 0$ and $b = 1$, then it will be the normal sequence.)

Finding the average of two values

Obviously, you can do it like this: `int c = (a + b) >> 1;`

However, that can lead to overflows. This will not: `int c = a + ((b - a) >> 1);`

Alternatively: `((x ∧ y) >> 1) + (x & y)`

Log properties

$$log_b(x) = \frac{log_y(x)}{log_y(b)}, \text{ where } y \text{ is any base}$$

Pascal's triangle

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Pascals triangle is a triangular array of the binomial coefficients, that is $(x + y)^n$. The rows (designated by n) and columns (designated by k) are 0-indexed. So 6 is at row 4, column 2. This is also known as n *choose* k or $\binom{n}{k}$.

```
1  int[][] pascals = new int[n+1][];
2  pascals[0] = new int[]{1};
3  for (int i = 1; i <= n; i++) {
4      pascals[i] = new int[i+1];
5      pascals[i][0] = 1;
6      pascals[i][i] = 1;
```

```
    for (int j = 1; j <= i - 1; j++)
        pascals[i][j] = pascals[i-1][j-1] + pascals[i-1][j];
}
```

Given the n$^{\text{th}}$ row of Pascal's triangle (in `coefficients`) and values of a and b (from the expression $(ax + by)^{pow}$), this will give you the coefficients of the expanded binomial expression.

```
for (int x = pow, y = 0; y <= pow; x--, y++) {
    coefficients[y] *= Math.pow(a, x) * Math.pow(b, y);
}
```

Pigeonhole principle

The pigeonhole principle states that if k objects are placed into n boxes, then at least one box must hold at least $\lceil \frac{n}{k} \rceil$ objects.

Primality

This is a longer isPrime method, but one that is faster than the naïve version.

```
static boolean isPrime(int n) {
    if (n <= 1) return false;
    else if (n < 4) return true;
    else if (n % 2 == 0) return false;
    else if (n < 9) return true;
    else if (n % 3 == 0) return false;
    else {
        // Only numbers of the form 6k - 1 and 6k + 1 can be prime (though
        // they are not guaranteed to be prime).
        for(int i = 6; i*i <= n; i += 6) {
            if (n % (i-1) == 0) return false;
            if (n % (i+1) == 0) return false;
        }
        return true;
    }
}
```

Prime factorization

This finds all the prime factors (and not non-prime factors) because of the same principle as the prime sieve.

```
TreeMap<Integer, Integer> factors = new TreeMap<>();
int origNum = num;
for (int i = 2; i <= origNum; i++) {
    while (num % i == 0) {
        if (factors.containsKey(i)) factors.put(i, factors.get(i) + 1);
        else factors.put(i, 1);
        num /= i;
    }
    if (num == 1) break;
}
```

Pythagorean triple

A Pythagorean Triple is 3 numbers, a, b, c, such that $a^2 + b^2 = c^2$. For all Pythagorean Triples, there exist positive integers x and y, with x > y, such that:

$$a = x^2 - y^2 \quad \Big| \quad b = 2xy \quad \Big| \quad c = x^2 + y^2$$

Sequences

Sum of an arithmetic sequence: $\frac{n(a_1+a_n)}{2}$

Sum of a geometric sequence: $a_1(\frac{1-r^n}{1-r})$

Sum of infinite geometric sequence (where $|r| < 1$): $\frac{a}{1-r}$

Sum of the first n numbers: $\frac{n(n+1)}{2}$

Sum of the first n odd numbers: n^2

Sum of the first n even numbers: $n(n+1)$

Sum of the first n squares: $\frac{n(2n+1)(n+1)}{6}$

Sum of the first n cubes: $(\frac{n(n+1)}{2})^2$

Stirling numbers (second kind)

The number of ways to partition a set of n distinguishable objects into k non-empty subsets (denoted $S(n,k)$ or $\left\{ {n \atop k} \right\}$).

$$S(n,k) = S(n-1,k-1) + k * S(n-1,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$$

n / k	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	0	1									
2	0	1	1								
3	0	1	3	1							
4	0	1	7	6	1						
5	0	1	15	25	10	1					
6	0	1	31	90	65	15	1				
7	0	1	63	301	350	140	21	1			
8	0	1	127	966	1701	1050	266	28	1		
9	0	1	255	3025	7770	6951	2946	462	36	1	
10	0	1	511	9330	34105	42525	22827	5880	750	45	1

The sum over the values for k of the Stirling numbers of the second kind, gives us $B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$, ie. the n^{th} Bell number.

Stirling's approximation

$$\ln(n!) \approx n\ln(n) - n \quad \Big| \quad n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Sieves

Sieve of Eratosthenes

Runtime: O(n log(log(n)))

The Sieve of Erastosthenes computes primes by marking off multiples of all the primes before the current prime. The limit of how large a number you can check with this is based on how large an array you can make. However, you can double the max number by using a Sieve that only checks odd numbers. If you need a really large sieve, use a BitSet.

```
1 boolean[] isComposite = new boolean[n+1];
2 isComposite[0] = isComposite[1] = true;
3 for (int i = 2; i*i <= n; i++) {
4     if (!isComposite[i])
5         for (int j = i*i; j <= n; j += i)
6             isComposite[j] = true;
7 }
8 return isComposite;
```

You can count the number of prime factors a number has by changing lines 5-7 to:

```
1 if (numPrimeFactors[i] == 0)
2     for (int j = i*2; j <= upperBound; j += i)
3         numPrimeFactors[j]++;
```

(A number i is prime if `numPrimeFactors[i]` is 0.)

Totient sieve

This can also be used as a prime sieve, because `p` is prime if `phi[p] == p-1`.

```
1 int[] phi = new int[n+1];
2 for (int i = 0; i <= n; i++) phi[i] = i;
3 for (int i = 2; i <= n; i++)
4     // If you don't have a prime sieve, 'if(phi[i] == i)' also works.
5     if (!isComposite[i])
6         for (int j = i; j <= n; j += i)
7             phi[j] -= phi[j] / i;
```

Mod inverse sieve

The modular inverse of `i` *mod* `m`.

```
1 int[] modInverse = new int[n+1];
2 modInverse[1] = 1;
3 for (int i = 2; i <= n; i++)
4     modInverse[i] = ( -(m/i) * modInverse[m % i] ) % m + m;
```