Team 386491

Cheatsheet for SOMERANDOMCONTEST

Frank Maurix Patrick Shaw Luca Weibel

Contents

1 General Stuff

1.1 Running times

Value	Possible running times	Which algorithms to use?
$n \leq 10^9$	$O(1), O(\log(n)), O(\sqrt{n})$	Function, Binary Search
$n \leq 10^6$	$O(n), O(n \cdot \log(n))$	Greedy, sorting, Binary Search + Greedy, Divide and Conquer
$n \leq 10^3, W \leq 10^3$	$O(n \cdot W), O(n^2)$	Dynamic Programming with a table $n \times W$ or $n \times n$
$n \leq 10^2$	$O(n^3)$	All-pairs shortest path
$n \leq 16$	$O(2^n), O(n \cdot 2^n)$	Brute-force all bitstrings with size n
$n \leq 8$	$O(n!)$	Brute-force all permutations of n things

1.2 Template

```
import java.util.*;
class ScaryProblem {
   static Scanner sc = new Scanner(System.in);
   void run() {
       // Insert magic here
   }
   public static void main(String args[]){
      int runs = sc.nextInt();
      for (int i = 0; i < runs; i++) {
         new ScaryProblem().run();
}}}
```

1.3 Sorting

When sorting arrays, never use `int[]`, `double[]` or `char[]`, but always `Integer[]`, `Double[]` or `Character[]`.

To sort arrays, use `Arrays.sort(array);`. For `ArrayList`, use `Collections.sort(ArrayList);`. For sorting in reverse order, use `Arrays.sort(array, Collections.reverseOrder());` or `Collections.sort(arraylist, Collections.reverseOrder());`. When sorting objects you specified yourself or when sorting in some non default way, use a `Comparable` Some examples:

```
Item[] items;
Arrays.sort(items);
class Item implements Comparable<Item> {
    int price, reliability, coolfactor;
    public int compareTo(Item other) {
        //If a>other.b (when sorting in increasing order), return positive. If a==other.b, return 0. else, return negative.
        return other.price-price;//Same:
        if(price < other.price) return 1;
        if(price > other.price) return -1;
        return 0;
}}
//Sort in decreasing order based on item price
```

1.4 Outputting

When outputting a lot of data, `System.out.print` may be too slow. Instead, do: `import java.io.*;`

```
BufferedWriter out = new BufferedWriter(
     new OutputStreamWriter(System.out));
```

Then, use `out.write(SOMETHING);` to output, where `out.newLine();` creates a line break. After having put everything in `out`, use `out.flush();` to actually output it. Note that this will require some try/catch statements. Also, sometimes values don't get properly converted to strings. Use `String.valueOf()` for that.

1.4.1 Outputting to a file

Note: every time your code is executed, the file will be overwritten if it has already been created. Make sure that file is closed (not open in any program) before running your code. Using this code, you can keep using `System.out.print()` / `println()`. Use `System.err.println()` for testing, debugging, etc. Let `LOCATION` be a path + file name where the output should be stored, for example:
`C://Users/DERP/Desktop/output.txt`

```
import java.io.*;
public class ScaryProblem {
    public static void main(String[] args)
            throws Exception {
        System.setOut(new PrintStream(
                new BufferedOutputStream(
                new FileOutputStream(
                "LOCATION")), true));
}}
```

1.5 Miscellaneous tips

- `Math.sqrt()` is very inaccurate. Apply it last. E.g. $\sqrt{\frac{a}{b}}$ is better than $\frac{\sqrt{a}}{\sqrt{b}}$
- `String s += something` / concatenating strings takes time relative to length. Use a `StringBuilder` instead.
- For infinity, use `Integer.MIN_VALUE` $= -\infty$ and `Integer.MAX_VALUE` $= \infty$. Be wary of over/underflow.
- For rounding to n decimals, use `DecimalFormat`. E.g. with $n = 4$ (*Note: if* $x = 0.123$*, then* $p = 0.1230$ *and* $q = 0.123$):

```
import java.text.*;
DecimalFormat four = new DecimalFormat("#0.0000");
String p = four.format(x);
Double q = Double.valueOf(p);
```

- In a DP, when you have for example a list of locations each containing an amount of items and each location has its own price/cost per item, sorting may help. Only sort on the price/cost properties dependent on the amount of items such that the location with the best items will be considered first when looking for the items. Don't sort on properties relevant to either buy or not buy (e.g. cost to get to the location).
- `Arrays.fill(A, x);` Fills all cells in array `A` with value `x`. `Arrays.fill(A, i, j, x);` Does the same, but then only for the cells `A[k]`, where $i \leq k < j$
- `Arrays.binarySearch(A, x);` Binary search on sorted array `A` for value `x`. Returns index or negative number if not present. Comparable must be defined. `Arrays.binarySearch(A, x);` Does the same, but then only for the cells `A[k]`, where $i \leq k < j$

- Many built in Data Structures offer a constructor where you have the option to specify initial capacity. Use this with capacity (not too big) $\geq$ than capacity it will ever reach to speed up.

2 Data Structures

2.1 LinkedList/ArrayDeque (Built in)

Note: also consider an `ArrayDeque`. It works very similar, but uses less overhead. Good for Queues and add/poll/peek first/last. Not for operations at current position.
When to use: when a doubly-linked list is required.
Example: Add/remove/get first/last. Add/remove/get some element when you have the pointer to that element.
Creation: `LinkedList<Object> list = new LinkedList();`
Operations: *When a `ListIterator` is used, add/remove using the iterator, else you get errors*

- `list.addFirst()/ list.getfirst()/ list.pollFirst()` Respectively adds an element to the front, gets an element from the front, retrieves and removes the first element. Change First to Last for operations at the end. O(1)
- `list.add(index, element)/ list.get(index)/ list.remove(index)`. O(n)

Getting all elements(in order): use a `ListIterator`.

2.1.1 ListIterator

What it is: Basically a cursor in between two elements.
When to use: to read the contents of a list or to simulate a cursor on some object.
Creation: `ListIterator<Object> cursor = list.listIterator()`. This places the cursor in front of the first element of the list. To place it in front of element i, use `list.listIterator(i)`. To place it after the last element, use `i = list.size();`
Note: When using a `ListIterator`, all add/remove instructions on a list should be done by the `ListIterator`, else errors will occur.
Operations: the following all run in O(1) for `LinkedList`

- `cursor.hasNext()/cursor.hasPrevious()`
- `cursor.next()/cursor.previous()`, retrieves next/ previous and moves 1 element forward/backward.
- `cursor.add(element)`, inserts element at current position and moves the cursor after element.
- `cursor.next(); cursor.remove();`, removes the element to the right of the cursor.
- `cursor.previous(); cursor.remove();`, removes the element to the left of the cursor.

2.2 HashSet (Built in)

When to use: When a set is needed.
Creation: `HashSet<Type> set = new HashSet();`
Warning: *Doesn't allow duplicate keys*
Operations:
Run in $O(1)$ time, assuming simple uniform hashing

- `map.add(Type)`. If Type already present, doesn't add.
- `map.remove(Type)`
- `map.contains(Key)` returns either `true` or `false`

Getting all elements: O(n)

```
Iterator iterator = newset.iterator();
while (iterator.hasNext()){
   System.out.println(iterator.next());
}
```

2.2.1 HashSets with custom classes/objects

```
HashSet<VeryProblemSuchScareSoWow> set;
class VeryProblemSuchScareSoWow {
   Key key; Values v1, v2;
   @Override
   public int hashCode() {
      return key.hashCode();
   }@Override
   public boolean equals(Object o) {
      boolean a = v1 == ((
         VeryProblemSuchScareSoWow) o).v1;
      a = a && v2 == ((
         VeryProblemSuchScareSoWow) o).v2;
      return a; //True if equal
}}
```

2.3 HashMap (Built in)

When to use: to map a set of keys to a set of values.
Example: For a set of objects, if they are identified by a `String`, you can store the objects in an array(list) and use a `HashMap<String, Integer>` where the `Integer` is the index of the object in the array identified by `String`.
Creation: `HashMap<Key, Value> map = new HashMap();`
Custom objects in `HashMap`: See `HashSet`
Warning: *Doesn't allow duplicate keys*
Operations:
Most run in $O(1)$ time, assuming simple uniform hashing

- `map.put(Key, Value)`, binds Key to Value. If Key already present, replaces old value with new one.
- `map.get(Key)`, gets Value associated to Key.
- `map.remove(Key)`
- `map.containsKey(Key)` returns either `true` or `false`
- `map.containsValue(Value)` similar, but O(n) time.)

Getting all keys/values: O(n)
`for(Key key : map.keySet())map.get(key);`

2.4 PriorityQueue (Built in)

When to use: when a min-heap/max-heap is needed.
Creation: `PriorityQueue<Type> Q=new PriorityQueue();`
Default order: Lowest priority is on top of the queue.
Operations:

- `Q.add(element)` O(log(n))
- `Q.peek()` look at element on top of Queue. O(1)
- `Q.poll()` retrieve and remove top element. O(log(n))
- `Q.remove(key)` O(n)
- `Q.contains(key)` O(n)

Getting all elements as array (unsorted):
`Type[] A = new Type[Q.size()]; Q.toArray(A);`
Using custom sort for determining order:

```
PriorityQueue<Type> Q=new PriorityQueue();
class Type implements Comparable<Type> {
   @Override
   public int compareTo(Type o) {
      return -1;//if this comes before o
      return 0;//if it doesn't matter
      return 1;//if o comes before this
}}
```

2.5 TreeMap/TreeSet (Built in)

Note: for custom elements/custom ordering a `Comparable` must be defined
When to use: when a red/black tree (balanced binary search tree) is needed.

Creation: `TreeMap<Key, Value> T = new TreeMap();` or do `TreeSet<Key> T ...`
Operations: run in $O(\log(n))$.
TreeSet: `add(K)`, `remove(K)`, `contains(K)`, `first()` (minimum), `higher(K)`(successor)
TreeMap: `put(K, V)`, `get(K)`, `remove(K)`, `firstKey()` (minimum key), `lastKey()`(maximum), `containsKey(K)`, `ceilingKey(K)` (minimum key such that $key \geq K$), `floorKey(K)`, `higherKey(K)`(strictly greater successor)

2.6 Bitmask (Built in)

When to use: when modifications of bits are needed. Also needed for Subset DP.
How to use: just have `int x` variables.
Note: when numbering the bits, the rightmost bit has the lowest index. zerobased indexing is used. So, for 0010001, the 0^{th} and 4^{th} bit are on.
Operations:

- `P | Q` $= P \cup Q$
- `P & Q` $= P \cap Q$
- `P & ~Q` $= P \setminus Q$
- `1<<i` returns a number with only the i^{th} bit on.
- `x & (1<<i)` returns 0 if the i^{th} bit is off, not 0 if it is on.
- `x<<i` shifts the bits i places to the left/multiplies by 2^i
- `x>>i` shifts the bits i places to the right/divides by 2^i
- `x |= (1<<i)` turns on the i^{th} bit in `x`.
- `x &= ~(1<<i)` turns off the i^{th} bit in x.
- `x ^= (1<<i)` toggles the state of the i^{th} bit in `x`.
- `x & (-x)` returns rightmost 1 (least significant 1) of `x`. E.g. `x = 10110` $\rightarrow$ `x & (-x)= 00010`.
- `~x & x+1` turns on only the rightmost 0 of `x`. E.g. `x = 01011` $\rightarrow$ `~x & x+1 = 00100`.
- `((1 << i)-1)<< j` turns on bits `j` upto and including `i+j-1`. E.g. `i = 4, j = 2` $\rightarrow$ `((1 << i)-1)<< j = 00111100`
  Watch out for overflow

2.7 BigInteger (Built in)

When to use: When dealing with very big integers ($\geq 2^{63} (\approx 10^{18})$) to prevent overflow from occuring.
Creation: `BigInteger x=new BigInteger(String value)` or `BigInteger y=new BigInteger(String value,int radix)` (`radix` is the base of the number system, 10 by default, 2 for binary numbers, etc.)
Operations: *Note: operations are quite slow and don't modify x or y, only return a new BigInteger*

- `x.intValue(),x.longValue()` an int/long of the value in the bigInteger. Doesn't take overflow into account.
- `x.add(y),x.subtract(y),x.multiply(y),x.divide(y)`
- `x.negate()` returns -x
- `x.max(y),x.min(y)` returns Math.max(x,y)/min(x,y)
- `x.gcd(y)`

2.8 Union Find Disjoint Set

When to use: When merging disjoint sets/checking which element is in which set.
Creation: First 4 lines of code, where $n \in \mathbb{N}$ indicates the number of elements, each identified by a unique number $0 \leq i < n$. Each set has a number in that range too, but some may not appear.
Operations: $O(1)$ (amortized, averages out)

- `return numSets;` number of disjoint sets
- `return S[find(x)];` number of elements in the same set as x

```
int[] P=new int[n];int[] rank=new int[n];
int[] S=new int[n]; int numSets=n;
for(int i=0; i<n; i++) P[i] = i;
Arrays.fill(S,1);
int find(i) {
    if (P[i] == i) return i;
    else {
        int R = find(P[i]); P[i] = R;
        return R;
}}
boolean isSame(int i, int j) {
    return find(i) == find(j);
}
void union(int i, int j) {
    if(isSame(i, j)) return;
    numSets--; int x=find(i), y=find(j);
    S[x] += S[y]; S[y] = S[x];
    if (rank[x] > rank[y]) P[y] = x;
    else P[x] = y;
    if (rank[x]==rank[y]) rank[y]++;
}
```

2.9 Trie

When to use: when doing something for strings with same prefix. Some other things with strings too.
Creation: `Node root = new Node(null, false, null);`
Implementation details: Assumes alphabet consists of set {A,B,...,Z}(uppercase only). For lowercase only: change all `-'A'` into `-'a'`. For more complex alphabet (e.g. A-Z, a-z, 0-9): use `HashMap`(see section 2.3) of nodes to represent the children instead of array. Changes insert into $O(L)$, but generally slower+more space.
Sorted retrieval: DFS (see section 5.2), first report current node (if necessary), then visit children.
Retrieving strings: Either visit path from root to current node, or only store for each node with `used = true` the String (yields same insert time).
Operations: (`L=word.length;A=C.length;`)
Note: Operations are done on word `W` (`char[]` array) .

- `root.insert(W,0)` $O(L \cdot C)$
- `root.search(W,0)` Return `Node`/`null`(no node) $O(L)$
- `removeWord(W)` Remove only `W`. $O(L)$
- `removePrefix(W)` Remove all with prefix `W`. $O(L)$

```
class Node {
   Node[] C = new Node[26];//children
   Character ch;//last char of prefix
   Node P;//parent
   boolean used;//prefix in dictionary
   int nOC = 0;//number of children
   Node(Character c,boolean u,Node p){..}
   void insert(char[] W, int I) {
      if (C[W[I] - 'A'] == null) {
         nOC++;
         C[W[I]-'A'] =
            new Node(W[I],true,this);
         if (I != W.length - 1) {
            C[W[I]-'A'].used=false;
            C[W[I]-'A'].insert(W,I+1);
         }
      } else if (I == W.length - 1) {
         C[W[I] - 'A'].used = true;
      } else {
         C[W[I] - 'A'].insert(W, I + 1);
   }}
   Node search(char[] W, int I) {
      if (I == W.length - 1) {
```

```
            return C[W[I]-'A'];
        } else if (C[W[I]-'A'] == null) {
            return null;
        } else {
            return C[W[I]-'A'].search(W,I+1);
    }}
    //Removes this subtree+nodes above if
    //they are redundant
    void trieCleanup() {
        if (P != null) {
            P.C[ch - 'A'] = null;
            P.nOC--;
            if (P.nOC == 0 && !P.used) {
                P.trieCleanup();
}}}}
void removePrefix(char[] W) {
    Node R = root.search(W, 0);
    if (R != null) {
        R.trieCleanup();
}}
void removeWord(char[] W) {
    Node R = root.search(W, 0);
    if (R == null) {
    } else if (R.nOC == 0) {
        R.trieCleanup();
    } else {
        R.used = false;
}}
```

2.10 Binary Indexed Tree (Fenwick Tree)

Note: `SIZE` contains the size of BIT (Binary Indexed Tree). If the BIT stores values with indices in the range of $[1...N]$ *(0 not supported)*, then `SIZE = N + 1;`

When to use: when wanting to know the sum of values in a certain range of indices, while still being able to update those values.

Creation: See first line of code.

2.10.1 1D

Operations: $\mathrm{O}(\log(n))$

- `sum(i)` returns the sum of value with indices $[1 \ldots i]$
- `sum(i, j)` returns the sum of value with indices $[i \ldots j]$
- `sum(i, i)` returns the value at index i.
- `set(i, val)` changes the value at index i to val.
- `add(i, val)` adds val to the value at index i.

```
int[] BIT = new int[SIZE];
void add(int i, int val) {
    while (i < SIZE) {
        BIT[i] += val;
        i += (i & -i);
}}
int sum(int i, int j) {
    return sum(j) - sum(i - 1);
}
void set(int i, int val) {
    add(i, val - sum(i, i));
}
int sum(int i) {
    int sum = 0;
    while (i > 0) {
        sum += BIT[i];
        i -= (i & -i);
    }
    return sum;
}
```

2.10.2 2D

Note: goal of operations are rather similar to 1D. Don't call the methods with a Y at the end of its name yourself.

```
int[][] BIT = new int[SIZE][SIZE];
void add(int x, int y, int val) {
    while (x < SIZE) {
        addY(x, y, val);
        x += (x & -x);
}}
void addY(int x, int y, int val) {
    while (y < SIZE) {
        BIT[x][y] += val;
        y += (y & -y);
}}
int sum(int x1, int y1, int x2, int y2) {
    return (sum(x2, y2) + sum(x1-1, y1-1)
        - sum(x1-1, y2) - sum(x2, y1-1));
}
void set(int x, int y, int val) {
    add(x, y, val - sum(x, y, x, y));
}
int sum(int x, int y) {
    int sum = 0;
    while (x > 0) {
        sum += sumY(x, y);
        x -= (x & -x);
    }
    return sum;
}
int sumY(int x, int y) {
    int sum = 0;
    while (y > 0) {
        sum += BIT[x][y];
        y -= (y & -y);
    }
    return sum;
}
```

3 Basic Math / Number Theory

3.1 Prime numbers

3.1.1 isPrime

Pre: $n \in \mathbb{N}$

Out: True if n is a prime number, false otherwise

RT: $\mathrm{O}(\sqrt{n})$

```
if(n <= 1) return false;
if(n == 2) return true;
if(n % 2 == 0) return false;
for (int i = 3; i * i <= n; i += 2) {
    if (n % i == 0) return false;
}
return true;
```

3.1.2 Sieve Of Eratosthenes

Pre: $n \in \mathbb{N} \wedge n \geq 1$

Out: Array P, where for each $i \in \mathbb{N}$ at most n, if `P[i] = true`, then i is prime, false otherwise.

RT: *roughly* $\mathrm{O}(n \log(\log(n)))$

```
boolean[] P = new boolean[n+1];
Arrays.fill(P, true);
P[0] = false; P[1] = false;
for (int i = 2; i <= n; i++) {
    if (!P[i]) continue;
    for(int j=i*i; j<=n; j+=i) P[j]=false;
    primes.add(i);//List of primes <= N
```

```
}
return P;
```

3.1.3 Prime Factorization

Pre: $n \in \mathbb{N} \wedge n \geq 1$
Out: List of the prime factors of n
RT: $O(\sqrt{n})$
Example: $n = 28 \rightarrow 2, 2, 7$, since $28 = 2 \cdot 2 \cdot 7$

```
ArrayList<Integer> F = new ArrayList();
int i = 2;
while (n != 1 && (i * i <= n)) {
    while (n % i == 0) {
        n /= i; F.add((int)i);
    }
    i += 2; if (i == 4) i--;
}
if (n != 1) F.add(n);//if n is prime
return F;
```

3.2 Euclidean algorithm (GCD/LCM)

Note: $GCD(a, b, c, d) == GCD(a, GCD(b, GCD(c, d)))$
Pre: $a, b \in \mathbb{N} \wedge \neg(a = 0 = b)$
Out: For GCD: the greatest common divisor $\in \mathbb{N}$ of a and b. For LCM: the least common multiple $\in \mathbb{N}$ of a and b.
RT: $O(\log(\max(a, b)))$

```
int GCD(int a, int b) {
    if (a == 0) return b;
    return GCD(b % a, a);
}
int LCM(int a, int b) {
    return a*b/GCD(a, b);
}
```

3.2.1 Extended Euclidean algorithm

Pre: $a, b \in \mathbb{N} \wedge \neg(a = 0 = b)$
Out: $c \in \mathbb{N} \wedge x, y \in \mathbb{Z}$ such that $\text{GCD}(a, b) = c = a \cdot x + b \cdot y$
RT: $O(\log(\max(a, b)))$

```
int c, x = 0, y = 1;
void ExtendedEuclid(int a, int b) {
    if (a == 0) c = b;
    else {
        ExtendedEuclid(b % a, a);
        int temp = x;
        x = y - x * (b / a);
        y = temp;
}}
```

3.3 Combinatorics

Also see section 6 on Dynamic Programming
Number of distinct subsets of set of size n: 2^n
Number of distinct permutations: $n!$
Pick k out of n elements:

	duplicates	$\neg$duplicates
order	n^k	$\frac{n!}{(n-k)!}$
$\neg$order	$\binom{n-1+k}{k}$	$\binom{n}{k}$

Examples: take $\#V = n$
n^k: number with k digits from V
$\frac{n!}{(n-k)!}$: number with k distinct digits from V
$\binom{n}{k}$: # of sets of k distinct elements in V
$\binom{n-1+k}{k}$: # of sets of k elements in V

Where: $\binom{m}{m} = \binom{m}{0} = 1$ and $\binom{m}{n} = \frac{m!}{n! \cdot (m-n)!}$

4 Computational Geometry

4.1 Point, Line(segment) and Circle

Some notes on the implementation:

- `EPS` defines the maximum error margin that two values may deviate from each other to be considered equal.
- The constructor `Point(Point p, Point q)` turns the line starting at `p` and ending at `q` into a vector, represented by a `Point`.
- Currently sorts `Points` such that they are sorted by x-coordinate in increasing order. If tied, it sorts by y-coordinate in increasing order.

```
double EPS = Math.pow(10, -8);
class Point implements Comparable<Point> {
    double x, y;
    Point() { this(0.0, 0.0); }
    Point(Point p) { this(p.x, p.y);}
    Point(double X, double Y) {x=X; y=Y;}
    Point(Point p, Point q){
        this(q.x-p.x,q.y-p.y);
    }
    boolean equals(Point p) {
        return ((Math.abs(x-p.x) < EPS) &&
                (Math.abs(y-p.y) < EPS));
    }
    public int compareTo(Point p) {
        if (equals(p)) {
            return 0;
        } else if (Math.abs(x-p.x) < EPS){
           return (int)Math.signum(y-p.y);
        } else {
           return (int)Math.signum(x-p.x);
}}}
class Line {
    Point a, b;
    Line(Point A, Point B) { a=A; b=B; }
    boolean isParallel(Line l) {
        double d=(a.x-b.x) * (l.a.y-l.b.y)
               - (a.y-b.y) * (l.a.x-l.b.x);
        return (Math.abs(d) < EPS);
}}
class Circle {
    Point c;//Center
    double r;//Radius
    Circle(Point C, double R){ c=C; r=R; }
}
```

4.1.1 Common Point/Vector operations

- `angle(Point a, Point o, Point b)` returns the angle aob in radians. Note that this is always the smallest angle and hence the angle is at most π.
- `ccwAngle(Point a, Point o, Point b)` returns the counterclockwise angle aob in radians. Hence, this angle isn't always the smallest and lies in the range $[0 \dots 2\pi]$.
- `cross(Point p, Point q, Point r)` returns > 0 if smallest angle qpr is counterclockwise (r left of line pq). Returns < 0 if clockwise (r right of line pq). Returns 0 if angle 0 (r collinear to line pq / r lies on (extension) of line pq).
- The function `rotate(Point p, double angle)` rotates the `Point` counter clockwise (with respect to the center (0,0)) and assumes `angle` is in degrees. If in radians, remove the line with `toRadians()`.

```
double angle(Point a, Point o, Point b) {
    Point oa = new Point(o, a);
    Point ob = new Point(o, b);
```

```
    double A = oa.x*ob.x + oa.y*ob.y;
    double B = oa.x*oa.x + oa.y*oa.y;
    double C = ob.x*ob.x + ob.y*ob.y;
    return Math.acos(A / Math.sqrt(B*C));
}
double ccwAngle(Point a, Point o, Point b)
     {
    double A = angle(a, o, b);
    if (cross(0, a, b) > 0) {
        A = (-A) + (2 * Math.PI);
}}
double cross(Point p, Point q, Point r) {
    Point pq = new Point(p, q);
    Point pr = new Point(p, r);
    return (pq.x * pr.y - pq.y * pr.x);
}
double dot(Point p, Point q) {
    return (q.x*p.x + q.y*p.y);
}
void rotate(Point p, double angle) {
    angle = Math.toRadians(angle);
    double xNew = p.x*Math.cos(angle)
            - p.y*Math.sin(angle);
    double yNew = p.x*Math.sin(angle)
            + p.y*Math.cos(angle);
    p.x = xNew; p.y = yNew;
}
void scale(Point p, double factor) {
    p.x *= factor; p.y *= factor;
}
void translate(Point p,double X,double Y){
        p.x += X; p.y += Y;
}
```

4.2 Distance

For the distance between two linesegments $l1$ and $l2$, take: $min(dist(l1.a, l2), dist(l1.b, l2), dist(l2.a, l1), dist(l2.b, l1)$ For two polygons, take all possible pairs of linesegments.

```
double dist(Point p, Point q) {
    if (p.equals(q)) return 0.0;
    return Math.sqrt((q.x-p.x) * (q.x-p.x)
            + (q.y-p.y) * (q.y-p.y));
}
double dist(Point p, Line l) {
    Point ap = new Point(l.a, p);
    Point ab = new Point(l.a, l.b);
    double u = ap.x*ab.x + ap.y*ab.y;
    u /= (ab.x*ab.x + ab.y*ab.y);
    //Remove if's, if l not linesegment
    if (u < 0.0) return dist(p, l.a);
    if (u > 1.0) return dist(p, l.b);
    ab.x *= u; ab.y *= u;
    ab.x += l.a.x; ab.y += l.a.y;
    return dist(p, ab);
}
double dist(Point p, Circle c) {
    return Math.max(0, dist(p, c.c)-c.r);
}
double dist(Circle c, Line l) {
    return Math.max(0, dist(c.c, l)-c.r);
}
double dist(Circle c1, Circle c2) {
    return Math.max(0,
        dist(c1.c, c2.c) - c1.r - c2.r);
}
```

4.3 Intersection

4.3.1 Line - line intersection

```
Point intersection(Line l1, Line l2) {
    if (l1.isParallel(l2)) return null;
    Point a = l1.a, b = l1.b;
    Point c = l2.a, d = l2.b;
    double D = (a.x-b.x) * (c.y-d.y)
          - (a.y-b.y) * (c.x-d.x);
    double A = a.x*b.y - a.y*b.x;
    double B = c.x*d.y - c.y*d.x;
    double x = (A * (c.x - d.x)
          - (a.x - b.x) * B) / D;
    double y = (A * (c.y - d.y)
          - (a.y - b.y) * B) / D;
    //Remove if, if l1 not line segment
    if ((x+EPS)<Math.min(a.x,b.x)
            || (x-EPS)>Math.max(a.x,b.x)
            || (y+EPS)<Math.min(a.y,b.y)
            || (y-EPS)>Math.max(a.y,b.y)){
        return null;//No intersection
    }
    //Remove if, if l2 not line segment
    if ((x+EPS)<Math.min(c.x,d.x)
            || (x-EPS)>Math.max(c.x,d.x)
            || (y+EPS)<Math.min(c.y,d.y)
            || (y-EPS)>Math.max(c.y,d.y)){
        return null;//No intersection
    }
    return new Point(x, y);
}
```

4.4 Polygon

A polygon is often listed as a `List<Point> P`, where each `Point` appears exactly once and if you visit the `Points` in order, you traverse the perimeter of the Polygon.

I.e. Line(n-1, 0) $\cup(\bigcup_{i=0}^{n-2}$(Line(i, i+1)) = border of Polygon.

4.4.1 Area

Pre: List of `Points P` describing the perimeter of a simple Polygon, every `Point` occurring once.
Out: Area of the Polygon.
RT: O(`P.size()`)

```
P.add(P.get(0));//1st point added to end
double area = 0.0;
for (int i = 0; i < P.size()-1; i++) {
    Point p = P.get(i), q = P.get(i+1);
    area += p.x*q.y - q.x*p.y;
}
area = Math.abs(area) * 0.5;
return area;
```

4.4.2 Point inside/outside Polygon

Note: uses operations from section 4.1.1.
Pre: List of `Points P` describing the perimeter of a simple Polygon, every `Point` occurring once and a `Point q`.
Out: 1 if it is inside the polygon counterclockwise, -1 if inside clockwise, 0 if outside and *undefined* if on the boundary.

```
int j = P.size()-1, c = 0;
for (int i = 0; i < P.size(); j = i++) {
    Point a = P.get(i), b = P.get(j);
    if (b.y <= q.y && q.y < a.y
            && cross(q, b, a) > 0) {
        c++;
    } if (a.y <= q.y && q.y < b.y
```

```
            && cross(q, b, a) < 0) {
        c--;
}}
return c;
```

4.4.3 Check If Convex

Note: uses operations from section 4.1.1. Assumes that there are no points on border are collinear to lines on it (i.e. `cross(p, q, r)= 0` *implies* `q` *can be removed.*

Pre: List of `Points` `P` describing the perimeter of a simple Polygon, every `Point` occurring once.
Out: `true` if the polygon is convex, `false` otherwise.
RT: O(`P.size()`)

```
if (P.size() < 3) return false;
P.add(P.get(0)); P.add(P.get(1));
boolean CCW = cross(P.get(0), P.get(1),
      P.get(2)) > 0;
for(int i = 1, i < P.size()-2; i++) {
    if (CCW ^ cross(P.get(i), P.get(i+1),
          P.get(i+2)) > 0) {
        return false;
}}
return true;
```

4.4.4 Centroid (Zwaartepunt)

Pre: List of `Points` `P` describing the perimeter of a simple Polygon, every `Point` occurring once.
Out: The centroid (zwaartepunt) of the Polygon.
RT: O(`P.size()`)

```
P.add(P.get(0));//1st point added to end
double cX = 0.0, cY = 0.0, area = 0.0;
for (int i = 0; i < P.size()-1; i++) {
    Point p = P.get(i), q = P.get(i+1);
    area += p.x*q.y - q.x*p.y;
    cX += (p.x+q.x) * (p.y*q.x - p.x*q.y);
    cY += (p.y+q.y) * (p.y*q.x - p.x*q.y);
}
area = Math.abs(area) * 3;
cX /= area;
cY /= area;
return new Point(cX, cY);
```

4.4.5 Graham Scan (2D Convex Hull)

Note: uses operations from section 4.1.1

Pre: List of `Points` `P` where every point appears exactly once.
Out: List of `Points`, sorted, containing the `Points` in the Convex Hull.
RT: $O(n \log(n))$

```
if (P.size() <= 3) return P;
int low = 0;
for (int i = 1; i < P.size(); i++) {
    if (P.get(i).y < P.get(low).y) low=i;
}
Collections.swap(P, 0, low);
Point base = new Point(
        P.get(0).x + 1, P.get(0).y);
P.get(0).angle = 0;
for (int i = 1; i < P.size(); i++) {
    P.get(0).angle = ccwAngle(
            base, P.get(0), P.get(i));
}
Collections.sort(P);//Sort on angle
ArrayList<Point> R = new ArrayList();
R.add(P.get(0)); R.add(P.get(1));
int j = 1;
for (int i = 2; i < P.size(); i++) {
    while (cross(R.get(j-1), R.get(j),
            R.get(i) < 0) < 0) {
        R.remove(j--);
    }
    R.add(P.get(i));
    j = i;
}
return R;
```

5 Graphs

5.1 Graph Template

```
class Node {
    boolean visited;
    int dist = Integer.MAX_VALUE/2;
    ArrayList<Edge> adj = new ArrayList();
}
class Edge {
    int target,weight;
    Edge (int t, int w){ ... }
} //Read input/ create graph:
Node[] V = new Node[N];
for (int i = 0; i < N; i++) {
    V[i] = new Node();
} for (int i = 0; i < E; i++) {
    int a, b, w;//From input
    //Watch out which value is which
    //From a to b, having weight w
    V[a].adj.add(new Edge(b,w));
    //In case of undirected graph:
    V[b].adj.add(new Edge(a,w));
}
```

5.2 Depth-first search (DFS)

Note: this can be used for Flood Fill too, every v with `v.visited == true` *can be considered 'filled'.*

Out: Visits all $v \in V$ for which $\exists$ path(source, v), such that `v.visited == true`.
RT: $O(V + E)$

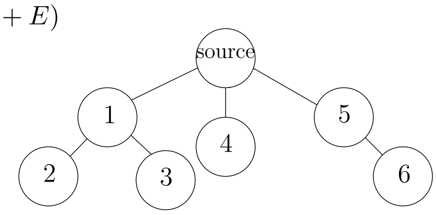

Figure 1: Example of the visit order in a DFS.

5.2.1 Graph-based DFS

Pre: A graph $G = (V, E)$ and a source $S \in V$

```
void DFS(int S) {
    V[S].visited = true;
    for (Edge edge : V[S].adj) {
        if (!(V[S].visited ||
                V[S].obstacle)) {
            DFS(edge.target);
}}}
```

5.2.2 Grid-based DFS

Pre: A 2D array of nodes and a point (x, y) in the array.

```
int[] dx ={-1,0,1,0};//Can be extended for
int[] dy ={0,1,0,-1};//diagonal adjacency
```

```
Node[][] V;
void DFS(int x, int y) {
    V[x][y].visited = true;
    for (int i = 0; i < 4; i++) {
        if ((x+dx[i],y+dy[i]).inBounds()&&
                !(V[x][y].visited ||
                 V[x][y].obstacle)){
            DFS(x+dx[i],y+dy[i]);
}}}//Use i<8 to support diagonal adjacency
```

5.2.3 Connected Components

Note: for directed graphs, see Tarjan's (section 5.2.4).
Pre: An undirected graph $G = (V, E)$ and a source $S \in V$.
Out: C is the number of components, $\forall_v\ v \in V$, v.component is the 0-based component number of v.
RT: $\mathrm{O}(V + E)$

```
int C = 0; //Number of components
for(int i = 0; i < N; i++){
    if(!V[i].visited) DFS(i, C++);
}
void DFS (int S, int c) {
    V[S].visited = true;
    V[S].component = c;
    for (Edge e : V[S].adj) {
        if(!V[e.target].visited) {
            DFS(e.target,c);
}}}
```

5.2.4 Tarjan (Strongly Connected Comp.)

Note: if you construct a graph by taking SCC's as nodes and insert the edges between distinct SCC's, the resulting graph is a DAG
Pre: A directed graph $G = (V, E)$
Out: For every node $u \in V$, `V[u].c` is the Strongly Connected Component(SCC) that node is currently in. Two nodes $u, v \in V$ are in the same SCC if and only if u can reach v and v can reach u.
RT: $\mathrm{O}(V + E)$

```
class Node { int i=-1, l, c=-1; ...}
int index = 0, C = 0;
ArrayDeque<Integer> stack =
        new ArrayDeque<Integer>();
for (int i = 0; i < N; i++) {
    if (V[i].i < 0) tarjan(i);
}
void tarjan(int i) {
    V[i].i = index++;
    V[i].l = V[i].i;
    stack.push(i);
    for (Edge e: V[i].adj) {
        if (V[e.target].i == -1) {
            tarjan(e.target);
            V[i].l = Math.min(
                V[i].l, V[e.target].l);
        } else if (V[e.target].c == -1) {
            V[i].low = Math.min(
                V[i].l, V[e.target].i);
    }}
    if (V[i].i == V[i].l) {//SCC found
        int j;
        do {
            j = stack.pop();
            V[j].c = C;
        } while(j != i);
        C++;
}}
```

5.2.5 Topological sort (DAG)

Note: if after this $\exists u \in V$: `V[u].in != -1`*, then* G *is cyclic.*
Pre: A DAG(= directed, acyclic) $G = (V, E)$
Out: `ArrayList` of nodes, sorted in topological order.
RT: $\mathrm{O}(V + E)$

```
class Node{int in; ...}//Nr incoming edges
ArrayList<Node> order = new ArrayList();
for(int i = 0; i < N; i++) {
    if (V[i].in == 0) DFS(i);
}
return order;
void DFS(int S) {
    V[S].in = -1;
    order.add(S);
    for (Edge edge : V[S].adj) {
        if (--V[edge.target].in == 0) {
            DFS(edge.target);
}}}
```

5.3 Breadth-first search (BFS)

Pre: A graph $G = (V, E)$ and a source $S \in V$ with $N = \#V$
Out: Visits all $v \in V$ for which $\exists$ path(source, v), such that `v.visited == true` and `v.dist` contains the distance from source to v.
RT: $\mathrm{O}(V + E)$

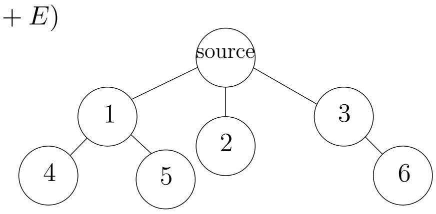

Figure 2: Example of the visit order in a BFS.

```
class Node { int dist = -1; ... }
ArrayDeque<Integer> Q = new ArrayDeque(N);
V[S].dist = 0;
Q.add(S);
while(!Q.isEmpty()) {
    int u = Q.poll();
    for(Edge e : V[u].adj) {
        if (V[e.target].dist>=0) continue;
        V[e.target].dist = V[u].dist + 1;
        Q.add(e.target);
}}
```

5.4 Shortest path

Note: if all edges have equal weights, you can use a BFS. For each $v \in V$*, multiply* `v.dist` *with the weight to get the distance of* v*.*

In case of a DAG, topological sort (section 5.2.5) first, then visit nodes in order and relax all outgoing edges.

5.4.1 Dijkstra (single source, positive weight)

Pre: A weighted graph $G = (V, E)$, where all edges have weight ≥ 0 and a source $S \in V$.
Out: For each $v \in V$, `v.dist` is the length of the shortest path from source to v and `v.parent` is the previous node on that path.
RT: $\mathrm{O}((V + E) \cdot \log(V))$

```
class Node {int parent; ... }
class ND implements Comparable<ND> {
    int index, dist;
    ND(int i, int t) { ... }
    public int compareTo(ND other) {
```

```
        return (dist - other.dist);
}} //ND = NodeDistance
PriorityQueue<ND> Q = new PriorityQueue();
V[S].dist = 0;
Q.add(new ND(S, 0));
while(!Q.isEmpty()) {
    ND nd = Q.poll();
    int u = nd.index;
    int d = nd.dist;
    if (V[u].dist < d) continue;
    for (Edge edge : V[u].adj) {
        int newD = d + edge.weight;
        int v = edge.target;
        if (newD < V[v].dist) {
            V[v].dist = newD;
            V[v].parent = u;
            Q.add(new ND(v, newD));
}}}//Nodes now contain distance + parent
```

5.4.2 Bellman-Ford (negative weight)

Pre: A weighted graph $G = (V, E)$, where $N = \#V$ and a source $S \in V$. Some edges may have negative weight.
Out: For each $v \in V$, `v.dist` is the length of the shortest path from source to v and `v.parent` is the previous node on that path.
RT: $\mathrm{O}(V \cdot E)$

```
V[S].dist = 0;
for (int k = 1; k < N; k++) {
   for (int i = 0; i < N; i++) {
      for (Edge e: V[i].adj) {
         int newD=V[i].dist+e.weight;
         if (newD < V[e.target].dist){
            V[e.target].dist = newD;
            V[e.target].parent = i;
}}}}
```

Negative weight cycles

If the sum of the weights of all edges in a cycle is negative, you have a negative weight cycle. Computing the shortest path in that case is impossible. To detect if such a cycle is reachable from node source, you can do the following in $\mathrm{O}(V + E)$ time.

```
BellmandFord(source);
for (int i = 0; i < N; i++) {
   for (Edge e: V[i].adj) {
      int newD=V[i].dist+e.weight;
      if (newD<V[e.target].dist){
         return true; //Neg weight cycle
}}}
return false; //No negative cycle found
```

5.4.3 Floyd-Warshall (all pairs shortest path)

Pre: A weighted graph $G = (V, E)$ where $N = \#V$.
Out: $\forall u, v \in V$: `dist[u][v]` is the distance from u to v.
RT: $\mathrm{O}(V^3)$

```
int[][] dist = new int[N][N];
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        dist[i][j] = Integer.MAX_VALUE/2;
        if (i == j) dist[i][j] = 0;
}}
for (int u = 0; u < N; u++) {
    for (Edge e : nodes[u].adj) {
        dist[u][e.target] = e.weight;
}}//Assume: only 1 edge from u to e.target
for (int k = 0; k < N; k++) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            dist[i][j] = Math.min(
                dist[i][j], dist[i][k]
                + dist[k][j]);
}}}
```

5.5 Prim (Minimum Spanning Tree/MST)

Pre: A weighted undirected graph $G = (V, E)$.
Out: A list of NW's, which describe the source, target and weight of the edges in the MST. `L2` contains only those edges, `L` contains additional edges which are $\notin E$.
RT: $\mathrm{O}(E \cdot \log(V))$

```
class Node { boolean inTree = false; ...}
class NW implements Comparable<NW> {
   int i,w,p;//node index, weight, parent
   NW(int I, int W, int P) {...};
   public int compareTo(NW other) {
      return (w - other.w);
}}
ArrayList<NW> L = new ArrayList();
ArrayList<NW> L2 = new ArrayList();
PriorityQueue<NW> Q = new PriorityQueue();
for (int i = 0; i < N; i++) {
   Q.add(new NW(
      i, Integer.MAX_VALUE / 2, -1));
}
while (!Q.isEmpty()) {
   NW nw = Q.poll();
   int k = nw.i;
   if (V[k].inTree) continue;
   V[k].inTree = true;
   L.add(nw);
   for (Edge e: V[k].adj) {
      if (V[e.target].inTree) continue;
      Q.add(new NW(e.target,e.weight,k));
}}
for(int i = 0; i < L.size(); i++) {
    if(L.get(i).p >= 0) L2.add(L.get(i));
}
```

5.6 Computing Euler Tour/Path

Pre: A graph $G = (V, E)$ for which an Euler tour/path is possible and a starting vertex `start`. If G undirected, `start` can be arbitrary, else `start` needs to have odd number of outgoing edges.
Out: A `L` containing a possible order to visit each edge exactly once. If tour, then start = end. If path, start $\neq$ end. If G doesn't admit an Euler tour/cycle, weird output.
RT: $\mathrm{O}(V + E)$

```
class Edge {boolean used; Edge back;...}
for (int i = 0; i < E; i++) {
    int a, b;//from input
    Edge e = new Edge(b);
    Edge f = new Edge(a);//If G undirected
    e.back=f; f.back=e;//Else f&e.back not
    a.adj.add(e); b.adj.add(f);//needed
}
LinkedList<Integer> L = new LinkedList();
L.add(start);
euler(start, L.listIterator(1));
return L;
void euler(int i,ListIterator<Integer> C){
    for(Edge e : V[i].adj) {
        if (e.used) continue;
```

```
            e.used=true;
            e.back.used=true;//If G undirected
            C.add(e.target);
            euler(e.target, cur);
        }
        C.previous();
}
```

5.7 Maximum Flow

The `augment()` *method is supplied by either Ford Fulkerson (5.7.1), Edmonds Karp (5.7.2) or Min Cost Flow (5.7.3)*

Pre: A flow network $G = (V, E)$ with a source $s \in V$ and a sink $t \in V$ such that `nodes[s]` and `nodes[t]` are valid nodes, $N = \#V$

Out: The maximum flow f^* of G.

```
class Node {
    Edge parent; int flow;
}
class Edge{
    int capacity, flow = 0; Edge back;
}
void main() {
    Edge e = new Edge(b, capacity);
    Edge f = new Edge(a,0);//0 if directed
                            //else capacity
    e.back = f; f.back = e;
    V[a].adj.add(e);
    V[b].adj.add(f);//also if directed
}
int totalFlow = 0;
while (true) {//Find an augmenting path
    for (int i = 0; i < N; i++) {
        V[i].visited = false;
        V[i].parent = null;
    }
    flow = augment(s, t);
    if (flow == 0) break;
    totalFlow += flow;
    int x = t;
    while (x != s) {//update flow on path
        V[x].parent.flow -= flow;
        V[x].parent.back.flow += flow;
        x = nodes[x].parent.target;
}}
return totalFlow;
```

5.7.1 Ford Fulkerson (DFS)

RT: $O(E \cdot f^*)$ ($f^* =$ max flow)

```
int augment(int i, int t) {
    if (V[i].visited) return 0;
    V[i].visited = true;
    if (i == t) return Integer.MAX_VALUE;
    for (Edge e : V[i].adj) {
        if(e.capacity-e.flow<=0) continue;
        int f = augment(e.target, t);
        if (f > 0) {//e.target in path
            f = Math.min(f,
                e.capacity - e.flow);
            V[e.target].parent=e.back;
            return f;
    }}
    return 0; // no flow to t found
}
```

5.7.2 Edmonds Karp (BFS)

RT: $O(V \cdot E^2)$

```
int augment(int s, int t) {
    ArrayList<Integer> Q=new ArrayList();
    V[s].visited = true;
    V[s].flow = Integer.MAX_VALUE / 2;
    V[t].flow = 0;
    Q.add(s);
    for (int i = 0; i < Q.size(); i++) {
        int u = Q.get(i);
        if (u == t) break;//
        for (Edge e: V[u].adj) {
            if (e.capacity - e.flow <= 0
                ||V[e.target].visited)
                    continue;
            V[e.target].flow =
                Math.min(V[u].flow,
                    e.capacity - e.flow);
            V[e.target].visited=true;
            V[e.target].parent=e.back;
            Q.add(e.target);
    }}
    return V[t].flow;
}
```

5.7.3 Minimum Cost Maximum Flow

Uses code from section 5.7

Out: computes the maximum flow, but will pick the edges in such a way that the total cost is minimized.

RT: $O(V^2 \cdot E^2)$

```
void main() {
    Edge e = new Edge(b, capacity);
    Edge f = new Edge(a,0);
    e.cost = cost(a, b); f.cost = -e.cost;
    e.back = f; f.back = e;
    V[a].adj.add(e);
    V[b].adj.add(f);
}
int augment(int s, int t) {
    for (Node u : V) {
        u.cost = Integer.MAX_VALUE / 2;
    }
    V[s].flow = Integer.MAX_VALUE / 2;
    V[t].flow = 0;
    V[s].cost = 0.0;
    for (int k = 1; k < V.length; k++) {
        for (Node u : V) {
            for (Edge e : u.adj) {
                if (e.capacity-e.flow<=0){
                    continue;
                }
                int newC= u.cost+e.weight;
                Node v = V[e.target];
                if (newC < v.cost) {
                    v.parent = e.back;
                    v.cost = newC;
    }}}}
    int x = t;
    int flow = Integer.MAX_VALUE / 2;
    while (x != s) { //Find flow
        Edge e = V[x].parent;
        if (e == null) return 0;
        flow = Math.min(e.back.capacity
                - e.back.flow, flow);
        x = e.target;
    }//update flow
    x = t;
    while (x != s) {
        V[x].flow += flow;
        x = V[x].parent.target;
```

```
    }
    return flow;
}
```

5.8 Bipartite Matching

Pre: Bipartite graph, represented by 2 arrays of nodes; A of length p and B of length q. Edges only from A to B, all with weight/capacity 1.
Out: M, where M is the maximum size set $S \subseteq V$ such that $\forall x \in S : deg(x) = 1$. I.e. create bijective function $F : A \to B$. Also, if $B[i].match = j$, then node $j \in A$ maps to $i \in B$.
RT: $O(V \cdot E)$

```
class Node {
    int match=-1; boolean visited=false;
}
Node[] A,B;
int M = 0; // matching size/solution
for (int i = 0; i < p; i++) {
    for (Node u : A) {
        u.visited = false;
    }
    if (augment(i)) M++;
}
return M;
boolean augment(int i) {
    if (A[i].visited) return false;
    A[i].visited = true;
    for (Integer j: A[i].adj) {
        B[j].visited = true;
        if (B[j].match == -1 ||
                augment(B[j].match)) {
            B[j].match = i;
            return true;
    }}
    return false;
}
```

For the **Minimum Vertex Cover** problem, see section 8.5

Maximum Independent Set
Problem: Maximum size set $S \subseteq V$ such that there does not exist an edge between nodes in S.
Solution: $p + q - M$

5.9 Useful theorems/lemmas/tricks

1. A tree has $V - 1 = E$
2. A flow network G_f with only integral capacities has integral max flow + all flows for the edges are integral (if you use Ford-Fulkerson/Edmonds Karp)
3. Take a flow network G_f with a cut (S, T), where source $\in S$, sink $\in T$, $V = S \cup T$, $S \cap T = \emptyset$. Then a cut makes it impossible to go from source to sink. The value of a cut is the sum of the capacities of edges from S to T(reverse not included). Minimum value of that cut = Max Flow.
4. To check if a graph is a DAG, topological sort it and see if there is no $u \in V$ with $V[u].in \geq 0$. If that is true, then the graph is DAG.
5. To check if a graph is bipartite, start naming vertices either 1 or 2, such that no two vertices connected by an edge have the same name. If this is possible, then the graph is bipartite.
6. To check if a graph admits an Euler tour (start = end), check if every vertex has an even number of edges connected.
7. To check if a graph admits an Euler path (start $\neq$ end), check if every vertex has an even number of edges connected, except for two vertices u, v, who needs uneven number of edges. Case directed: u needs one more outgoing edge then there are incoming, v one less. Then, u is start, v is end.

6 Dynamic Programming (DP)

6.1 Knapsack Problem

Pre: $n, m \in \mathbb{N} \wedge n, m \geq 1$ and an array `items[]` of size n, containing for each `Item i` in `items`, `i.value` and `i.weight`
Out: Max value of picking n items with total weight $\leq m$
RT: $O(m \cdot n)$

```
int[][] dp = new int[n+1][m+1];
for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= m; j++) {
        dp[i][j] = dp[i - 1][j];
        int val = items[i - 1].value;
        int W = items[i - 1].weight;
        if (W <= j) {
            dp[i][j] = Math.max(dp[i][j],
                dp[i - 1][j - W] + val);
}}}
System.out.println(dp[n][m]);
```

6.2 Longest Common Subsequence

Pre: Char/int/double array A[] of length P and Char/int/double array B[] of length Q.
Out: Length of longest common subsequence (= sequence that can be obtained by omitting 0 or more characters) of A and B. Get retrieves an ArrayList with the actual sequence. If `L = get(P,Q)`, then `L.get(0)` is first element of the LCS and `L.get(dp[P][Q]-1)` is the last element.
RT: $O(P \cdot Q)$ for determining length, $O(P + Q)$ for `get(P,Q)`

```
int[][] dp = new int[P+1][Q+1];
for (int i = 1; i <= P; i ++) {
    for (int j = 1; j <= Q; j ++) {
        if (A[i-1]==B[j-1]) {
            dp[i][j] = dp[i-1][j-1] + 1;
        } else {
            dp[i][j] =Math.max(dp[i][j-1],
                    dp[i-1][j]);
}}}
return dp[P][Q]; //Length of best solution
get(P,Q); //Retrieves actual sequence
ArrayList<Integer> get(int i, int j) {
    if (i == 0 || j == 0) {
        return new ArrayList<Integer>();
    } else if (A[i-1] == B[j-1]) {
        ArrayList<Integer> L=get(i-1,j-1);
        L.add(A[i-1]);
        return L;
    } else if (dp[i][j-1] > dp[i-1][j]) {
        return get(i,j-1);
    } else {
        return get(i-1,j);
}}
```

6.3 Edit Distance (Difference of Strings)

Pre: Char array A[] of length M and Char array B[] of length N. `iC(x)` is a function that calculates the cost of inserting `x`. `dC(x)` is delete cost for `x`. `sC(x,y)` is the cost for changing `x` into `y`.

Out: Minimum cost needed to change A into B.
RT: $O(M \cdot N)$.

```
int[][] dp = new int[M+1][N+1];
for (int i = 1; i <= M; i++) {
   dp[i][0] = dp[i - 1][0] + dC(A[i-1]);
} for (int i = 1; i <= N; i++) {
   dp[0][i] = dp[0][i - 1] + iC(B[i-1]);
} for (int i = 1; i <= M; i ++) {
   for (int j = 1; j <= N; j ++) {
       if (A[i-1]==B[j-1]) {
           dp[i][j] = dp[i-1][j-1];
       } else {
           dp[i][j] = Math.min(dp[i-1][j-1]
              + sC(A[i-1], B[j-1]),
              dp[i-1][j] + dC(A[i-1]));
           dp[i][j] = Math.min(dp[i][j],
              dp[i][j-1] + iC(B[j-1]));
}}}
return dp[M][N]; //Length of best solution
```

6.4 Coin Counting

Pre: $n \in \mathbb{N}$ and an array `coins[]` specifying for each coin `i` its value `coins[i]`, where `coins[i] > 0` and each value in `coins[]` is unique.
Out: the number of distinct ways to create the value n using only the coins in `coins[]`.
RT: $O(n^2)$

```
long[] dp = new long[n + 1];
dp[0] = 1L;
for (int i = 0; i < coins.length; i++) {
    for (int j = coins[i]; j<=n; j++){
        dp[j] += dp[j - coins[i]];
}}
dp[0] = 0L;//Problem dependant
return dp;
```

6.5 Coin Change

Pre: $n \in \mathbb{N}$ and an array `coins[]` specifying for each coin `i` its value `coins[i]`, where `coins[i] > 0` and each value in `coins[]` is unique.
Out: the minimum number of coins needed to create `n` from `coins`, where `R` contains the actual coins used to form `n`.
RT: $O(n^2)$

```
int[] dp = new int[n+1];//answer at dp[n]
int[] T = new int[n + 1];
ArrayList<Integer> R = new ArrayList();
for (int i = 1; i <= n; i++) {
    dp[i] = Integer.MAX_VALUE / 2;
    for (int j : coins) {
        if(j<=i && (dp[i-j]+1)< dp[i]) {
            dp[i] = dp[i-j] + 1; T[i]=j;
}}}
return dp[n];//Number of coins needed
while(n > 0) {
    R.add(T[n]); n -= dp[n];
}
return R;//List of coins used
```

6.6 Subset DP

Note: this makes use of bitmasks (section 2.6)
This is a technique for solving problems. Use it when it looks like a DP problem, but the solution depends on which subset of things you already have. The number of elements in the set should be small (≤ 16). Also consider applying memoization, especially for multiple dimensions.

7 Miscellaneous Stuff

7.1 String Matching (KMP)

Pre: `char[] pattern` of size m and `char[] text` of size n.
Out: All indexes where pattern occurs in text.
RT: $O(n + m)$

```
int[] PM = new int[m];//Parial Match Table
void kmpPreProcess() {
    int i = 0, j = -1;
    PM[0] = -1;
    while (i < m) {
        while (j >= 0 && pattern[i] !=
            pattern[j]) {
            j = PM[j];
        }
        i++; j++;
        PM[i] = j;
}}
void kmpSearch() {
    int i = 0, j = 0;
    while (i < n) {
        while (j >= 0 && text[i] !=
            pattern[j]) {
            j = PM[j];
        }
        i++; j++;
        if (j == m) {//Match at index i-j
            //Last char of pattern at i-1
            solution.add(i-j);
            j =PM[j];
}}}
```

7.2 Longest Increasing Subsequence

Pre: An array of integers `A` with `n = A.length`;
Out: `ans` contains the elements of the longest *strictly* increasing subsequence, where `ans.size()= length + 1`.
RT: $O(n \log(n))$
Example: `A = {3,4,-1,5,8,2,3,12,7,9,10,10};`. Then, `n = 12;` and a possible LIS = `{-1,2,3,7,9,10};` Note that there does not exist a longer one.

```
int[] T = new int[n], R = new int[n];
Arrays.fill(R, -1);
int length = 0;
ArrayList<Integer> ans = new ArrayList();
for (int i=1; i < n; i++) {
    if (A[T[0]] > A[i]) {
        T[0] = i;
    } else if(A[T[length]] < A[i]) {
        T[++length] = i;
        R[T[length]] = T[length - 1];
    } else {
        int j = search(length, A[i]);
        if (j < 0) continue;
        T[j] = i;
        R[T[j]] = T[j - 1];
}}
int i = T[length];
while(i >= 0) {
    ans.add(A[i]); i = R[i];
}
Collections.reverse(ans);
return ans;
int search (int E, int v){
    int S = 0, L = E, M;
```

```
    while(S <= E){
        M = (S + E) / 2;
        if(M<L&&A[T[M]]<v&&v<=A[T[M+1]]){
            return M + 1;
        } else if(A[T[M]] < v){
            S = M + 1;
        } else {
            E = M-1;
    }}
    return -1;
}
```

7.3 Impartial Game Theory

Identifiers of an impartial game:

1. Two player game where moves are (usually) alternated
2. No simultaneous moves are allowed
3. For every state specified which moves are legal
4. Game ends when no move is possible in a turn
   4.1. Normal play rule: last player to move wins
   4.2. Misère play rule: last player to move loses (hard)
5. No draws are allowed
6. Game always has a finite number of moves
7. Both players have the same set of moves available. So, legal set of moves only depends current state, not which of the two players is moving

Labeling states with P/N:
First create a graph where nodes represent legal states and edges represent legal moves. Label states with P(revious) if it secures a win for the player who has just moved. Label states with N(ext) if it secures a win for the player about to make a move. Generally, final states are P (normal play rule). If dealing with objects on a pile, state with 0 items on pile is final state.

For nodes which have not been determined: label with P if all next nodes are N. If at least one moves leads to a P node, label it N. If initial state is N, first player to moves can win, else the other player. Strategy is to make moves leading to P nodes.

7.3.1 Sprague-Grundy Function

The **mex** of a set is the smallest value (≥ 0) not contained in the set. Some examples:

- $mex(\emptyset) = 0$ (often the final state)
- $mex(\{1, 2, 3\}) = 0$
- $mex(\{0, 2, 4, 6, ...\}) = 1$
- $mex(\{0, 1, 2, 4, 5\}) = 3$

Function: $G(v) = mex(\{G(w) \text{ if } (v, w) \in E\})$. Generally, if $G(v) = 0$, the state is P. If bigger, then the state is N.

In case of multiple games forming one game, for every subgame, calculate its $G(v)$ value and take the XOR of all those values.

8 Rare Problems and Solutions

8.1 Traveling Salesman Problem (TSP)

Problem: Given a set of n cities $0, \ldots, n-1$ and for every pair of cities (i, j) the distance between them, `dist(i, j)`, what is the shortest cycle visiting every city exactly once?
Solution: $O(n^2 \cdot 2^n)$ Subset DP. Create a 2D array `int[n][(1<<n)] dp`. The solution will be in `dp[0][1]`.
`dp[i][j] = dist(i,0)` if `i == (1<<n)`
`dp[i][j] = min{dist(i,nxt)+dp[nxt][j|(1<<nxt)]}`
$\forall_{nxt}\ 0 \leq \texttt{nxt} < n \wedge \texttt{nxt} \neq \texttt{j} \wedge (\texttt{j \& (1 << nxt)}) = 0$

8.2 Bitonic TSP

Problem: Given a set of n points on a 2D plane. What is the shortest cycle starting at the leftmost point, moving onl to the right and when at the rightmost point moving only to the left.
Solution: $O(n^2)$ Dynamic Programming. Sort all points by x-coordinate (leftmost point with index 0, rightmost point with index $n-1$). Create a 2D array `int[n-1][n-1] dp` and solve it using the recurrence below. Finally, compute the solution as follows:
$dp[n-1][n-1] = min_{0 \leq k < n-1}\{dp[n-1, k] + dist(k, n-1)\}$

$$dp[i,j] = \begin{cases} dist(0,1) & \text{if } i = 1 \wedge j = 0 \\ dp[i-1,j] + dist(i-1,i) & \text{if } (i-1) > j \\ min_{0 \leq k < j}\{dp[j][k] + dist(k,i)\} & \text{if } (i-1) = j \end{cases}$$

8.3 2-Satisfiability (2SAT)

Problem: Given a boolean formula of the following form: $F = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee x_4)$, is there a true/false assignment to the variables such that F yields true.
Solution: Create a directed graph. For each variable, create two nodes, a and $\neg a$. Then, for every $(a \vee b)$, add an edge from $\neg a$ to b and from $\neg b$ to a. Then, run Tarjan's algorithm (see section 5.2.4). For every SCC, check for every variable if a and $\neg a$ are present. If so, it is not possible.

To compute the actual assignment, turn each SCC into one node. New graph is a DAG. Topologically sort the nodes (see section 5.2.5). Visit SCC's in reverse topological order. Set all variables in last SCC to true. In all complementary SCC's (which contains a, whilst $\neg a$ has been determined), turn every variable to the complementary value.

8.4 Bracket Matching

Problem: Given a set of brackets: (){[}])(, does it follow a standard bracket structure (e.g. ()([()]))
Solution: $O(n)$. Push every opening bracket on top of a `Stack<B>` (or `ArrayDeque`, section 2.1). For closing brackets, poll the topmost element from the stack and see if it matches. If all match, it is valid.

8.5 Minimum Vertex Cover

Problem: NP-hard. Given a graph G, what is the minimum number that you can select such that each edge is touched by at least one selected vertex. (Or: what is the minimum number of vertices to remove such that every edge gets removed).
Solution: $O(2^n)$ brute force. For every edge, recurse with two possible options: remove left node (and all connected edges) or remove right node (and all connected edges).

If the graph is bipartite, you can use Bipartite Matching (section 5.8) and return M.

8.6 Stable Marriage

Problem: Given two disjoint sets of the same size, A and B, where each element of A has rated how much they want to be paired up with each element of B and vice versa, where no element has rated two other elements the same. What is the best possible matching such that each element from A is paired with an element from B and that it is not possible for one of the two to find another element such that both elements in the new pair are happier.
Solution: $O(n^2)$. For each element in A, let them pick the element they have ranked highest. Each element in B always accepts. If an element in B has multiple elements to

choose from, it chooses the one it ranked highest. If it has already picked an element, it can drop it for a new element which it ranked higher. Repeat until done.