Contents

1. Code Templates

1.1. KattIO.

```
class Kattio {
  private BufferedReader r;
  private String line;
  private StringTokenizer st;
  private String token;

  public Kattio(InputStream i) {
    r = new BufferedReader(new InputStreamReader(i));
  }

  public boolean hasMoreTokens() {
    return peekToken() != null;
  }
  public int getInt() {
    return Integer.parseInt(getWord());
  }
  public double getDouble() {
    return Double.parseDouble(getWord());
  }
  public long getLong() {
    return Long.parseLong(getWord());
  }
  public String getWord() {
    String ans = peekToken();
    token = null;
    return ans;
  }

  private String peekToken() {
    if (token == null)
      try {
        while (st == null || !st.hasMoreTokens()) {
          line = r.readLine();
          if (line == null) return null;
          st = new StringTokenizer(line);
        }
        token = st.nextToken();
      } catch (IOException e) { }
    return token;
  }
}
```

2. Math

2.1. Trigonometry. Common formulas for sin and cos.

$$
\begin{array}{rcl}
\tan x & = & \frac{\sin x}{\cos x} \\
\sin(-x) & = & -\sin x \\
\cos(-x) & = & \cos x \\
\sin(\pi/2-x) & = & \cos x \\
\cos(\pi/2-x) & = & \sin x \\
\sin(\pi-x) & = & \sin x \\
\cos(\pi-x) & = & -\cos x \\
\hline
\sin(\alpha+\beta) & = & \sin\alpha\cdot\cos\beta+\cos\alpha\cdot\sin\beta \\
\cos(\alpha+\beta) & = & \cos\alpha\cdot\cos\beta-\sin\alpha\cdot\sin\beta \\
\sin(\alpha-\beta) & = & \sin\alpha\cdot\cos\beta-\cos\alpha\cdot\sin\beta \\
\cos(\alpha-\beta) & = & \cos\alpha\cdot\cos\beta+\sin\alpha\cdot\sin\beta \\
\hline
\sin 2x & = & 2\cdot\sin x\cdot\cos x \\
\cos 2x & = & \cos^2 x-\sin^2 x \\
2\cdot\sin x\cdot\sin y & = & \cos(x-y)-\cos(x+y) \\
2\cdot\cos x\cdot\cos y & = & \cos(x-y)+\cos(x+y) \\
2\cdot\sin x\cdot\cos y & = & \sin(x-y)+\sin(x+y)
\end{array}
$$

Law of Cosines:
$c^2 = a^2 + b^2 - 2ab\cos\gamma$

Law of Sines:
$\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma}$

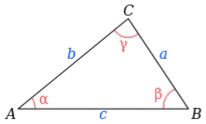

FIGURE 1. A triangle with three corners, used in the Laws of Cosines/Sines.

2.2. **Geometry.** *Given a triangle abc*, $u = a \to b, v = a \to c$

Cross product:
$u \times v = u_x v_y - u_y v_x$

Dot product:
$u \cdot v = u_x v_x + u_y v_y$

Orthogonal projection:
$u' = \frac{u \cdot v}{|v^2|} v$

Angle between vectors $[-\pi, \pi]$**:**
$\text{atan2}(u \times v, u \cdot v) =$
$\text{atan2}(c_y - a_y, c_x - a_x) - \text{atan2}(b_y - a_y, b_x - a_x)$

Triangle area:
$\frac{1}{2}(u \times v) = \frac{1}{2}((b-a) \times (c-a))$

Polygon area:
$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$
(where n is #vertices)

Polygon center:
$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$
$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$

Point inside polygon:
$S = \sum_{i=0}^{n-2} \text{angle}(p - v_i, p - v_{i+1}) + \text{angle}(p - v_{n-1}, p - v_0)$
if($S == \pm 2k\pi$): inside
if($S == 0$): outside
(where p is a point)

A faster way to calculate would be using raycasting and counting intersecting edges.

Formulate plane given normal:
Given a normal $n = (a, b, c)$ and a point on the plane $P = (x_0, y_0, z_0)$ we can formulate the plane as $ax + by + cz + d = 0$ where $d = -(ax_0 + by_0 + cz_0)$.

Line equation
$ax + by + c = 0 \Leftrightarrow y = -\frac{a}{b}x - \frac{c}{b}$

2D Line intersection:
Given two line equations: $y = a_1 x + b_1$, $y = a_2 x + b_2$
$x = \frac{b_2 - b_1}{a_1 - a_2}$ // if $a_1 = a_2$, the lines are parallel
$y = a_1 x + b_1 = a_2 x + b_2$

Point-Line distance (in plane):
Given a line and a point: $ax + by + c = 0, (x_0, y_0)$
$dist = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$
$x_{closest} = \frac{b(bx_0 - ay_0) - ac}{a^2 + b^2}$ and $y_{closest} = \frac{a(-bx_0 + ay_0) - bc}{a^2 + b^2}$

Point-Plane distance (in 3D space):
Given a plane and a point: $ax + by + cz + d = 0, (x_0, y_0, z_0)$
$dist = \frac{|ax_0 + by_0 + cz_0 + d|}{\sqrt{a^2 + b^2 + c^2}}$

Point-Line distance (in 3D space):
Given a line and a point: $l = \mathbf{u} + \mathbf{v}t, P$
> Find P_0, any point on the line.
> $\mathbf{u_0} = \overrightarrow{P_0 P}$
> $\mathbf{u_1} = \frac{\mathbf{u_0} \cdot \mathbf{v}}{|\mathbf{v}|^2} \mathbf{v}$ // *Projection of* $\mathbf{u_0}$ *onto* $\mathbf{v}$
> $\mathbf{u_2} = \mathbf{u_0} - \mathbf{u_1}$ // *Orthogonal vector*
$dist = |\mathbf{u_2}|$

Line-Line distance:
if the lines are parallel in 2D:
$d = \frac{|c_2 - c_1|}{\sqrt{a^2 + b^2}}$ *($ax + by + c = 0$)* or $d = \frac{|b_2 - b_1|}{\sqrt{a^2 + 1}}$ *($y = ax + b$)*

in 3D, given $l_1 = \mathbf{u_1} + \mathbf{v_1}t$ *and* $l_2 = \mathbf{u_2} + \mathbf{v_2}t$*:*
> $\mathbf{n} = \mathbf{v_1} \times \mathbf{v_2}$
$dist = \frac{\mathbf{n} \cdot (\mathbf{u_1} - \mathbf{u_2})}{||\mathbf{n}||}$

2.3. **Combinatorics.** Various useful combinatoric formulas. Formulas for the number of ways of taking k from n items:

	With repetitions	No repetitions
Order matters	n^k	$\frac{n!}{(n-k)!}$
Any order	$\binom{n+k-1}{k}$	$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$

Formulas progressions and sums of arithmetic and geometric sequences:

	Arithmetic	Geometric
Progression	$a_n = a_{n-1} + d = a_1 + d \cdot (n-1)$	$a_n = a_{n-1} \cdot r = a_1 \cdot r^{n-1}$
Sum	$S_n = \frac{n(a_1 + a_n)}{2}$	$S_n = \frac{a(r^n - 1)}{r - 1}$

The calculation of combinations and permutations can be implemented efficiently in $\mathcal{O}(n/2)$ and $\mathcal{O}(n)$ respectively. The following code has a high risk of overflow, consider using `BigInteger` for large numbers:

```
// Calculates #combinations (n over k)
long nCr(int n, int k) {
    if (n < k)
      return 0;

  if (k > n / 2)
    k = n - k;
  long ans = 1;
  for (int i = 1; i <= k; i++) {
    ans *= n - k + i;
    ans /= i;
  }
  return ans;
}

// Calculates #permutations
long nPr(int n, int k) {
    if (n < k)
      return 0;

  long ans = 1;
  for (int i = 1; i <= k; i++) {
    ans *= n - k + i;
  }
  return ans;
}
```

2.4. Number Theory. Various useful number theory formulas.

```
// Calculates the greatest common divisor of a and b
int gcd(int a, int b) {
  while (b > 0) {
    int t = b;
    b = a % b;
    a = t;
  }
  return a;
}

// Calculates the least common multiple of a and b
int lcm(int a, int b) {
  return a / gcd(a, b) * b;
}
```

2.5. Prime factorization. Time complexity is $\mathcal{O}(\sqrt{n})$ and space complexity is $\mathcal{O}(n)$.

```
# A fast way to factorize a number into primes.
def primes(N):
  factors = []
  i = 2
  while i**2 <=N:
    while N % i == 0:
      N //= i
      factors.append(i)
    i += 1

  if N != 1:
    factors.append(N)
  return factors
```

2.6. Systems of Equations. Time complexity is $\mathcal{O}(n^3)$ and space complexity is $\mathcal{O}(n)$. Uses Gaussian elimination with scaled partial pivoting for numerical stability. The for-loop with the scaling may be removed if precision is not a problem.

This only works for $N \times N$ matrices; if you have an $N \times M$ matrix (with N > M) you can solve it by first computing $A' = A^\top A$ and $b' = A^\top b$ and then running the algorithm. That would result in a least squares solution.

```
// Solves Ax = b by computing x = A^-1 * b
public class Gauss {
  private static final double THRESHOLD = 0.000001;

  // A: NxN and b: Nx1 => x: Nx1
  public double[] solve(double[][] A, double[] b) {
    int N = A.length;
    // Rescale (scaled pivoting), skip if not needed!
    for (int i = 0; i < N; i++) {
      double max = -Double.MAX_VALUE;
      for (int j = 0; j < N; j++) {
        max = Math.max(max, Math.abs(A[i][j]));
      }
      if (max < THRESHOLD)
        return null;  // Not full rank

      for (int j = 0; j < N; j++) {
        A[i][j] /= max;
      }
      b[i] /= max;
    }

    // Forward propagation
    for (int i = 0; i < N; i++) {
      // Find largest pivot
      int biggestIdx = i;
      for (int j = i; j < N; j++) {
        if (Math.abs(A[j][i]) >
            Math.abs(A[biggestIdx][i]))
          biggestIdx = j;
      }

      if (biggestIdx != i) { // Swap if necessary
        double[] tmps = A[biggestIdx];
        A[biggestIdx] = A[i];
        A[i] = tmps;
        double tmp = b[biggestIdx];
        b[biggestIdx] = b[i];
        b[i] = tmp;
      }

      double pivot = A[i][i];
      if (Math.abs(pivot) < THRESHOLD)
        return null;  // Not full rank

      for (int j = i+1; j < N; j++) {
        double mult = A[j][i]/pivot;
        for (int k = 0; k < N; k++) {
          A[j][k] -= mult * A[i][k];
        }
        b[j] -= mult * b[i];
      }
    }

    // Backwards substitution
    double[] X = new double[N];
    for (int i = N-1; i >= 0; i--) {
      for (int j = i+1; j < N; j++) {
        b[i] -= A[i][j]*X[j];
      }
      X[i] = b[i]/A[i][i];
    }

    return X;
  }
}
```

3. Algorithmic concepts

3.1. Inclusion-Exclusion principle. This principle may be useful for problems that you can model as k overlapping subsets over n values, where you are interested in finding the union of the k subsets.

An example of this may be ”Find the amount of numbers between 1 and 2^{30} that are divisible by neither 2, 3 nor 5”. Model this as three sets A, B and C representing numbers from $[1, 2^{30}]$ not divisible by 2, 3 and 5. Calculate the following:

$$\begin{aligned}|A \cup B \cup C| = &|A| + |B| + |C| - |A \cap B| - |A \cap C| \\ &- |B \cap C| + |A \cap B \cap C|\end{aligned}$$

This is visualized in Figure 2 below. Note that all intersections with an even amount of terms will be negative, even in the general case with k sets.

These types of problems are characterized by *huge output* (often modulo m) and *few subsets* k.

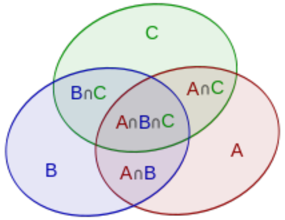

FIGURE 2. The three sets visualized, gives intuition about why we need to subtract even terms and add odd ones.

3.2. **Meet in the middle.** This method is useful for problems that are a little too large to be brute forced and have a structure that allows it to be split.

An example of this may be "Given an array of $n \in [1, 2^{36}]$ numbers find the maximum subset sum modulo m". Naively testing the sum of all subsets will not work (2^{36} is too large), instead split the array in two and find all possible sums in each half (only 2^{18} in each). When this is done we merge them in a smart way that is faster than $\mathcal{O}(n^2)$.

These types of problems are characterized by an *input size* just beyond the range of brute force and *easily partitioned* data.

4. Search & Sort

4.1. **Binary Search.** Time complexity is $\mathcal{O}(n \log n)$ and space complexity is $\mathcal{O}(1)$.

```java
public int search(int[] data, int target) {
  int l = 0;
  int r = data.length - 1;
  while (l < r) {
    int m = (l+r)/2;
    if (data[m] < target)
      l = m+1;
    else if (data[m] > target)
      r = m-1;
    else
      return m;
  }
  return -1;
}
```

4.2. **Sorting.** Time complexity is $\mathcal{O}(n \log n)$ for both algorithms and space complexity is $\mathcal{O}(\log n)$.

- `Collections.sort()` uses Merge Sort
- `Arrays.sort()` uses Quick Sort

4.3. **Quick Select.** Time complexity is $\mathcal{O}(n)$ on average and $\mathcal{O}(n^2)$ in the worst case. The space complexity is $\mathcal{O}(\log n)$.

```java
// Finds k'th smallest element in array[l..r]
public static int kthSmallest(int[] array,
    int low, int hi, int k) {
  if (k > 0 && k <= hi - low + 1) {
    int pos = partition(array, low, hi);
    if (pos - low == k - 1)
      return array[pos];
    if (pos - low > k - 1)
      return kthSmallest(array, low, pos - 1, k);
    return kthSmallest(array, pos+1, hi, k+low-pos-1);
  }
  return Integer.MAX_VALUE;
}

static void swap(int[] array, int i, int j) {
  int temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}

static int partition(int[] array, int low, int hi) {
  int n = hi - low + 1;
  int pivot = (int) (Math.random() * n);
  swap(array, low + pivot, hi);

  int x = array[hi], i = low;
  for (int j = low; j < hi; j++) {
    if (array[j] <= x) {
      swap(array, i, j);
      i++;
    }
  }
  swap(array, i, hi);
  return i;
}
```

4.4. **Knuth-Morris-Pratt Algorithm.** Time complexity is $\mathcal{O}(n)$ and space complexity is $\mathcal{O}(\log n)$. Good when the alphabet is small (around 4-5 characters).

```java
// Finds patterns in a text
private static class KMP {
  public static int match(String text, String pat) {
    int[] lps = new int[pat.length()];

    int len = 0;
    for (int i = 1; i < lps.length; i++) {
      if (pat.charAt(i) == pat.charAt(len)) {
        len++;
        lps[i] = len;
      } else if (len != 0) {
        len = lps[len-1];
        i--;
      } else {
        lps[i] = 0;
      }
    }

    int i = 0;
    int j = 0;
    while (i < text.length()) {
      if (pat.charAt(j) == text.charAt(i)) {
        i++;
        j++;
      }
      if (j == pat.length()) {
        return i-j;
        //j = lps[j-1]; //Uncomment to continue search
      }
      else if (i < text.length() &&
          pat.charAt(j) != text.charAt(i)) {
        if (j != 0)
          j = lps[j-1];
        else
          i++;
      }
    }
    return -1;
  }
}
```

4.5. Z-Array Algorithm. Time complexity is $\mathcal{O}(n)$ and space complexity is $\mathcal{O}(n)$. Good when the alphabet is large.

```
// Finds patterns in text, also constructs a z-array
private static class ZArray {
  static int search(String text, String pat) {
    // Note: replace $ if found in text or pat!
    String str = pat + "$" + text;
    int[] z = getZArray(str);

    for (int i = 1; i < z.length; i++) {
      if (z[i] == pat.length())
        return i-1-pat.length();// Return or collect
                                // when equal
    }
    return -1;
  }

  static int[] getZArray(String str) {
    int[] z = new int[str.length()];
    int low, hi, k;
    low = hi = 0;
    for (int i = 1; i < str.length(); i++) {
      if (i > hi) {
        low = hi = i;
        while (hi < str.length() &&
            str.charAt(hi-low) == str.charAt(hi))
          hi++;
        z[i] = hi-low;
        hi--;
      } else {
        k = i-low;
        if (z[k] <= hi-i)
          z[i] = z[k];
        else {
          low = i;
          while (hi < str.length() &&
              str.charAt(hi-low) == str.charAt(hi))
            hi++;
          z[i] = hi-low;
          hi--;
        }
      }
    }
    return z;
  }
}
```

5. Data Structures

5.1. Fenwick tree. Time complexity is $\mathcal{O}(\log n)$ for all operations and space complexity is $\mathcal{O}(1)$.

```
// Calculates sums for index 0 - i, good if both
// queried and updated often
private static class BinaryIndexTree {
  long[] tree;
  public BinaryIndexTree(int size) {
    tree = new long[size+1];
  }
  long sum(int index) {
    long sum = 0;
    index++;
    while (index > 0) {
      sum += tree[index];
      index -= index & (-index);
    }
    return sum;
  }
  void update(int index, int delta) {
    index++;
    while (index < tree.length) {
      tree[index] += delta;
      index += index & (-index);
    }
  }
}
```

5.2. Segment Tree. Time complexity is $\mathcal{O}(n)$ for construction and $\mathcal{O}(\log n)$ for all operations and space complexity is $\mathcal{O}(n)$.

```
// Calculates max/min/sum of a range of values
public class SegmentTreeRMQ {
  public int[] segmentTree;
  public int length;

  // Constructs a segment tree
  public SegmentTreeRMQ(int[] input) {
    length = input.length;
    int x = (int) Math.ceil(
      Math.log(length) / Math.log(2));
    int size = 2 * (int)Math.pow(2, x) - 1;
    segmentTree = new int[size];
    construct(input, 0, length-1, 0);
  }
  private int construct(int[] input, int low,
      int hi, int i) {
    if (low >= input.length)
      return Integer.MAX_VALUE; //or min / 0
    if (low == hi) {
      segmentTree[i] = input[low];
      return input[low];
    }
    int mid = (low + hi) / 2;
    //can replace with max / sum
    segmentTree[i] = Math.min(
        construct(input, low, mid, 2*i + 1),
        construct(input, mid+1, hi, 2*i + 2));
    return segmentTree[i];
  }

  // Returns the minimum in the given range of indices
  public int rmq(int low, int hi) {
    return find(0, length-1, low, hi, 0);
  }
  private int find(int segLow, int segHi,
      int queryLow, int queryHi, int i) {
    if (queryLow <= segLow && queryHi >= segHi)
      return segmentTree[i];
    if (queryLow > segHi || queryHi < segLow)
      return Integer.MAX_VALUE; //or min / 0
    int mid = (segLow + segHi) / 2;
    return Math.min( //or max / sum
      find(segLow, mid, queryLow, queryHi, 2*i + 1),
      find(mid+1, segHi, queryLow, queryHi, 2*i + 2));
  }

  // Replaces the value at the given index
  public void update(int index, int value) {
    index = segmentTree.length/2 + index;
    segmentTree[index] = value;
    while (index > 0) {
      index = (index - 1) / 2;
      //or max / sum
```

```
        segmentTree[index] = Math.min(
            segmentTree[index*2+1],
            segmentTree[index*2+2]);
    }
  }
}
```

5.3. **Monotone Queue.** Time complexity is amortized $\mathcal{O}(1)$ for all operations and space complexity is $\mathcal{O}(w)$, where w is the size of the sliding window.

```
// Finds the min value in a sliding window, use push()
// to add new points to the window and poll() to
// remove points that are outside the window
public class MinMonoQueue<T extends Comparable<T>> {
  Deque<T> queue = new LinkedList<>();

  public void push(T obj) { // Use < for max queue
    while (!queue.isEmpty() &&
            queue.peekFirst().compareTo(obj) > 0)
      queue.pollFirst();
    queue.offerFirst(obj);
  }

  public T min() {
    return queue.peekLast();
  }

  public void pop(T obj) {
    if (queue.peekLast().compareTo(obj) == 0)
      queue.pollLast();
  }
}
```

5.4. **Union-Find.** Time complexity is amortized $\mathcal{O}(\log^* n)$ for all operations and space complexity is $\mathcal{O}(1)$.

```
// Used to efficiently build large sets and verify
// which set a node belongs to
public class UnionFind {
  public Node find(Node n) {
    if (n.parent != n)
      n.parent = find(n.parent);
    return n.parent;
  }
  public void union(Node a, Node b) {
    Node ra = find(a);
    Node rb = find(b);
    if (ra.rank > rb.rank) {
      rb.parent = ra;
    } else if (ra.rank > rb.rank) {
      ra.parent = rb;
    } else {
      ra.parent = rb;
      rb.rank++;
    }
  }

  static class Node {
    Node parent = this;
    int rank = 1;
  }
}
```

5.5. **Suffix Array.** Time complexity for construction is $\mathcal{O}(n \log n \log n)$ and space complexity is $\mathcal{O}(n)$.

```
// Sorts all the suffixes of a string into an array
public class SuffixArray {
  private String text;
  private Suffix[] suffixes;

  public SuffixArray(String text) {
    this.text = text;
    int N = text.length();
    suffixes = new Suffix[N];
    for (int i = 0; i < N; i++) {
      Suffix s = new Suffix();
      s.index = i;
      s.rank[0] = text.charAt(i);
      s.rank[1] = (N - i) < 2 ? -1 : text.charAt(i+1);
      suffixes[i] = s;
    }

    Arrays.sort(suffixes);

    int[] index = new int[N];
    for (int i = 2; i < N; i *= 2) {
      int prevRank = suffixes[0].rank[0];
      suffixes[0].rank[0] = 0;
      index[suffixes[0].index] = 0;
      for (int j = 1; j < N; j++) {
        Suffix suffix = suffixes[j];
        Suffix prevSuffix = suffixes[j-1];
        if (suffix.rank[0] == prevRank &&
            suffix.rank[1] == prevSuffix.rank[1]) {
          prevRank = suffix.rank[0];
          suffix.rank[0] = prevSuffix.rank[0];
        } else {
          prevRank = suffix.rank[0];
          suffix.rank[0] = prevSuffix.rank[0] + 1;
        }
        index[suffix.index] = j;
      }

      for (int j = 0; j < N; j++) {
        int nextIndex = suffixes[j].index + 2;
        suffixes[j].rank[1] = nextIndex < N ?
            suffixes[index[nextIndex]].rank[0] : -1;
      }

      Arrays.sort(suffixes);
    }
  }

  private static class Suffix implements
      Comparable<Suffix> {
    int index;
    int[] rank = { 0, 0 };

    @Override
    public int compareTo(Suffix o) {
      if (rank[0] != o.rank[0])
        return rank[0] - o.rank[0];
      if (rank[1] != o.rank[1])
        return rank[1] - o.rank[1];
      return index - o.index;
    }
  }
}
```

5.6. **Treap.** Time complexity for construction is $\mathcal{O}(n)$, for all operations $\mathcal{O}(\log n)$ and space complexity is $\mathcal{O}(n)$.

```java
// A randomly balanced binary search tree
public class Treap<T extends Comparable<T>> {
  Node root;

  // Adds key to this treap
  public void add(T key) {
    Node n = new Node(key);
    root = add(root, n);
  }
  private Node add(Node curr, Node newNode) {
    if (curr == null)
      return newNode;
    if (curr.priority > newNode.priority) {
      List<Node> res = split(curr, newNode.value);
      newNode.left = res.get(0);
      newNode.right = res.get(1);
      curr = newNode;
    } else if (curr.value.compareTo(newNode.value)
        <= 0) {
      curr.right = add(curr.right, newNode);
    } else {
      curr.left = add(curr.left, newNode);
    }
    updateSize(curr);
    return curr;
  }

  // Removes key from this treap, true on success
  public boolean remove(T key) {
    int s = size();
    root = remove(root, key);
    return size() < s;
  }
  private Node remove(Node curr, T key) {
    if (curr == null)
      return null;
    int comp = curr.value.compareTo(key);
    if (comp == 0) {
      curr = merge(curr.left, curr.right);
    } else if (comp < 0) {
      curr.right = remove(curr.right, key);
    } else {
      curr.left = remove(curr.left, key);
    }
    updateSize(curr);
    return curr;
  }

  // Merges this with a treap with larger elements
  public void merge(Treap<T> larger) {
    root = merge(root, larger.root);
  }
  private Node merge(Node l, Node r) {
    if (l == null || r == null)
      return l != null ? l : r;

    if (l.priority < r.priority) {
      l.right = merge(l.right, r);
      updateSize(l);
      return l;
    } else {
      r.left = merge(l, r.left);
      updateSize(r);
      return r;
    }
  }

  // Returns values that <= SP, leaves > SP behind
  public Treap<T> split(T splitPoint) {
    Treap<T> left = new Treap<>();
    List<Node> result = split(root, splitPoint);
    left.root = result.get(0);
    root = result.get(1);
    return left;
  }
  private List<Node> split(Node tree, T key) {
      Node l = null;
      Node r = null;
      if (tree != null) {
        if (tree.value.compareTo(key) <= 0) {
          List<Node> res = split(tree.right, key);
          tree.right = res.get(0);
          r = res.get(1);
          l = tree;
        } else {
          List<Node> res = split(tree.left, key);
          tree.left = res.get(1);
          r = tree;
          l = res.get(0);
        }
        updateSize(tree);
      }

      List<Node> landr = new ArrayList<>(2);
      landr.add(l);
      landr.add(r);
      return landr;
  }

  // Returns the size of this treap
  public int size() {
    return size(root);
  }
  private int size(Node n) {
    return n != null ? n.size : 0;
  }
  private void updateSize(Node node) {
    if (node != null)
      node.size = size(node.left) + size(node.right)+1;
  }

  // Returns whether or not this treap contains key
  public boolean contains(T key) {
    return contains(root, key);
  }
  private boolean contains(Node curr, T key) {
    if (curr == null)
      return false;
    int comp = curr.value.compareTo(key);
    if (comp < 0)
      return contains(curr.right, key);
    if (comp > 0)
      return contains(curr.left, key);
    return true;
  }

  class Node {
    T value;
    double priority;
    int size = 1;
    Node left, right;
```

```java
    public Node(T value) {
      this.value = value;
      priority = Math.random();
    }
  }
}
```

6. Graph Algorithms

6.1. Graph definition. This graph class is used for the graph algorithms. Not all attributes of the classes are needed in all problems.

```java
public class Graph {
  public class Node implements Comparable<Node> {
    int index;
    List<Edge> edges = new ArrayList<>();
    long cost = Long.MAX_VALUE;
    boolean taken;

    public Node(int idx) {
      index = idx;
    }

    public int compareTo(Node o) {
      if (cost == o.cost)
        return index - o.index;
      return (cost - o.cost) < 0 ? -1 : 1;
    }
  }

  public class Edge {
    int index;
    Node start, end;
    long cost;

    public Edge(int idx, Node s, Node e, long c) {
      index = idx;
      start = s;
      end = e;
      cost = c;
    }
  }
}
```

6.2. Dijsktra's Algorithm. Time complexity is $\mathcal{O}(|E|\log|V|)$ and space complexity is $\mathcal{O}(|V|)$.

```java
// Returns the cost from node s to t
public class Dijkstra {
  public long solve(Node s, Node t) {
    TreeSet<Node> queue = new TreeSet<>();
    s.cost = 0;
    queue.add(s);
    while (!queue.isEmpty()) {
      Node u = queue.pollFirst();
      if (u == t)
        return u.cost;

      for (Edge e : u.edges) {
        Node v = e.end == u ? e.start : e.end;
        long cost = u.cost + e.cost;
        if (cost < v.cost) {
          queue.remove(v);
          v.cost = cost;
          queue.add(v);
        }
      }
    }

    return -1;
  }
}
```

6.3. Bellman-Ford Algorithm. Time complexity is $\mathcal{O}(|E||V|)$ and space complexity is $\mathcal{O}(|V|)$. Handles negative weights and finds negative cycles.

```java
// Returns the cost from node s to all nodes (by index)
public class BellmanFord {
  public long[] solve(Node[] nodes, Edge[] edges, int s) {
    int V = nodes.length;
    long[] dist = new long[V];
    for (int i = 0; i < dist.length; i++)
      dist[i] = Long.MAX_VALUE;
    dist[s] = 0;

    for (int i = 0; i < V-1; i++) {
      for (Edge e : edges) {
        int n1 = e.start.index;
        int n2 = e.end.index;
        if (dist[n1] != Long.MAX_VALUE &&
            dist[n2] > dist[n1] + e.cost)
          dist[n2] = dist[n1] + e.cost;
      }
    }

    for (Edge e : edges) {
      int n1 = e.start.index;
      int n2 = e.end.index;
      if (dist[n1] != Long.MAX_VALUE &&
          dist[n2] > dist[n1] + e.cost)
        return null; // Negative cycle found!
    }
    return dist;
  }
}
```

6.4. All Pairs Shortest Paths. Time complexity is $\mathcal{O}(|V|^3)$ and space complexity is $\mathcal{O}(|V|^2)$. Implemented using the Floyd-Warshall algorithm. Handles negative weights and finds negative cycles.

```java
// Returns an adjacency matrix containing the costs
// between each pair of nodes
public class FloydWarshall {
  // Adjacency matrix; Long.MAX_VALUE means no edge
  public long[][] solve(long[][] adjacency) {
    int V = adjacency.length;
    long[][] dist = new long[V][V];
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        dist[i][j] = adjacency[i][j];

    for (int k = 0; k < V; k++) {
      for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
          if (dist[i][k] != Long.MAX_VALUE &&
              dist[k][j] != Long.MAX_VALUE &&
              dist[i][k] + dist[k][j] < dist[i][j]) {
            dist[i][j] = dist[i][k] + dist[k][j];
          }
        }
      }
    }
```

```
    for (int i = 0; i < V; i++) {
      if (dist[i][i] < 0)
        return null; // Negative cycle found!
    }
    return dist;
  }
```

6.5. **Minimum Spanning Tree.** Time complexity is $\mathcal{O}(|E|\log|V|)$ and space complexity is $\mathcal{O}(|V|)$. Implemented using Prim's algorithm.

```
// Returns the edges of the MST
public class MinimumSpanningTree {
  public List<Edge> solve(Node[] nodes, Node start) {
    Edge[] source = new Edge[nodes.length];
    TreeSet<Node> queue = new TreeSet<>();
    start.cost = 0;
    queue.add(start);
    while (!queue.isEmpty()) {
      Node u = queue.pollFirst();
      u.taken = true;
      for (Edge e : u.edges) {
        Node v = e.end == u ? e.start : e.end;
        if (e.cost < v.cost && !v.taken) {
          queue.remove(v);
          v.cost = e.cost;
          source[v.index] = e;
          queue.add(v);
        }
      }
    }

    List<Edge> mst = new ArrayList<>();
    for (Edge e : source) {
      if (e != null)
        mst.add(e);
    }
    return mst;
  }
```

6.6. **Topological Sort.** Time complexity is $\mathcal{O}(|E|+|V|)$ and space complexity is $\mathcal{O}(|V|)$. Running this on a graph with cycles yields an incorrect result.

```
// Returns topologically sorted nodes with the root
// as the first element
public class TopologicalSort {
  public List<Node> solve(Node[] nodes) {
    Stack<Node> stack = new Stack<>();
    for (Node u : nodes) {
      if (!u.taken)
        doSolve(stack, u);
    }
    List<Node> result = new ArrayList<>(stack);
    Collections.reverse(result);
    return result;
  }

  private void doSolve(Stack<Node> stack, Node u) {
    u.taken = true;
    for (Edge e : u.edges) {
      Node v = e.end;
      if (!v.taken)
        doSolve(stack, v);
    }
    stack.push(u);
  }
}
```

6.7. **Strongly Connected Components.** Time complexity is $\mathcal{O}(|E|+|V|)$ and space complexity is $\mathcal{O}(|V|)$. Implemented using Tarjan's algorithm.

```
// Finds all cycles (SCCs) in a graph
public class StronglyConnectedComponents {
  Stack<Node> stack;
  int nextIndex = 1;
  int[] indices; // 0 => uninitialized
  int[] lowLink; // 0 => uninitialized
  List<Node[]> sscs;

  public List<Node[]> solve(Node[] nodes) {
    stack = new Stack<>();
    sscs = new LinkedList<>();
    indices = new int[nodes.length];
    lowLink = new int[nodes.length];
    for (Node node : nodes) {
      if (indices[node.index] == 0) {
        stronglyConnected(node);
      }
    }
    return sscs;
  }

  private void stronglyConnected(Node u) {
    indices[u.index] = nextIndex;
    lowLink[u.index] = nextIndex++;
    u.taken = true;
    stack.push(u);

    for (Edge e : u.edges) {
      Node v = e.end;
      if (indices[v.index] == 0) {
        stronglyConnected(v);
        lowLink[u.index] = Math.min(lowLink[u.index],
                                    lowLink[v.index]);
      } else if (v.taken) {
        lowLink[u.index] = Math.min(lowLink[u.index],
                                    indices[v.index]);
      }
    }

    if (lowLink[u.index] == indices[u.index]) {
      List<Node> ssc = new LinkedList<>();
      Node v;
      do {
        v = stack.pop();
        v.taken = false;
        ssc.add(v);
      } while(u != v);
      sscs.add(ssc.toArray(new Node[0]));
    }
  }
}
```

6.8. **Network Flow/Min Cut.** Time complexity is $\mathcal{O}(|V||E|^2)$ and space complexity is $\mathcal{O}(|V|+|E|)$. Implemented using the Edmond-Karp algorithm. Solves both max flow and min cut.

If the graph is very large the running time can be improved to $\mathcal{O}(|E|^2\log C)$ (where C is the maximum flow). Find Δ, the largest POT that is smaller than the largest flow out of s. Run the algorithm but only allow edges with a capacity of at least Δ. When there are no more paths between s and t let $\Delta = \Delta/2$ and repeat until $\Delta < 0$.

```java
// Renamed Edge.cost -> capacity
public class NetworkFlow {
  // Find minimum s-t cut
  public List<Edge> solveMinCut(Node[] nodes,
      Edge[] edges, int s, int t) {
    List<Edge> result = new LinkedList<>();
    boolean[] visited = new boolean[nodes.length];

    solveFlow(nodes, edges, s, t);

    Queue<Node> queue = new LinkedList<>();
    queue.add(nodes[s]);
    visited[s] = true;
    while (!queue.isEmpty()) {
      Node u = queue.poll();
      for (Edge e : u.edges) {
        Node v = e.end;
        if (e.capacity > 0 && !visited[v.index]) {
          queue.offer(v);
          visited[v.index] = true;
        }
      }
    }

    for (Edge e : edges) {
      if (visited[e.start.index] &&
          !visited[e.end.index]) {
        result.add(e);
      }
    }

    return result;
  }

  // Find maximum s-t flow
  public long solveFlow(Node[] nodes, Edge[] edges,
      int s, int t) {
    Edge[] redges = new Edge[edges.length];
    for (int i = 0; i < redges.length; i++) {
      redges[i] = new Edge(i, edges[i].end,
        edges[i].start, 0);
      edges[i].end.edges.add(redges[i]);
    }
    long maxFlow = 0;
    List<Edge> path = new LinkedList<>();
    while (bfs(nodes, path, s, t)) {
      long minFlow = Long.MAX_VALUE;
      for (Edge e : path) {
        minFlow = Math.min(e.capacity, minFlow);
      }
      maxFlow += minFlow;
      for (Edge e : path) {
        Edge re = e == redges[e.index] ?
            edges[e.index] : redges[e.index];
        e.capacity -= minFlow;
        re.capacity += minFlow;
      }
    }
    return maxFlow;
  }

  private boolean bfs(Node[] nodes, List<Edge> path,
      int s, int t) {
    boolean[] visited = new boolean[nodes.length];
    Edge[] parent = new Edge[nodes.length];
    Queue<Node> queue = new LinkedList<>();
    queue.offer(nodes[s]);
    while (!queue.isEmpty()) {
      Node u = queue.poll();
      if (u.index == t)
        break;
      for (Edge e : u.edges) {
        Node v = e.end;
        if (e.capacity > 0 && !visited[v.index]) {
          queue.offer(v);
          visited[v.index] = true;
          parent[v.index] = e;
        }
      }
    }

    if (visited[t]) {
      path.clear();
      Node n = nodes[t];
      while (n != nodes[s]) {
        path.add(parent[n.index]);
        n = parent[n.index].start;
      }
      return true;
    }

    return false;
  }
```

6.9. **Bipartite Matching/Minimum Vertex Cover.** Time complexity is $\mathcal{O}(|E|\sqrt{|V|})$ and space complexity is $\mathcal{O}(|V|)$. Implemented using the Hopcroft-Carp algorithm. Gives both a maximum bipartite matching and a minimum vertex cover. Can be converted to a maximum independent set by selecting all vertices not in the vertex cover.

```java
public class HopcroftCarp {
  public static final int INF = Integer.MAX_VALUE;
  public static final int NIL = 0;
  Node[] L, R, G;

  // All indices for the nodes must be unique!
  public HopcroftCarp(Node[] L, Node[] R) {
    this.L = L;
    this.R = R;
    G = new Node[L.length + R.length + 1];
    for (Node n : L)
      G[++n.index] = n;
    for (Node n : R)
      G[++n.index] = n;
    G[NIL] = new Node(0);
  }

  // Returns the minimum vertex cover
  public Set<Node> solveMinVTC() {
    Map<Node, Node> Lm = solveMatching();
    Map<Node, Node> Rm = Lm.entrySet().stream().
        collect(Collectors.toMap(Map.Entry::getValue,
            Map.Entry::getKey));

    Queue<Node> queue = new LinkedList<>();
    boolean[] Z = new boolean[L.length + R.length + 1];
    for (Node n : L) {
      if (!Lm.containsKey(n)) {
        Z[n.index] = true;
        queue.add(G[n.index]);
      }
```

```java
    }

    while (!queue.isEmpty()) {
      Node u = queue.poll();
      for (Edge e : u.edges) {
        Node v = e.end == u ? e.start : e.end;
        if (!Z[v.index]) {
          Z[v.index] = true;
          if (Rm.containsKey(v)) {
            Node w = Rm.get(v);
            if (!Z[w.index]) {
              Z[w.index] = true;
              queue.add(w);
            }
          }
        }
      }
    }

    Set<Node> K = new HashSet<>();
    for (Node node : L) {
      if (!Z[node.index])
        K.add(node);
    }
    for (Node node : R) {
      if (Z[node.index])
        K.add(node);
    }
    return K;
  }

  // Returns the maximum bipartite matching
  public Map<Node, Node> solveMatching() {
    int[] pairs = new int[L.length + R.length + 1];
    int[] distance = new int[L.length + R.length + 1];

    while (bfs(G, L, pairs, distance)) {
      for (Node n : L) {
        if (pairs[n.index] == NIL)
          dfs(G, pairs, distance, n);
      }
    }
    Map<Node, Node> matches = new HashMap<>();
    for (Node n : L) {
      if (pairs[n.index] != NIL)
        matches.put(n, G[pairs[n.index]]);
    }
    return matches;
  }

  private boolean bfs(Node[] G, Node[] L, int[] pair,
      int[] distance) {
    Queue<Node> queue = new LinkedList<>();
    for (Node u : L) {
      if (pair[u.index] == NIL) {
        distance[u.index] = 0;
        queue.offer(u);
      } else {
        distance[u.index] = INF;
      }
    }

    distance[NIL] = INF;
    while (!queue.isEmpty()) {
      Node u = queue.poll();
      if (distance[u.index] < distance[NIL]) {
        for (Edge e : u.edges) {
          Node v = e.end == u ? e.start : e.end;
          if (distance[pair[v.index]] == INF) {
            distance[pair[v.index]] =
              distance[u.index] + 1;
            queue.offer(G[pair[v.index]]);
          }
        }
      }
    }
    return distance[NIL] < INF;
  }

  private boolean dfs(Node[] G, int[] pair,
      int[] distance, Node u) {
    if (u.index != NIL) {
      for (Edge e : u.edges) {
        Node v = e.end == u ? e.start : e.end;
        if (distance[pair[v.index]] ==
            distance[u.index] + 1 &&
            dfs(G, pair, distance, G[pair[v.index]])) {
          pair[v.index] = u.index;
          pair[u.index] = v.index;
          return true;
        }
      }
      distance[u.index] = INF;
      return false;
    }

    return true;
  }
}
```

7. Geometry

7.1. Convex Hull. Time complexity is $\mathcal{O}(n \log n)$ and space complexity is $\mathcal{O}(n)$. Implemented using Graham scan.

```java
// Finds the convex hull of an array of points
public class GrahamScan {
  public Point[] solve(Point[] points) {
    int N = points.length;
    Point minY = points[0];
    int index = 0;
    for (int i = 0; i < N; i++) {
      Point p = points[i];
      if (p.y < minY.y ||
          p.y == minY.y && p.x < minY.x) {
        minY = p;
        index = i;
      }
    }
    points[index] = points[N-1];
    points[N-1] = minY;

    Point.root = minY;
    Arrays.sort(points, 0, N-1);

    Point[] H = new Point[N+1];
    H[0] = points[N-2];
    H[1] = minY;
    for (int i = 2; i < N+1; i++) {
      H[i] = points[i-2];
    }

    int M = 1;
```

```java
    for (int i = 2; i <= N; i++) {
      while (Point.cross(H[M-1], H[M], H[i]) <= 0) {
        if (M > 1)
          M--;
        else if (i == N)
          break;
        else
          i++;
      }

      M++;
      Point tmp = H[i];
      H[i] = H[M];
      H[M] = tmp;
    }

    return Arrays.copyOfRange(H, 0, M);
  }

  static class Point implements Comparable<Point> {
    static Point root;
    double x, y;

    public Point(double x, double y) {
      this.x = x;
      this.y = y;
    }

    @Override
    public int compareTo(Point o) {
      int cross = cross(this, root, o);
      if (cross == 0) {
        return distSq(root, this) > distSq(root, o) ?
          1 : -1;
      }
      return cross;
    }

    static int cross(Point A, Point R, Point B) {
      double x1 = A.x - R.x;
      double x2 = B.x - R.x;
      double y1 = A.y - R.y;
      double y2 = B.y - R.y;
      return (int) -Math.signum(x1*y2 - x2*y1);
    }

    static double distSq(Point A, Point B) {
      double dx = A.x - B.x;
      double dy = A.y - B.y;
      return dx * dx + dy * dy;
    }
  }
}
```

8. Dynamic Programing

8.1. Knapsack 1/0. Given a set of items each with a value v_i and a weight w_i you want to maximize the value while limited by a total weight W. The following recursion relation solves the problem in $\mathcal{O}(nW)$:

$$Opt(i, W) = \begin{cases} 0 & \text{if } i = 0 \\ Opt(i-1, W) & \text{if } W < w_i \\ \max\{Opt(i-1, W), & \text{if } W \geq w_i \\ \qquad Opt(i-1, W-w_i) + v_i\} & \end{cases}$$

The answer is $Opt(n, W)$.

8.2. Knapsack Unbounded. The same problem as above but with an unlimited amount of each item. The following recursion relation solves the problem in $\mathcal{O}(nW)$:

$$Opt(W) = \begin{cases} 0 & \text{if } W = 0 \\ \max\limits_{w_i \leq W} \{Opt(W - w_i) + v_i\} & \text{otherwise} \end{cases}$$

The answer is $Opt(W)$.

8.3. Subset Sum. Given a set of values you want to select a subset that sum to W. This is solved by knapsack by letting $w_i = v_i$ and checking if $Opt(n, W) = W$.

8.4. Minimum Partition Distance. Given a set of n numbers s_i you want to split them into two sets A and B such that $|\sum a_i| - |\sum b_i|$ is minimized. The following recursion relation solves the problem in $\mathcal{O}(nS)$ (where S is the sum of all numbers):

$$Opt(i, d) = \begin{cases} d & \text{if } i = 0 \\ \arg\min\limits_x (x \in \{Opt(i-1, d-s_i), & \text{if } i > 0 \\ \qquad Opt(i-1, d+s_i)\} : |x|) & \end{cases}$$

The answer is $Opt(n, 0)$.

8.5. Edit distance. Given two strings a and b of length m and n find the minimum edit distance using penalties p_m for mismatches and p_s when padding with spaces. The following recursion relation solves the problem in $\mathcal{O}(mn)$:

$$Opt(i, j) = \begin{cases} j * p_s & \text{if } i = 0 \\ i * p_s & \text{if } j = 0 \\ Opt(i-1, j-1) & \text{if } a_i = b_j \\ \min\{Opt(i-1, j-1) + p_m, & \text{if } a_i \neq b_j \\ \qquad Opt(i-1, j) + p_s, & \\ \qquad Opt(i, j-1) + p_s\} & \end{cases}$$

The answer is $Opt(m, n)$. Example: ED("ABC", "ACD") = 2 ("ABC-" vs. "A-CD") where $p_m = p_s = 1$.

8.6. Longest Common Subsequence. Related to edit distance, you want to compute the longest common subsequence of two strings a and b of length m and n. The result of the algorithm is the string itself ($\frown$ appends to the result). The following recursion relation solves the problem in $\mathcal{O}(mn)$:

$$Opt(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ Opt(i-1, j-1) \frown a_i & \text{if } a_i = b_j \\ \text{longest}\{Opt(i-1, j), & \text{if } a_i \neq b_j \\ \qquad Opt(i, j-1)\} & \end{cases}$$

The answer is $Opt(m, n)$. Example: LCS("ABCD", "A-B-D-C") = "ABD".

8.7. Longest Increasing Subsequence. Time complexity is $\mathcal{O}(n \log n)$ and space complexity is $\mathcal{O}(n)$.

```java
// lis([1, 2, 5, 3]) == [1, 2, 3]
public class LongestIncreasingSubsequence {
  public int[] solve(int[] values) {
    int N = values.length;
    int[] indices = new int[N];
    int[] parents = new int[N];
    int top = 0;

    for (int i = 1; i < N; i++) {
      int v = values[i];
      int l = 0;
      int r = top;
      while (l <= r) {
```

```
      int m = (l+r+1)/2;
      if (values[indices[m]] < v)
        l = m+1;
      else
        r = m-1;
    }
    indices[l] = i;
    if (l > 0)
      parents[i] = indices[l-1];
    top = Math.max(top, l);
  }

  int[] lis = new int[top+1];
  int ind = indices[top];
  for (int i = top; i >= 0; i--) {
    lis[i] = values[ind]; // = ind; to get indices
    ind = parents[ind];
  }
  return lis;
 }
}
```

9. Scheduling

All the following problems consider the case where you get a list of n tasks t_i which may each have a start time s_i an end time e_i and a value v_i.

9.1. **1 machine, maximum tasks.** The goal is to maximize the amount of tasks done. Can be trivially solved by sorting the tasks by e_i in ascending order and greedily pick as many as possible. Time complexity is $\mathcal{O}(n \log n)$.

9.2. **1 machine, maximum time.** The goal is to maximize the amount of time spent working during a timeslot of length W. The tasks have a duration but no start time. This is solved by dynamic programming like the subset sum problem (see 8.3) by letting the task durations be the weights w_i.

9.3. **1 machine, maximum value.** The goal is to maximize the total value V of all the tasks that are serviced. This is solved by first sorting by e_i and then using dynamic programming. The following recursion relation solves the problem:

$$Opt(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max(v_i + Opt(p(i)), Opt(i-1)) & \text{if } i > 0 \end{cases}$$

Where $p(i)$ is the index of first task (backwards in time) that does not overlap with task i. The answer is $Opt(n)$ and the total time complexity is $\mathcal{O}(n \log n)$.

9.4. ***k*** **machines, maximum tasks.** The goal is to maximize the amount of tasks done given k machines. To solve this we first sort the tasks by e_i, then build a timeline of tasks as follows:

(1) Pick a task and try to put it on the timeline.
(2) If it collides with a previous task add a new layer to the timeline and try to put it there, if it still collides add another layer, etcetera...
(3) Return to the lowest level and go back to step 1.

When this is done you have a timeline of multiple layers each with an amount of tasks. To get the solution sort the layers in the timeline by decreasing amount of tasks and assign the k first layers to your machines. Time complexity is $\mathcal{O}(n \log n)$.

9.5. **Minimize machines, all tasks.** The goal is to minimize the amount of machines k needed to service *all* of the tasks. This can be solved by sorting the tasks by s_i and finding the maximum depth d (the maximum amount of simultaneous tasks). To solve it we need $k = d$ machines. To assign work go through the list and give each task to an idle machine. Time complexity is $\mathcal{O}(n \log n)$.

10. Checking for errors

10.1. **Wrong Answer.**

- Test minimal input
- Integer overflow?
- Double precision too low?
- Reread the problem statement
- Look for edge-cases
- Start creating small testcases

10.2. **Time Limit Exceeded.**

- Is the time complexity checked?
- Is the output efficient?
- If written in python, rewrite in java?
- Can we apply DP anywhere?
- Create worst case input

10.3. **Runtime Error.**

- Stack overflow?
- Index out of bounds?
- Division by 0?
- Concurrent modification?

10.4. **Memory Limit Exceeded.**

- Create objects outside recursive function
- Convert recursive functions to iterative with your own stack

11. Running time

The following table contains the number of elements that can be processed per second given the algorithm complexity in n.

Alg. Complexity	Input size/s
$\mathcal{O}(\log^* n)$	$\to \infty$
$\mathcal{O}(\log n)$	2 ^100 000 000
$\mathcal{O}(n)$	100 000 000
$\mathcal{O}(n \log n)$	4 500 000
$\mathcal{O}(n \log n \log n)$	300 000
$\mathcal{O}(n^2)$	10 000
$\mathcal{O}(n^2 \log n)$	3 000
$\mathcal{O}(n^3)$	450
$\mathcal{O}(2^n)$	26.5
$\mathcal{O}(3^n)$	16.5
$\mathcal{O}(n!)$	10

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

FIGURE 3. The ASCII table.