

# Hendrix Programming Team Reference

December 17, 2018



---

# Contents

<b>1</b>	<b>Limits</b>	<b>5</b>
<b>2</b>	<b>Java Reference</b>	<b>7</b>
2.1	Template . . . . .	7
2.2	Scanner . . . . .	7
2.3	String . . . . .	8
2.4	Arrays . . . . .	8
2.5	ArrayList . . . . .	8
2.6	Stack . . . . .	8
2.7	Queue . . . . .	8
2.8	PriorityQueue . . . . .	9
2.9	Set . . . . .	9
2.10	Map . . . . .	9
2.11	BigInteger . . . . .	9
2.12	Sorting . . . . .	9
2.13	BitSet . . . . .	9
2.14	Fast I/O . . . . .	9
<b>3</b>	<b>Data Structures</b>	<b>13</b>
3.1	Union-find . . . . .	13
3.2	Heaps . . . . .	14
3.3	Tries . . . . .	14
3.4	Red-black trees . . . . .	14
3.5	Segment trees and Fenwick trees . . . . .	14
<b>4</b>	<b>Search</b>	<b>15</b>
4.1	Complete search . . . . .	15
4.2	Binary and ternary search . . . . .	15
<b>5</b>	<b>Graphs</b>	<b>17</b>
5.1	Graph basics . . . . .	17
5.2	Graph representation . . . . .	17
5.3	BFS . . . . .	17
5.4	DFS, SCCs, topological sorting . . . . .	17
5.5	Single-source shortest paths (Dijkstra) . . . . .	17
5.6	All-pairs shortest paths (Floyd-Warshall) . . . . .	17
5.7	Min spanning trees (Kruskal) . . . . .	17
5.8	Max flow . . . . .	17
<b>6</b>	<b>Dynamic Programming</b>	<b>21</b>
<b>7</b>	<b>Strings</b>	<b>23</b>
7.1	Suffix arrays . . . . .	23

<b>8 Divide &amp; Conquer</b>	<b>25</b>
8.1 Counting inversions . . . . .	25
<b>9 Mathematics</b>	<b>27</b>
9.1 GCD/Euclidean Algorithm . . . . .	27
9.2 Fractions . . . . .	27
9.3 Primes and factorization . . . . .	27
9.4 Factorial . . . . .	27
9.5 Combinatorics . . . . .	27
<b>10 Bit Tricks</b>	<b>29</b>
<b>11 Geometry</b>	<b>31</b>
<b>12 Miscellaneous</b>	<b>33</b>
12.1 2D grids . . . . .	33
12.2 Range queries . . . . .	34
12.2.1 Prefix scan (inverse required; $O(1)$ queries; no updates) . . . . .	34
12.2.2 Kadane's Algorithm . . . . .	35
12.2.3 2D prefix scan . . . . .	35
12.2.4 Doubling windows (no inverse; $O(1)$ queries; no updates) . . . . .	36
12.2.5 Fenwick trees (inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates) . . . . .	36
12.2.6 Segment trees (no inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates) . . . . .	37
<b>13 Python</b>	<b>39</b>

# Chapter 1


## Limits

As a rule of thumb, you should assume about  $10^8$  (= 100 million) operations per second. If you can think of a straightforward brute force solution to a problem, you should check whether it is likely to fit within the time limit; if so, go for it! Some problems are explicitly written to see if you will recognize this. If a brute force solution won't fit, the input size can help guide you to search for the right algorithm running time.

Example: suppose a problem requires you to find the length of a shortest path in a weighted graph.

- If the graph has  $|V| = 400$  vertices, you should use Floyd-Warshall (§5.6, page 17): it is the easiest to code and takes  $O(V^3)$  time which should be good enough.
- If the graph has  $|V| = 4000$  vertices, especially if it doesn't have all possible edges, you can use Dijkstra's algorithm (§5.5, page 17), which is  $O(E \log V)$ .
- If the graph has  $|V| = 10^5$  vertices, you should look for some special property of the graph which allows you to solve the problem in  $O(V)$  or  $O(V \log V)$  time—for example, perhaps the graph is a tree (§5.1, page 17), so you can run a DFS (§5.4, page 17) to find a unique path and then add up the weights. An input size of  $10^5$  is a common sign that you are expected to use an  $O(n \lg n)$  or  $O(n)$  algorithm—it's big enough to make  $O(n^2)$  too slow but not so big that the time to do I/O makes a big difference.

$n$	Worst viable running time	Example
11	$O(n!)$	Generating all permutations (§9.5, page 27)
25	$O(2^n)$	Generating all subsets (§10, page 29)
100	$O(n^4)$	Some brute force algorithms
400	$O(n^3)$	Floyd-Warshall (§5.6, page 17)
$10^4$	$O(n^2)$	Testing all pairs
$10^6$	$O(n \lg n)$	BFS/DFS; sort+greedy

 [bing](#), [transportationplanning](#), [dancerecital](#), [prozor](#), [rectanglesurrounding](#), [weakvertices](#)

- $2^{10} = 1024 \approx 10^3$ .
- One `int` is 32 bits = 4 bytes. So *e.g.* an array of  $10^6$  `ints` requires  $< 4$  MB—no big deal since the typical memory limit is 1 GB. Don't be afraid to make arrays with millions of elements!
- `int` holds 32 bits; the largest `int` value is `Integer.MAX_VALUE` =  $2^{31} - 1$ , a bit more than  $2 \cdot 10^9$ .
- `long` holds 64 bits; the largest `long` value is `Long.MAX_VALUE` =  $2^{63} - 1$ , a bit more than  $9 \cdot 10^{18}$ .
- If you need larger values, use [BigInteger](#) (§2.11, page 9) or just use [Python](#) (§13, page 39); see [Combinatorics](#) (§9.5, page 27).



## Chapter 2

# Java Reference

### 2.1 Template

```
// *Don't* include a package declaration!

import java.util.*;
import java.math.*;

public class ClassName {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        // Solution code here

        System.out.println(answer);
    }
}
```

### 2.2 Scanner

`Scanner` is relatively slow but should usually be sufficient for most purposes. If the input or output is relatively large (> 1MB) and you suspect the time taken to read or write it may be a hindrance, you can use `Fast I/O` (§2.14, page 9).

```
import java.util.*;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        // All these read a single token (ignore leading whitespace,
        // then read up until but not including the next whitespace)
        String s = in.next();
        int     n = in.nextInt();
        long    l = in.nextLong();
        double  d = in.nextDouble();

        // WARNING!! A previous call to nextXXX will read up to a
        // newline character but leave it unconsumed in the input, so
        // the next call to nextLine() will just read that newline and
```

```

    // return an empty string!
    in.nextLine();    // throw away the empty line to get ready for the next

    // Read a whole line up to the next newline character.
    // Consumes the newline but does not include it in the
    // returned String.
    String line = in.nextLine();

    // Read until end of input
    while (in.hasNext()) {
        line = in.nextLine();
    }
}

```

## 2.3 String


The `String` type can be used in Java to represent sequences of characters.

[TODO: Useful `String` methods: concatenation, substring, `charAt`, ...?] [TODO: Converting between `String` and `char[]`, advantages and disadvantages of each]

 [battlesimulation](#), [bing](#), [connectthedots](#), [itsasecret](#), [shiritori](#), [suffixarrayreconstruction](#)

## 2.4 Arrays

[TODO: Basic array template/examples.] [TODO: Useful `Arrays` methods: `sort`, `copyOf`, `copyOfRange`, `fill`, `binarySearch`]


 [falcondive](#), [freefood](#), [traveltheskies](#)

## 2.5 ArrayList

[TODO: Basic template and examples. Useful `ArrayList` methods: `add`, `get`, `set`. Advantages/disadvantages compared to arrays.]


## 2.6 Stack

[TODO: Example using `Stack` class.] [TODO: Mention balanced parentheses, DFS]

 [backspace](#), [islands](#), [pairingsocks](#), [reservoir](#), [restaurant](#), [throws](#), [zgrade](#)

## 2.7 Queue


[TODO: `Queue` interface, `ArrayDeque` class]

 [brexit](#), [coconut](#), [ferryloading4](#), [shuffling](#)



## 2.8 PriorityQueue

[TODO: Examples of using `PriorityQueue`. Show how to construct with custom `Comparator`, eg. using lambda notation] [TODO: Note lack of `decreaseKey` operation (Dijkstra), use `remove` + `add`, not as fast]

 [bank](#), [guessthedatastructure](#), [knigsoftheforest](#)

## 2.9 Set

[TODO: `HashSet`, `TreeSet`]

## 2.10 Map

[TODO: `HashMap`, `TreeMap`] [TODO: Iterating over keys, values, both (`MapEntry`)]

## 2.11 BigInteger

[TODO: Examples. Useful methods, constructors (`gcd`, `mod`, base conversion!).]


 [basicremains](#)

## 2.12 Sorting

[TODO: Basic template for implementing `Comparable`] [TODO: `Arrays.sort`, `Collections.sort`] [TODO: Sorting with a custom `Comparator`] [TODO: Include code for basic sorting implementations (in case it's useful to code them up explicitly so they can be enhanced with extra info): insertion sort, mergesort, quicksort)]


## 2.13 BitSet

[TODO: Basic examples of `BitSet` use.]

 [primesieve](#)

## 2.14 Fast I/O

Typically ACM ICPC problems are designed so `Scanner` and `System.out.println` are fast enough to read and write the required input and output within the time limits. However, these are relatively slow since they are unbuffered (every single read and write happens immediately). Occasionally it can be useful to have faster I/O; indeed, some problems on Kattis cannot be solved in Java without using this. [TODO: [Link to some examples.](#)]

 Be sure to call `flush()` at the end of your program or else some output might be lost!

```
/* Example usage:
 *
 * Kattio io = new Kattio(System.in, System.out);
 *
 * while (io.hasMoreTokens()) {
 *     int n = io.getInt();
 *     double d = io.getDouble();
```

```

*    double ans = d*n;
*
*    io.println("Answer: " + ans);
* }
*
* io.flush();    // DON'T FORGET THIS LINE!
*/

import java.util.*;
import java.io.*;

class Kattio extends PrintWriter {
    public Kattio(InputStream i) {
        super(new BufferedOutputStream(System.out));
        r = new BufferedReader(new InputStreamReader(i));
    }
    public Kattio(InputStream i, OutputStream o) {
        super(new BufferedOutputStream(o));
        r = new BufferedReader(new InputStreamReader(i));
    }

    public boolean hasMoreTokens() {
        return peekToken() != null;
    }

    public int getInt() {
        return Integer.parseInt(nextToken());
    }

    public double getDouble() {
        return Double.parseDouble(nextToken());
    }

    public long getLong() {
        return Long.parseLong(nextToken());
    }

    public String getWord() {
        return nextToken();
    }

    private BufferedReader r;
    private String line;
    private StringTokenizer st;
    private String token;

    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;
}

private String nextToken() {
    String ans = peekToken();
    token = null;
    return ans;
}
}
```

[TODO: Add `getLine()` method]




## Chapter 3

# Data Structures

### 3.1 Union-find

A union-find structure can be used to keep track of a collection of disjoint sets, with the ability to quickly test whether two items are in the same set, and to quickly union two given sets into one. It is used in Kruskal's Minimum Spanning Tree algorithm (§5.7, page 17), and can also be useful on its own. `find` and `union` both take essentially constant amortized time.

 `drivingrange, islandhopping, kastenlauf, lostmap, minspantree, numbersetseasy, treehouses, unionfind, virtualfriends, wheresmyinternet`

```
class UnionFind {
    private byte[] r; private int[] p; // rank, parent


    // Make a new union-find structure with n items in singleton sets,
    // numbered 0 .. n-1 .
    public UnionFind(int n) {
        r = new byte[n]; p = new int[n];
        for (int i = 0; i < n; i++) {
            r[i] = 0; p[i] = i;
        }
    }

    // Return the root of the set containing v, with path compression. O(1).
    // Test whether u and v are in the same set with find(u) == find(v).
    public int find(int v) {
        return v == p[v] ? v : (p[v] = find(p[v]));
    }

    // Union the sets containing u and v. O(1).
    public void union(int u, int v) {
        int ru = find(u), rv = find(v);
        if (ru != rv) {
            if (r[ru] > r[rv]) p[rv] = ru;
            else if (r[rv] > r[ru]) p[ru] = rv;
            else { p[ru] = rv; r[rv]++; }
        }
    }
}
```

## 3.2 Heaps

## 3.3 Tries

 [boggle](#), [heritage](#), [herkabe](#), [phonelist](#)

## 3.4 Red-black trees

## 3.5 Segment trees and Fenwick trees

See [Range queries](#) (§12.2, page 34).

# Chapter 4

## Search

### 4.1 Complete search

[TODO: Complete search aka brute force]

### 4.2 Binary and ternary search

[TODO: Binary search on an array; binary search on unbounded function on the integers; binary search on real interval; ternary search] [TODO: Point out `Arrays.binarySearch`]





# Chapter 5

## Graphs

### 5.1 Graph basics

[TODO: Directed, undirected, weighted, unweighted, self loops, multiple edges] [TODO: characterization of trees] [TODO: New virtual source/sink node trick]

### 5.2 Graph representation

[TODO: Adjacency matrix, adjacency maps. Edge objects. Implicit graphs.]

### 5.3 BFS

[TODO: Code for BFS with level labelling, parent map.]



### 5.4 DFS, SCCs, topological sorting

[TODO: Code for DFS, start/finish labelling, top sorting, Tarjan's SCC algorithm]

### 5.5 Single-source shortest paths (Dijkstra)

### 5.6 All-pairs shortest paths (Floyd-Warshall)

### 5.7 Min spanning trees (Kruskal)


### 5.8 Max flow

A *flow network* is a directed, weighted graph where the edge weights (typically integers) are thought of as representing *capacities* (e.g. imagine pipes of varying sizes). The *max flow problem* is to determine, given a flow network, the maximum possible amount of *flow* which can move through the network between given source and sink vertices, subject to the constraints that the flow on any edge is no greater than the capacity, and the sum of incoming flows equals outgoing flows at every vertex other than the source or sink. Flow networks can be used to model a wide variety of problems.

[TODO: Enumerate a few problem types: item assignment; max bipartite matching; min cut]

[TODO: choose directed/undirected edges carefully!]

[TODO: Requires vertices  $0 \dots n-1$ : either carefully keep track of which numbers are for which vertices, or use lookup tables]

 [copsandrobbers](#), [escapeplan](#), [gopher2](#), [guardianofdecency](#), [marblestree](#), [maxflow](#), [mincut](#), [paintball](#), [waif](#)

Dinitz' Algorithm is probably the best all-around algorithm to use for solving max flow problems in competitive programming. It takes  $O(V^2E)$  in theory (although is often much faster in practice). In the special case where we are modelling a bipartite matching problem, Dinitz' Algorithm reduces to the Hopcroft-Karp algorithm which runs in  $O(E\sqrt{V})$ .

```
class FlowNetwork {
    private static final int INF = ~(1<<31);
    int[] level;
    boolean[] pruned;
    HashMap<Integer, HashMap<Integer, Edge>> adj;

    public FlowNetwork(int n) {
        level = new int[n];
        pruned = new boolean[n];
        adj = new HashMap<>();

        for (int i = 0; i < n; i++)
            adj.put(i, new HashMap<>());
    }

    public void addDirEdge(int u, int v, long cap) {
        if (adj.get(u).containsKey(v)) {
            adj.get(u).get(v).capacity = cap;
        } else {
            Edge e = new Edge(u,v,cap);
            Edge r = new Edge(v,u,0);
            e.setRev(r);
            adj.get(u).put(v, e);
            adj.get(v).put(u, r);
        }
    }

    // Add an UNdirected edge u<->v with a given capacity
    public void addEdge(int u, int v, long cap) {
        Edge e = new Edge(u,v,cap);
        Edge r = new Edge(v,u,cap);
        e.setRev(r);
        adj.get(u).put(v, e);
        adj.get(v).put(u, r);
    }

    public long maxFlow(int s, int t) {
        if (s == t) return INF;
        else {
            long totalFlow = 0;
            while (bfs(s,t)) totalFlow += sendFlow(s,t);
            return totalFlow;
        }
    }
}
```

```

private long sendFlow(int s, int t) {
    for (int i = 0; i < pruned.length; i++)
        pruned[i] = false;
    return sendFlowR(s, t, INF);
}

private long sendFlowR(int s, int t, long available) {
    if (s == t) return available;

    long sent = 0;
    for (Edge e : adj.get(s).values()) {
        if (e.remaining() > 0 && !pruned[e.to] && level[e.to] == level[s] + 1) {
            long flow = sendFlowR(e.to, t, Math.min(available, e.remaining()));
            available -= flow; sent += flow;
            e.flow += flow; e.rev.flow -= flow;
            if (available == 0) break;
        }
    }
    if (sent == 0) pruned[s] = true;
    return sent;
}

private boolean bfs(int s, int t) {
    for (int i = 0; i < level.length; i++) level[i] = -1;

    Queue<Integer> q = new ArrayDeque<>();
    q.add(s); level[s] = 0;
    while (!q.isEmpty()) {
        int cur = q.remove();
        for (Edge e : adj.get(cur).values()) {
            if (e.remaining() > 0 && level[e.to] == -1) {
                level[e.to] = level[cur] + 1;
                q.add(e.to);
            }
        }
    }
    return level[t] >= 0;
}

class Edge {
    int from, to;
    long capacity, flow;
    Edge rev;
    public Edge(int from, int to, long cap) {
        this.from = from; this.to = to; this.capacity = cap; this.flow = 0;
    }
    public void setRev(Edge rev) { this.rev = rev; rev.rev = this; }
    public long remaining() { return capacity - flow; }
}

```

[TODO: Include a sample solution using a flow network]



## Chapter 6

# Dynamic Programming



## Chapter 7

# Strings

### 7.1 Suffix arrays





## Chapter 8

# Divide & Conquer

### 8.1 Counting inversions



[excursion](#), [froshweek](#)



## Chapter 9

# Mathematics


### 9.1 GCD/Euclidean Algorithm

### 9.2 Fractions

### 9.3 Primes and factorization


[TODO: Basic primality testing and factorization with trial division. Sieving (primes, factors, Euler totient).]

### 9.4 Factorial

 `eulersnumber`, `factstone`, `howmanydigits`, `lastfactorialdigit`, `inversefactorial`, `loworderzeros`


[TODO: Computing factorials; size using logs, etc]

### 9.5 Combinatorics

 `anagramcounting`, `secretsanta`

[TODO: Basic principles of combinatorics. Code for computing binomial coefficients. Multinomial coefficients.]

[TODO: mod  $10^9 + 7$ .]

 Remember to use `long` if you need an answer mod( $10^9 + 7$ ) (which would fit in an `int`) but computing the answer requires *multiplying* mod( $10^9 + 7$ ).

[TODO: Heap's Algorithm for generating all permutations. See Bit Tricks for generating all subsets.]


[TODO: PIE?]



## Chapter 10

# Bit Tricks

[TODO: Basic bit manipulation. Using bitstrings to compactly represent sets/states. Iterating through all subsets with counter.]

 flipfive

[TODO: BitSet instead of array of booleans.]



## Chapter 11

# Geometry

[TODO: Points, vectors, angles. Degrees/radians. `atan2`. Dot product. Rotation. Vector magnitude, norm (squared), normalize. Perpendicular (generate, test).] [TODO: Cross product in 2D. Signed area (parallelogram, triangle), polygon area, right/left turn, inside/outside testing.] [TODO: Lines/rays (point + vector). Line intersection. Segment intersection. Closest point on a line/segment. Point/line distance.] [TODO: Convex hull.]





# Chapter 12

## Miscellaneous

### 12.1 2D grids

2D grids/arrays (of characters, numbers, booleans...) are a popular feature of many competitive programming problems.

- In many cases the grid should be thought of as a graph where each cell is a vertex which is connected by edges to its neighbors. Note that in these cases one rarely wants to explicitly construct a different representation of the graph, but simply use the grid itself as an (implicit) graph representation.
- It is often useful to be able to assign a unique number to each cell in the grid, so we can store ID numbers of cells in data structures rather than making some class to represent a pair of a row and column index. The easiest method is to number the first row from 0 to  $C - 1$  (where  $C$  is the number of columns), then the second row  $C$  to  $2C - 1$ , and so on.

0	1	2	...	$C - 1$
$C$	$C + 1$	$C + 2$	...	$2C - 1$
$2C$	$2C + 1$	$2C + 2$	...	$3C - 1$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$(R - 1)C$	$(R - 1)C + 1$	$(R - 1)C + 2$	...	$RC - 1$

- Using this scheme, to convert between  $(r, c)$  pairs and ID numbers  $n$ , one can use the formulas

$$(r, c) \mapsto r \cdot C + c \quad n \mapsto (n/C, n\%C)$$

- To list the four neighbors of a given cell  $(r, c)$  to the north, east, south, and west, one can of course simply list the four cases manually, but sometimes this is tedious and error-prone, especially if there is a lot of code to handle each neighbor that needs to be copied four times.

Instead, one can use the following template. The idea is that  $(dr, dc)$  specifies the *offset* from the current cell  $(r, c)$  to one of its neighbors; each time through the loop we rotate it counterclockwise by 1/4 turn using the mapping  $(dr, dc) \mapsto (-dc, dr)$  (see [Geometry \(§11, page 31\)](#)).

```
int dr = 1, dc = 0;
for (int k = 0; k < 4; k++) {
    int nr = r + dr, nc = c + dc;
    // process neighbor (nr, nc)

    int tmp = dr; dr = -dc; dc = tmp; // rotate offset
}
```

## 12.2 Range queries

Suppose we have a 1-indexed array  $A[1 \dots n]$  containing some values, and there is some operation  $\oplus$  which takes two values and combines them to produce a new value. Given indices  $i$  and  $j$ , we want to quickly find the value that results from combining all the values in the range  $A[i \dots j]$ , i.e.  $A[i] \oplus A[i+1] \oplus \dots \oplus A[j]$ .

For example,  $A$  could be an array of integers, and  $\oplus$  could be max, that is, we want to find the maximum value in the range  $A[i \dots j]$ . Likewise  $\oplus$  could be sum, or product, or GCD. Or  $A$  could be an array of booleans, and we want to find the AND, OR, or XOR of the range  $A[i \dots j]$ .

- For this to make sense, the combining operation must typically be *associative*, i.e.  $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ . (This is called a *semigroup*.)
- Sometimes there is also an inverse operation  $\ominus$  which “cancels out” the effects of the combining operation (this is called a *group*). For example, subtraction cancels out addition. On the other hand, there is no operation that can cancel out the effect of taking a maximum.
- If we only need to find the value of combining a *single* range  $A[i \dots j]$ , then ignore everything in this section and simply iterate through the interval, combining all the values in  $O(n)$  time.
- More typically, we need to do many queries, and  $O(n)$  per query is not fast enough. The idea is to preprocess the array into a data structure which allows us to answer queries more quickly, i.e. in  $O(1)$  or  $O(\lg n)$ .
- Sometimes we also need to be able to *update* the array in between queries; in this case we need a more sophisticated query data structure that can be quickly updated.

Each of the below subsections outlines one approach to solving this problem; for quick reference, each subsection title says whether an inverse operation is required, how fast queries are, and whether the technique can handle updates.

### 12.2.1 Prefix scan (inverse required; $O(1)$ queries; no updates)

In a situation where we have an inverse operation and we do not need to update the array, there is a very simple solution. First, make a *prefix scan array*  $P[0 \dots n]$  such that  $P[i]$  stores the value that results from combining  $A[1 \dots i]$ . ( $P[0]$  stores the “identity” value, e.g. zero if the combining operation is sum.)  $P$  can be computed in linear time by scanning from left to right; each  $P[i] = P[i-1] \oplus A[i]$ . Now the value of  $A[i \dots j]$  can be computed in  $O(1)$  time as  $P[j] \ominus P[i-1]$ .

Note that having  $P[0]$  store the identity value is not strictly necessary, but it removes the need for a special case. If  $A$  is already 0-indexed instead of 1-indexed, then it’s probably easier to just put in a special case for looking up the value of  $A[0 \dots j]$  as  $P[j]$ , without the need for an inverse operation.

For example, suppose we are given an array of  $10^5$  integers, along with  $10^5$  pairs  $(i, j)$  for which we must output the sum of  $A[i \dots j]$ . Simply adding up the values in each range would be too slow. We could solve this with the following code:

```
import java.util.*;
public class PrefixSum {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        // Read array
        int n = in.nextInt();
        int[] A = new int[n+1];
        for (int i = 1; i <= n; i++) {
            A[i] = in.nextInt();
        }
    }
}
```


```

// Do prefix scan
int[] P = new int[n+1];
for (int i = 1; i <= n; i++) {
    P[i] = P[i-1] + A[i];
}

// Answer queries
int Q = in.nextInt();
for (int q = 0; q < Q; q++) {
    int i = in.nextInt(), j = in.nextInt();
    System.out.println(P[j] - P[i-1]);
}
}


```

More commonly, a prefix scan is a necessary first step in a more complex solution.

 [divisible](#), [dvoniz](#), [srednji](#), [subseqhard](#)

### 12.2.2 Kadane's Algorithm

As an aside, suppose we want to find the subsequence  $A[i \dots j]$  with the *biggest* sum. A brute-force approach is  $O(n^3)$ : iterate through all  $(i, j)$  pairs and find the sum of each subsequence. Using the prefix scan approach, we can cut this down to  $O(n^2)$ , since we can compute the sums of the  $O(n^2)$  possible subsequences in  $O(1)$  time each. However, there is an even better  $O(n)$  algorithm which is worth knowing, known as *Kadane's Algorithm*.

The basic idea is simple: scan through the array, keeping a running sum in an accumulator, and also keeping track of the biggest total seen. Whenever the running sum drops below zero, reset it to zero. Below is a sample solution to  [commercials](#). Note that subtracting  $P$  from each input is specific to the problem, but the rest is purely Kadane's Algorithm.

```

// url: http://open.kattis.com/problems/commercials
// difficulty: 2.0
// tags: list, maximum, range, sum, greedy, radio, commercial, Kadane

import java.util.*;

public class Commercials {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int N = in.nextInt(); int P = in.nextInt();

        int max = 0, sum = 0;
        for (int i = 0; i < N; i++) {
            sum += in.nextInt() - P;
            if (sum < 0) sum = 0; // or sum = Math.max(sum, 0);
            if (sum > max) max = sum; // or max = Math.max(max, sum);
        }
        System.out.println(max);
    }
}

```

### 12.2.3 2D prefix scan

[TODO: make pictures]


It is possible to extend the prefix scan idea to two dimensions. Given a 2D array  $A$ , we create a parallel 2D array  $P$  such that  $P[i][j]$  is the result of combining all the entries of  $A$  in the rectangle from the upper-left corner to  $(i, j)$  inclusive. The simplest way to do this is to compute

$$P[i][j] = A[i][j] + P[i-1][j] + P[i][j-1] - P[i-1][j-1]$$

Including  $P[i-1][j]$  and  $P[i][j-1]$  double counts all the entries in the rectangle from the upper left to  $(i-1, j-1)$  so we have to subtract them.

Given  $P$ , to compute the combination of the elements in some rectangle from  $(a, b)$  to  $(c, d)$ , we can compute

$$P[c][d] - P[a-1][d] - P[c][b-1] + P[a-1][b-1]$$

 **prozor** can be solved by brute force, but it's a nice exercise to solve it using the above approach.

### 12.2.4 Doubling windows (no inverse; $O(1)$ queries; no updates)

[TODO: Include link to discussion in CP3]

### 12.2.5 Fenwick trees (inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates)

 **fenwick**, **supercomputer**, **turbo**, **moviecollection**

We can use a *Fenwick tree* to query the range  $A[i..j]$  (i.e. get the combination of all the values in the range  $A[i] \dots A[j]$  according to the combining operation  $\oplus$ ) in  $O(\lg n)$  time. We can also dynamically update any entry in the array in  $O(\lg n)$  time. If dynamic updates are required but we have an invertible combining operation, a Fenwick tree should definitely be the first choice because the code is quite short.

The code shown here stores `int` values and uses addition as the combining operation, so range queries return the *sum* of all values in the range; but it can be easily modified for any other type of values and any other invertible combining operation: change the type of the array, change the `+` operation in the `prefix` and `add` methods, change the subtraction in the `range` method, and change the assignment `s = 0` in `prefix` to the identity element instead of zero.

 Note that this `FenwickTree` code assumes the underlying array is 1-indexed!

```
class FenwickTree {
    private long[] a;
    public FenwickTree(int n) { a = new long[n+1]; }

    // A[i] += delta. O(lg n).
    public void add(int i, long delta) {
        for (; i < a.length; i += LSB(i)) a[i] += delta;
    }

    // query [i..j]. O(lg n).
    public long range(int i, int j) { return prefix(j) - prefix(i-1); }

    private long prefix(int i) { // query [1..i]. O(lg n).
        long s = 0; for (; i > 0; i -= LSB(i)) s += a[i]; return s;
    }
    private int LSB(int i) { return i & (-i); }
}
```

- The constructor creates a `FenwickTree` over an array of all zeros.
- To create a `FenwickTree` over a given 1-indexed array  $A$ , simply create a default tree and then loop through the array, calling `ft.add(i, A[i])` for each  $i$ . This takes  $O(n \lg n)$ .

- `ft.add(i, delta)` can be used to update the value at a particular index by adding `delta` to it.
- If you want to simply replace the value at index  $i$  instead of adding something to it, you could use `ft.add(i, newValue - ft.range(i,i))`.
- `ft.range(i,j)` returns the sum  $A[i] + \dots + A[j]$ .

[TODO: Discuss CP3 presentation of Fenwick trees; explain how Fenwick trees work]

### 12.2.6 Segment trees (no inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates)

[TODO: Segment trees.]



## Chapter 13

# Python

Python's built-in support for arbitrary-size integers (using `BigInteger` in Java is a pain!) and built-in dictionaries with lightweight syntax make it attractive for certain kinds of problems.

Below is a basic template showing how to read typical contest problem input in Python:

```
import sys

if __name__ == '__main__':

    n = int(sys.stdin.readline()) # Read an int on a line by itself
    for _ in range(n):           # Do something n times

        # Read all the ints on a line into a list
        xs = map(int, sys.stdin.readline().split())

        # Read a known number of ints into variables
        p, q, r, y = map(int, sys.stdin.readline().split())
```

[TODO: Mention basic Python data structures such as set, deque, list methods]