

# Team Reference Document

The University of Manchester - Spirit Monkeys

November 6, 2017

# 1 Algorithms

## 1.1 Maximum subarray sum

Consider the subproblem of finding the maximum-sum subarray that ends at position  $k$ . There are two possibilities:

1. The subarray only contains the element at position  $k$
2. The subarray consists of a subarray that ends at position  $k - 1$ , followed by the element at position  $k$

---

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k], sum + array[k]);
    best = max(best, sum);
}
```

---

## 1.2 Max of increasing then decreasing sequence

Binary search can also be used to find the maximum value for a function that is first increasing and then decreasing. Our task is to find a position  $k$  such that

- $f(x) < f(x + 1)$  when  $x < k$ , and
- $f(x) > f(x + 1)$  when  $x = k$

The idea is to use binary search for finding the largest value of  $x$  for which  $f(x) < f(x + 1)$ . This implies that  $k = x + 1$  because  $f(x + 1) > f(x + 2)$ .

---

```
int x = -1;
for (int b = n/2; b >= 1; b /= 2) {
    while(f(x+b) < f(x+b+1)) x += b;
}
int k = x + 1;
```

---

## 1.3 Generating permutations

The following algorithm generates the next permutation lexicographically after a given permutation. It changes the given permutation in-place.

1. Find the largest index  $k$  such that  $a[k] < a[k + 1]$ . If no such index exists, the permutation is the last permutation.
2. Find the largest index  $l$  greater than  $k$  such that  $a[k] < a[l]$ .
3. Swap the value of  $a[k]$  with that of  $a[l]$ .
4. Reverse the sequence from  $a[k + 1]$  up to and including the final element  $a[n]$ .

## 1.4 Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given  $n$  events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially.

The idea is to always select the next possible event that ends as early as possible. It turns out that this algorithm always produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events. One way to argue that the algorithm works is to consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

## 1.5 Longest increasing subsequence

---

```
// assuming nums is nonempty
int[] max = new int[nums.length];
Arrays.fill(max, 1);

int result = 1;
for(int i = 0; i < nums.length; i++){
    for (int j = 0; j < i; j++) {
        if (nums[i] > nums[j]) {
            max[i] = Math.max(max[i], max[j] + 1);
        }
    }
    result = Math.max(max[i], result);
}
```

---

## 1.6 Edit/Levenshtein distance

The edit distance or Levenshtein distance 1 is the minimum number of editing operations needed to transform a string into another string. The allowed editing operations are as follows:

- insert a character (e.g. ABC to ABCA)
- remove a character (e.g. ABC to AC)
- modify a character (e.g. ABC to ADC)

Suppose that we are given a string  $x$  of length  $n$  and a string  $y$  of length  $m$ , and we want to calculate the edit distance between  $x$  and  $y$ . To solve the problem, we define a function  $distance(a, b)$  that gives the edit distance between prefixes  $x[0..a]$  and  $y[0..b]$ . Thus, using this function, the edit distance between  $x$  and  $y$

equals  $distance(n - 1, m - 1)$ . We can calculate values of distance as follows:

$$distance(a, b) = \min(distance(a, b - 1) + 1), \quad (1)$$

$$distance(a - 1, b) + 1), \quad (2)$$

$$distance(a - 1, b - 1) + cost(a, b)) \quad (3)$$

, where  $cost(a, b)$  is 0 if  $x[a] == x[b]$ , else it is 1.

## 1.7 Static array queries

... TODO add static array query for 2D ...

## 1.8 Fenwick tree

... TODO add Fenwick tree ...

## 1.9 Seg tree

... TODO add seg tree ...

## 1.10 Quicksort

---

```
int i = l, j = h;
int p = num[(h + l) / 2];
while (i <= j) {
    while (num[i] < p) i++;
    while (num[j] > p) j--;
    if (i <= j) {
        swap(i, j);
        i++; j--;
    }
}
if (l < j)
    quicksort(l, j);
if (i < h)
    quicksort(i, h);
```

---

## 1.11 Dijkstra

... TODO add Dijkstra ...

## 1.12 Kruskal's algorithm

... TODO Kruskal's algorithm...

## 1.13 Prim's algorithm

... TODO add Prim ...

### 1.14 Convex hull

Finds the convex hull of the set of points in  $\mathcal{O}(n \log n)$  time

---

```

class Point implements Comparable<Point> {
    int x,y;
    public int compareTo(Point p) {
        if (this.x == p.x) return this.y - p.y;
        else return this.x - p.x;
    }
}

long cross(Point O, Point A, Point B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

Point[] convexHull(Point[] p) {
    if (P.length > 1) {
        int n = P.length, k = 0;
        Point[] H = new Point[2 * n];
        Arrays.sort(P);
        // Build lower hull
        for (int i = 0; i < n; ++i) {
            while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0) k--;
            H[k++] = P[i];
        }
        // Build upper hull
        for (int i = n - 2, t = k + 1; i >= 0; i--) {
            while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0) k--;
            H[k++] = P[i];
        }
        if (k > 1) {
            H = Arrays.copyOfRange(H, 0, k - 1);
        }
        return H;
    } else if (P.length <= 1) {
        return P;
    } else {
        return null;
    }
}

```

---

## 2 Java structures

### 2.1 java.util.Stack

Stack is LIFO. It includes commands such as, push, pop, peek, isEmpty and can search for items to return positions or the data itself (just like a list) i.e. .contains() .get(i)

## 2.2 `java.util.PriorityQueue`

An unbounded priority queue based on a priority heap. Following are the important points about `PriorityQueue`:

- The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used.
- A priority queue does not permit null elements.
- A priority queue relying on natural ordering also does not permit insertion of non-comparable objects.
- For ordered traversal use `Arrays.sort(pq.toArray())`.

Note: Implement `Comparable < T >` for the class `T`

Time Complexity:  $\mathcal{O}(\log n)$  time for the enqueue/dequeue;  $\mathcal{O}(n)$  for the remove/contains; constant time for the retrieval.

## 2.3 `java.util.ArrayDeque`

Resizable-array implementation of the `Deque` interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than `Stack` when used as a stack, and faster than `LinkedList` when used as a queue.

Most `ArrayDeque` operations run in amortized constant time. Exceptions include `remove`, `removeFirstOccurrence`, `removeLastOccurrence`, `contains`, `iterator.remove()`, and the bulk operations, all of which run in linear time.