Team Reference Document

The University of Manchester - Spirit Monkeys

November 5, 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

... TODO quicksort ...

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

... TODO add Convex Hull...

2 Java structures

2.1 java.util.Stack

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

2.2 java.util.PriorityQueue

An unbounded priority queue based on a priority heap. 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 (doing so may result in ClassCastException).

The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements – ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

A priority queue is unbounded, but has an internal capacity governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator() is not guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray()).

Note that this implementation is not synchronized. Multiple threads should not access a PriorityQueue instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe PriorityBlockingQueue class.

Implementation note: this implementation provides O(log(n)) time for the enqueuing and dequeuing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).

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.

2.4 java.util.PriorityQueue

The java.util.PriorityQueue class is 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.

Note: Implement $Comparable < T >$ for the class T.