CS2040S: Data Structures and Algorithms

Problem Set 2

Overview. This problem set consists of two problems closely connected to binary search.

The first is an easy optimization problem: you are given an array, and your goal is to find the largest element in the array.

The second is a more challenging load balancing problem: you have a collection of tasks and a collection of processors, and your goal is to efficiently assign a sequence of tasks to each processor.

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so in Coursemology by adding a Comment. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. Optimization

Here's a simple problem—find the maximum element in an array! Implement the following function:

```
public static int searchMax(int[] dataArray)
```

Given an array `dataArray` where every element is unique, return the value of the largest integer in the array.

There is one additional piece of information you need: the data in the array increases (or decreases) from one end until it reaches its maximum (or minimum), and then decreases (or increases) until it reaches the other end.

More precisely, if the data array has size n, there is some index j such that the subarray `dataArray[0..j]` is sorted (either increasing or decreasing) and the subarray `dataArray[j..n-1]` is sorted (either increasing or decreasing).

For example, the following are possible input arrays:

```
[2, 5, 7, 9, 15, 23, 8, 6]

[73, 42, 13, 5, -17, -324]

[100, 42, 17, 3, 8, 92, 234]
```

Use the most efficient algorithm you can. State the running time of your algorithm.

To think about (*Optional*):

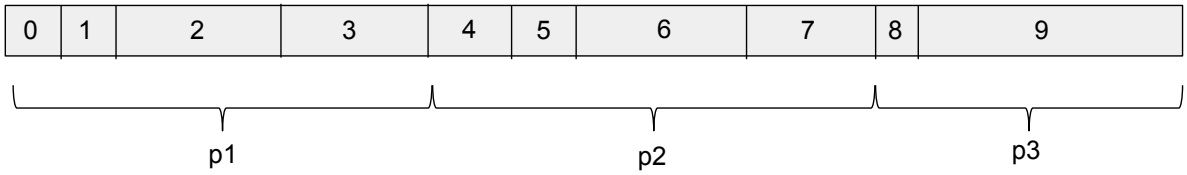
1. What is the best way to signal an error, e.g., if the data array is empty?
2. What if all the elements in the array are not unique? How efficiently can the problem be solved then? What is the worst-case example?
3. This array changes direction once. What if it changes direction twice? Is the problem much harder?
4. What if the array contains noisy data, so some of the entries in the array had $+1$ or -1 noise randomly added/subtracted from their initial value (with probability $1/2$ in each direction). Can you solve the problem in that case? What would the expected running time be?

Problem 2. Balancing the Load

Your boss, Paul R. Allel, storms into the office, shouting, "The system is too slow! Our users are sending me hate-tweets! Make it faster!" And then he stormed out again. What is he upset about? Well, right now, every day, your system churns through calculating dense matrix inverses for secret government agencies. And recently, there is so much work that one server just can't keep up. So you decide to parallelize: by buying more servers, you should be able to perform the computation faster!

Today, you have m jobs, stored sequentially on disk, in order. Each job has a size: `jobSize[i]` is the size of job i.

You have p processors that can perform these jobs. Each processor can be assigned a consecutive sequence of jobs, i.e., processor 2 might be assigned jobs $[4, 5, 6, 7]$. (For efficiency reasons[1], we do not wants gaps: it would not be legal to assign processor 2 the jobs $[4, 5, 7, 8]$.) Here, for example, we have 10 tasks assigned to three processors:



The total load of a processor is the sum of the job sizes, e.g., in the above case the load of processor 2 is:

```
jobSize[4] + jobSize[5] + jobSize[6] + jobSize[7]
```

Your goal in this problem is to find an assignment of jobs to processors in a manner that minimizes the maximum load on any processor (that is, consider the processor with the highest load — we want the load on this processor to be as small as possible). In fact, for today, we will just compute the minimum possible load.

Problem 2.a. The first step is to figure out, given a specific target load, whether p processors is sufficient. Design and implement the most efficient algorithm you can think of for the following function:

```
static boolean feasibleLoad(int jobSize[], int queryLoad, int p)
```

Your function should return `true` if it is possible to schedule the jobs on p processors in such a way that no processor has more load than `queryLoad`. State the running time of your algorithm.

[1]This is obviously an artificial constraint that we have added to make the problem tractable. See part (c), below.

Problem 2.b. The second step is to determine the minimum achievable load. Suppose we have 5 processors and jobs with the following sizes:

[1, 3, 5, 7, 9, 11, 10, 8, 6, 4]

To ensure that the maximum load on any one processor is as small as possible, we can assign the jobs to the 5 processors this way:

[1, 3, 5, 7] [9] [11] [10] [8, 6, 4]

Notice that the maximum load on any processor is 18 — that's the load on the last processor. With only 5 processors, this is the minimum achievable maximum load on any one processor, we can't go lower than this!

Design and implement the most efficient algorithm you can think of for the following function:

```
static int findLoad(int jobSize[], int p)
```

The function should return the minimum possible load for the given `jobSize` and `p`. State the running time of your algorithm.

Problem 2.c. *Optional, theory.* In fact, the constraint that each processor can only handle a consecutive set of jobs simplifies the problem, but in many practical systems is not necessary. Unfortunately, even for $p = 2$, the problem is NP-hard if you allow an arbitrary assignment of jobs to processors. However, there is a very simple algorithm that guarantees that no processor is assigned more than *twice* the minimum possible load. (That is, it computes a 2-approximation of optimal.) Give a simple algorithm that guarantees a 2-approximation, and prove that it is correct.

Hint 1: Think about a greedy algorithm that assigns each job to the least loaded processor.

Hint 2: Notice two facts about the "optimal" solution: its maximum load has to be at least as big as the biggest individual job, and its maximum load has to be at least as big as the sum of the job sizes divided by p.