Due: See website for due date. (no extensions).

What to submit: See website.

The theme of this exercise is automatic memory management. Please note that part 2 must be done on the rlogin machines and part 3 requires the installation of software on your personal machine.

1. Mark-and-Sweep Garbage Collection w/ Weak References

The first part of the exercise involves a recreational programming exercise that is intended to deepen your understanding of how a garbage collector works. You are asked to implement a variant of a mark-and-sweep collector using a synthetic heap dump given as input. On the heap, there are n objects numbered $0 \ldots n-1$ with sizes $s_0 \ldots s_{n-1}$. Also given are r roots and m pointers, i.e., references stored in objects that refer to other objects.

Write a program that performs a mark-and-sweep collection and outputs the total size of the live heap as well as the total amount of memory a mark-and-sweep collector would sweep if this heap were garbage collected.

In addition, some of the references (which correspond to edges in the reachability graph) are labeled as "weak" references. Weak references are supported in multiple programming languages as a means to designate non-essential objects that a garbage collector could free if it cannot reclaim enough memory from a regular garbage collection. For instance, in Java, weak references can be created using classes in the `java.lang.ref.-WeakReference` package. To ensure that the garbage collector does not cause unsafe accesses to objects when a weakly reachable object is freed, weak references in Java are wrapped in `WeakReference` objects which contain a `get()` method that provides a strong reference to the underlying object. If the object was reclaimed, `get()` would fail.

As a second part, your program should compute which objects would be part of the live heap, and which objects would be reclaimed, if the garbage collector chose to reclaim weakly reachable objects.

We will do this problem programming competition style. Write a program - in any language you choose[1] - that reads a heap description and outputs the size of the live heap and the amount of garbage swept assuming (a) that weakly reachable objects are kept and (b) that weakly reachable objects are reclaimed.

Your program *must* read from standard input. Each invocation of your program should process a single test case. The first line contains three non-negative 32-bit integers n, m, r such that $r \leq n$. The second line contains n positive 32-bit integers s_i that denote the size s_i of object i. Following that are m lines with tuples i, j for which $0 \leq i, j < n$, each of which denotes a pair of object indices. A tuple (i, j) means that object i stores a reference

[1]and which is available on the rlogin cluster so we can grade your submission. Contact techstaff@cs.vt.edu if you need a language not currently installed.

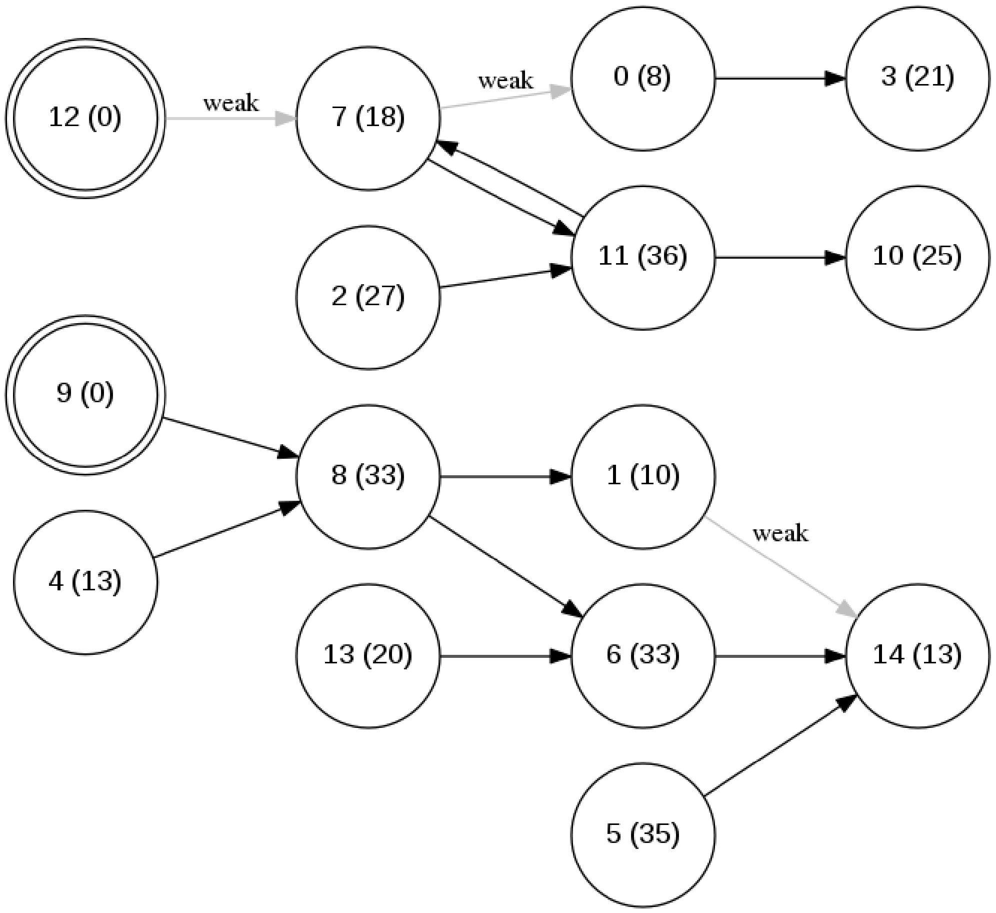

Figure 1: The reachability graph given in the sample input. Roots are shown using double circles. The numbers in parentheses are the sizes of individual nodes. Here, root 9 keeps alive object 8, which keeps 1 and 6 alive, which in turn keep 14. Root 12 keeps object 7 alive from which objects 0, 3, 10, and 11 are reachable. However, object 7 is kept alive only through a weak reference - thus, if the collector decides to collect objects that are only weakly reachable, object 7 would not be alive, causing 0, 3, 10, and 11 to not be reachable as well. On the other hand, 14 is not weakly reachable because there is also a strong reference to it from object 6.

to object j, keeping it alive (provided s_i is reachable from a root). If the tuple is followed by the letter `W`, then the reference is a weak reference. The description of the references is followed by a single line with r integers denoting the roots of the reachability graph $R_0 \ldots R_{r-1}$.

Your program should write to its standard output stream two lines. On the first, it should output two numbers `l s`, where l represents the total size of the live heap and s represents the amount of garbage that would be swept if the heap were collected and weakly reachable objects were not collected. On the second line, it should output the size of the live heap and the amount of garbage if weakly reachable objects were garbage collected as well.

Sample Input:

```
15 15 2
8 10 27 21 13 35 33 18 33 0 25 36 0 20 13
11 10
6 14
5 14
7 0 W
11 7
8 1
2 11
8 6
4 8
12 7 W
9 8
13 6
1 14 W
7 11
0 3
9 12
```

Sample Output:

```
197 95
89 203
```

Figure 1 shows the reachability graph for the sample input/output.

We will test your program on additional inputs. Your algorithm should have linear complexity in term of the number of edges.

2. Understanding Runtime Bloat

Programs that are subject to garbage collection run out of memory when the size of the live heap exceeds the maximum allowable size. When a program runs out of memory, one or both of the following reasons may be to blame:

- A Leak: objects are kept alive which the program will not access them on any future control flow path.
- Bloat: the program has either allocated too many objects or the objects allocated take up an unexpectedly large amount of memory.

In Java, a particular source of runtime bloat are the classes in the standard library [1]. In this part of the exercise, you are asked to perform a few experiments to quantify the cost of using certain standard library classes.

You will be using a JVMTI agent I created, which is an add-on the JVM loads when it starts. The agent allows a program to invoke a method to determine the size of the live heap at a given point in its execution and write it to a log file.

To run the heap tracker compile and run your Java program like so:

```
javac -cp /home/courses/cs3214/bin/heaptracker/heapTracker.jar:. LargeLiveHeap.java
```

```
runtracker LargeLiveHeap
```

As an example of how to use the heap tracker agent, consider the following program:

```
/*
 * Program that produces classic ramp for live memory
 * with linear allocation rate.
 */
public class LargeLiveHeap
{
    public static void main(String []av) {
        int numLive = 100;
        int numAllocations = 10000;

        HeapTracker.startTrace();
        byte[][] array = new byte[100][];
        HeapTracker.takeLiveHeapSample();
        for (int i = 0; i < 10000; i++) {
            array[i % array.length] = new byte[100000];
            HeapTracker.takeLiveHeapSample();
        }

        HeapTracker.stopTrace();
    }
}
```

Answer the following questions:

1. Provide a formula $s(i)$ that computes the value $s(i)$ in the second column of `heaptrace.dat` on line i (starting from 0). For instance, if `heaptrace.dat` contained

   ```
   0.124130 1
   0.126981 4
   0.129699 9
   ```

   then $s(i) = (i+1)^2$ would be a correct answer. Be sure to test your formula for errors against the actual heaptrace.dat file! Rename the `heaptrace.dat` as `tracelarge.dat` and include it in your submission.

2. Write a program to determine the asymptotic incremental per-object amount of live heap memory that is taken up by `java.lang.Integer` objects stored in an `Integer []`. As a reminder, recall that Java uses Autoboxing (link) and that the 8-bit integers are preallocated and cached (link). In other words, find out how many more bytes of memory are needed if one were to add a newly allocated `java.lang.Integer` to an `ArrayList`! Include in your number both the memory for the object itself as well as the memory to store references to it inside the `ArrayList`!

   Submit both your answer (a single integer) and the `heaptrace.dat` file you used to obtain this answer, renamed as `tracearraylist.dat`.

3. Repeat part 2 for a `Integers` stored in a `java.util.HashSet`! Submit both your answer and the `heaptrace.dat` file, renamed as `tracehashset.dat`.

4. Repeat part 2 for a `Integers` stored in a `java.util.LinkedList`! Submit both your answer and the `heaptrace.dat` file, renamed as `tracelinkedlist.dat`.

   For a deeper discussion of the issues surrounding runtime bloat, we recommend you read Mitchell et al [1], available on the course website (Link).

3. Reverse Engineering A Memory Leak

In this part of the exercise, you will be given a post-mortem dump of a JVM's heap that was obtained when running a program with a memory leak, which subsequently ran out of memory:

```
$ java -XX:+HeapDumpOnOutOfMemoryError -Xmx64m OOM
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid24735.hprof ...
Heap dump file created [64419731 bytes in 0.644 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at OOM$Cell.<init>(OOM.java:5)
        at OOM.main(OOM.java:16)
```

Your task is to examine the heap dump (oom.hprof) and reverse engineer the leaky program.

To that end, install the Eclipse Memory Analyzer on your computer. It can be downloaded from this URL. Open the heap dump.

Requirements and Hints

- Your program must run out of memory when run as shown above. You should double-check that the created heap dump matches the provided dump.
- The structure of the reachability graph of the subcomponent with the largest retained size should be the same in your heap dump as in the provided heap dump. (Other information may differ.)
- You will need to write one or more classes and write code that allocates these objects and creates references between them. You should choose the same field names and types in your program as in the heap dump, and no extra ones (we will check this). Think of field names as edge labels in the reachability graph.
- The heap dump contains only descriptions of the objects involved, but not their content. (For instance, you may see a `char[]` array, but you do not know which characters were stored in it when the `OutOfMemoryError` occurred.) Therefore, you cannot reverse engineer what content was contained in such objects.
- Hint: The program that was used to create the heap dump is 19 lines long (without comments, and including the main function).
- Hint: static inner classes are separated with a dollar sign $. For instance, `A$B` is the name of a static inner class called `B` nested in `A`. (Your solution should use the same class names as in the heap dump.)
- Hint: Start with the ”Leak Suspects” report, then look in Details. Use the ”List Objects ... with outgoing references” feature to find a visualization of the objects that were part of the heap when the program ran out of memory.
- Hint: The “dominator tree” option can also give you insight into the structure of the object graph. Zoom in on the objects that have the largest ”Retained Heap” quantity.

References

[1] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Four trends leading to java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.