Due: See website for due date.

What to submit: See website.

The theme of this exercise is automatic memory management and leak detection. Part 3 requires the installation of software on your personal machine.

1. Mark-and-Sweep Garbage Collection

The first part of the exercise involves a small 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, report the retained heap size for each of the roots. The retained heap size is the additional number of bytes that could be freed if this (and only this) root were removed from the graph and the garbage collection were repeated. Note that removing a root will also remove all of its outgoing edges.

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 if a garbage collection is performed. Then output the retained heap size for each root in the order in which they are given in the input.

As is customary for Unix programs, your program *should* read from its standard input stream. 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 to object j, keeping it alive (provided s_i is reachable from a root). The description of the references is followed by a single line with r integers denoting the roots of the reachability graph $R_0 \ldots R_{r-1}$. Nodes designated as roots do not have incoming edges that point to them.

Your program should write to its standard output stream $R + 1$ 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. On the following lines, for each root, it should output how much (additional) memory could be freed if this root were removed, in the order in which the roots appear in the input. In

[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.

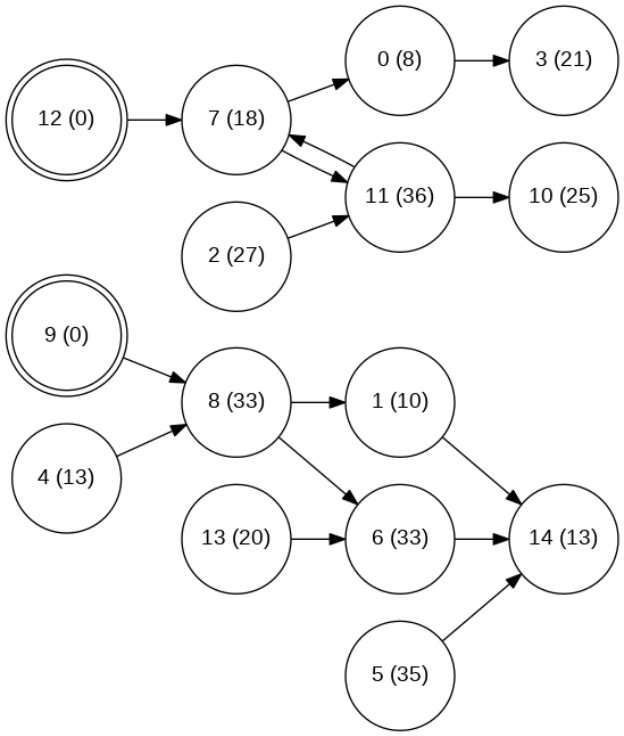

Figure 1: The reachability graph given in the sample input. Roots are shown using double circles. The numbers in parentheses are the sizes of individual nodes. Here, root 9 keeps alive object 8, which keeps 1 and 6 alive, which in turn keep 14. Root 12 keeps object 7 alive from which objects 0, 3, 10, and 11 are reachable. Note that in this example each root spans a different, disjoint subtree. This is not the case in general. Be sure you create test cases for which the sets of nodes that are reachable from each root overlap.

other words, compute the size of the additional garbage that would be produced if said root were removed.

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
11 7
8 1
2 11
8 6
4 8
12 7
9 8
13 6
1 14
7 11
0 3
9 12
```

Sample Output:

```
197 95
89
108
```

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

We will test your program on additional inputs. Your algorithm should have a complexity of at most $O(R \times (m + n))$ where R is the number of roots, n is the number of nodes, and m is the number of edges. You should further implement your algorithm under the assumption that the graph is sparse, that is, $m \ll n^2$.

Extra credit: Write an algorithm with a complexity of no worse than $O((m + n) \times \log n)$ where n is the number of nodes and m is the number of edges.

FAQ

- *Can I use graph library X, Y, or Z?*

  You may consult such code for reference, but do not use it directly. Keep in mind that tasks like these are tasks you may be asked during a programming interview where you also would need to be able to produce this code without the aid of auxiliary libraries.

- *Can I reuse code I've previously written for my data structure classes and/or programming competitions in which I've participated?*

  Yes. Please reference the original source.

2. Understanding valgrind's leak checker

Valgrind is a tool that can aid in finding memory leaks in C programs. To that end, it performs a garbage collection pass before a program exits and identifies still reachable objects, along with identifying leaks. For leaks, it uses the definition prevalent in C: objects that have been allocated but not yet freed, and there is no possible way for a legal program to access them in the future.

Read Section 4.2.8 Memory leak detection in the Valgrind Manual [URL] and construct a C program `leak.c` that, when run with

```
valgrind ./leak
```

produces the following output:

```
==3418358== Memcheck, a memory error detector
==3418358== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3418358== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==3418358== Command: ./leak
==3418358==
==3418358==
==3418358== HEAP SUMMARY:
==3418358==     in use at exit: 1,316 bytes in 7 blocks
==3418358==   total heap usage: 7 allocs, 0 frees, 1,316 bytes allocated
==3418358==
==3418358== LEAK SUMMARY:
==3418358==    definitely lost: 208 bytes in 2 blocks
==3418358==    indirectly lost: 300 bytes in 1 blocks
==3418358==      possibly lost: 700 bytes in 2 blocks
==3418358==    still reachable: 108 bytes in 2 blocks
==3418358==         suppressed: 0 bytes in 0 blocks
==3418358== Rerun with --leak-check=full to see details of leaked memory
==3418358==
==3418358== For lists of detected and suppressed errors, rerun with: -s
==3418358== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

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. The dump was produced at

the point in time when the program ran out of memory because its live heap size exceeded the maximum, which can be accomplished as shown in this log:

```
$ java -XX:+HeapDumpOnOutOfMemoryError -Xmx64m OOM
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid3420617.hprof ...
Heap dump file created [69287831 bytes in 0.137 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at OOM$Packet.<init>(OOM.java:6)
        at OOM.main(OOM.java:21)
```

Your task is to examine the heap dump (oom.hprof) and reverse engineer the leaky program. In addition, you should draw a sketch of a section of the reachability graph that shows the structure of the leak (you can't draw the entire reachability graph because that would be too large).

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

Requirements

- 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, where "matches" is defined as follows.
- 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 and class names** 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 (and you don't have to).
- Submit your sketch as a PDF file.

Hints

- The program that was used to create the heap dump is 24 lines long (without comments, and including the main function), though your line numbers may differ.

- 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.)
- 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.
- 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.
- Use the Java Tutor website to write small test programs and trace how the reachability graph changes over time.
- Do not forget the `-Xmx64m` switch when running your program, or else your program may run for several minutes before running out of memory, even if implemented correctly.
- Do not access the `oom.hprof` file through a remote file system path such as a mapped Google drive or similar. Students in the past have reported runtime errors in Eclipse MAT when trying to do that. Instead, copy it to your local computer's file system first as a binary file. The SHA256 sum of `oom.hprof` is

  `af8f0f0b3d62ba2c8ab52f9bb9b11939178c0273adee2a84cb74393071872f9e`