EECS 233 HW3

Ben Pierce
bgp12@case.edu

September 29, 2017

1 Question 1: Permutation Analysis

1.1 Part A

The constructor $Permutation()$ for the Permutation class is $O(n)$. This is because it goes through a single loop n times, with no deviation.

1.2 Part B

The method $getTrait()$ in Permutation.java is $O(1)$, because all it does is access an array once, or returns 0 if such an index does not exist. Only one operation is possible.

1.3 Part C

The method $getPerson()$ in Permutation.java is worst-case $O(n)$, because it computes a single $while$ loop. Once the condition $i < numPersons$ is met, the loop exits, going a maximum of n times. This means that the algorithm can take less time than predicted by its Big-O notation; for example, if the person being searched for is the first in the array, it will take a very small amount of time. However, if the person is the last element of the array, it will have to loop n times, causing the worst-case, $O(n)$.

1.4 Part D

The method $notEqual()$ in Permutation.java is $O(n^2)$. This is because there are two nested for loops, which conduct n operations n times, or $n * n = n^2$ times.

2 Question 2: Code Output

```
Without -Xint:

IntArrayBag tests:
Part A: N = 100000 3ms
Part B: N = 100000 3168 ms
Part C: N = 100000 345 ms
Part D: N = 100000 3ms

IntLinkedBag tests:
Part A: N = 100000 3ms
Part B: N = 100000 9547ms
Part C: N = 100000 19243ms
Part D: N = 100000 3ms

With -Xint

IntArrayBag tests:
Part A: N =  100000    5 ms
Part B: N =  100000    50540 ms
Part C: N =  100000    118608 ms
Part D: N =  100000    5 ms

IntLinkedBag tests:
Part A: N =  100000    10ms
Part B: N = 100000    52956ms
Part C: N =  100000    108035ms
Part D: N =  100000    10ms
```

3 Question 3: Big-O Comparisons

3.1 Part A

In the IntLinkedBag class, the *add*() method is $O(1)$. The method inserts a new IntNode as head, and increments the number of nodes by one. No loops are involved, and the operations inside the method are done only once, making the method $O(1)$. The *add*() method in the IntArrayBag class is slightly more interesting. When there is sufficient space to add a new element at the end of the array, the method does one assignment and returns, making it $O(1)$ as well for the best case. However, if there is not sufficient space, the method is forced to call the $ensureCapacity()$ method, which invokes $System.arraycopy()$. It would make logical sense that the $System.arraycopy()$ method is close to $O(n)$, due to the need to copy all elements of an array, unless the low-level nature of a System call can make this process slightly more efficent. However, the size need not increase with every call of $add()$. Therefore, the Big-O denotation for the worst-case $add()$ in IntArrayBag is somewhere between $O(n)$ and $O(1)$, with more weight towards $O(1)$ because of the infrequency of the invocation of $System.arraycopy()$. However, for the best-case, it is $O(1)$ like the method in IntLinkedBag, and very close to $O(1)$ for large n.

3.2 Part B

The remove methods in IntLinkedBag and IntArrayBag are worst-case $O(n)$. This is because both have a single for loop that executes N times. In IntArrayBag, the function simply searches for the target, and removes it if found and returns false if not. IntLinkedBag uses IntNode's $listSearch()$ method to find its target, which uses a single for loop that inspects every node for the target. For both of these methods, a single loop is executed n times, which makes both $O(n)$ algorithms.

3.3 Part C

The $countOccurrences()$ methods in IntLinkedBag and IntArrayBag are both worse-case $O(n)$. In IntLinkedBag, the method goes through a while loop that executes n times, incrementing a variable every time a match is found for the target. Since it is a single loop that executes n times, the method is $O(n)$. In IntArrayBag, the same is accomplished with a for loop. Since each algorithm utilizes a single loop that executes exactly n times, both are $O(n)$.

4 Question 4: Runtime Analysis

4.1 Part A

4.1.1 IntArrayBag

The runtimes for the $O(1)$ methods are shorter than the $O(n)$ methods. In Parts A and D, the $add()$ method is invoked, which is best-case $O(1)$. These run times were much shorter than their $O(n)$ counterparts in Parts B and C.

4.1.2 IntLinkedBag

Like IntArrayBag, the times for $O(n)$ operations are longer then $O(1)$ operations, although there is an odd result for Part C, which should return a larger runtime. Nonetheless, its run time is larger than the $O(1)$ operation.

4.2 Part B

The runtimes for the $O(1)$ were exactly the same for both classes, which makes sense in the context of this problem. With the -Xint option enabled, the times for the $O(n)$ options are also the same.

5 Experiment: Incrementing N

Out of curiosity, I decided to increment N by 10,000 and run each test 10 times. The results are below. Note that the program was run with the -Xint option, explained in Section 6 below.

```
IntArrayBag tests:
Part A: N =  10000   1 ms
Part B: N =  10000   508 ms
Part C: N =  10000   1164 ms
Part D: N =  10000   1 ms

IntLinkedBag tests:
Part A: N =  10000   1ms
Part B: N = 10000   539ms
Part C: N =  10000   1063ms
Part D: N =  10000   1ms
IntArrayBag tests:
Part A: N =  11000   1 ms
Part B: N =  11000   624 ms
Part C: N =  11000   1413 ms
Part D: N =  11000   0 ms

IntLinkedBag tests:
Part A: N =  11000   1ms
Part B: N = 11000   638ms
Part C: N =  11000   1277ms
Part D: N =  11000   1ms
IntArrayBag tests:
```

```
Part A: N =  12000   2 ms
Part B: N =  12000   740 ms
Part C: N =  12000   1687 ms
Part D: N =  12000   1 ms

IntLinkedBag tests:
Part A: N =  12000   1ms
Part B: N = 12000   763ms
Part C: N =  12000   1523ms
Part D: N =  12000   1ms
IntArrayBag tests:
Part A: N =  13000   1 ms
Part B: N =  13000   858 ms
Part C: N =  13000   1983 ms
Part D: N =  13000   1 ms

IntLinkedBag tests:
Part A: N =  13000   1ms
Part B: N = 13000   900ms
Part C: N =  13000   1768ms
Part D: N =  13000   1ms

IntArrayBag tests:
Part A: N =  14000   1 ms
Part B: N =  14000   987 ms
Part C: N =  14000   2250 ms
Part D: N =  14000   1 ms

IntLinkedBag tests:
Part A: N =  14000   2ms
Part B: N = 14000   1025ms
Part C: N =  14000   2287ms
Part D: N =  14000   1ms
```

It apppears that as N increases, the amount of time an $O(n)$ algorithim takes also increases in a linear fashion, as expected.

6 JVM Optimization Issues

After a discussion with Professor Fietkiewicz on Friday, I decided to do the additional test above. However, I ran into some issues. The time for Part C of the

IntArrayBag class would go down as N increased, which made no sense whatsoever. One possible explanation for this is that the JVM conducts runtime optimizations, which can cause errors. In order to disable these optimizations, I ran the programs with the -Xint option enabled (ex. java -Xint bagTest). This fixed that particular problem! The times for Part C increased in a linear fashion after this fix. I later went back and ran the original assignment with this option, as displayed in Section 2. Note that the times for IntArrayBag and IntLinkedBag are now close to equal. This makes intuitive sense, given that N is the same and both algorithms have the same Big-O value, which is $O(n)$ for parts B and C and $O(1)$ in parts A and B.