Lab 2: Big-O Analysis

September 24, 2017

Note that this is an **individual** lab. However, you are encouraged to discuss the problems with your neighbors, and, of course, ask the TAs for help.

Contents

Objectives

By the end of this lab, you will:

- understand how to find the absolute minimum bound for solving a simple problem.
- have a clear understanding of how to formally write up analysis of a function's big O runtime.

1 Minimum Possible Runtimes

Over the last week, we have been concerned with understanding and improving the runtime of our programs. This is a major concern in computer science, but there are limits to how much the solution to a given problem can be improved. Here, we will examine the "Minimum Possible Runtime" of several problems to gain the ability to understand goal states and when further runtime optimization is no longer useful.

So what is minimum possible runtime? It's the LOWEST POSSIBLE runtime a problem can conceivably be solved in. For example, if we had a list of n numbers and we wanted to find the minimum (or maximum) element, we know that our minimum possible runtime is

$$O([n \rightarrow n])$$

because we have to visit every element in the list. No matter what we do we cannot avoid checking every element.

Task: With this concept (minimum possible runtimes) in mind, we want to try to improve on an algorithm you learned in class. Shriram lectured about a sorting algorithm known as quicksort, which sorts lists in $O([n \rightarrow n \cdot \log n)]$ (in the average case). However, this algorithm was created in 1959, and the field of computer science has advanced since then, so surely we can do better now. Please design an algorithm which can sort a list of integers in $O([n \rightarrow n])$ using binary comparisons. Think about quicksort and how you could possibly improve it. What if you selected multiple pivots? Selected pivots based on some criteria? Or maybe quicksort is the wrong starting point and we should think about improving some other sorting algorithm? You do not need to implement it in Pyret, but be prepared to describe the process and runtime your algorithm will have to a TA.

Before continuing, call over a TA to check that your answer is right.

(**Note:** This is a challenging problem. If after working on it for 10-15 minutes you are stuck, please continue to the next task which will provide some pointers to help you in the right direction.)

Task: Please read this paper on time complexities for sort up through the "lower bound" section, and prepare to explain its contents to a TA, as well as how it relates to the previous problem.

Note: A tree is composed of nodes and leaves. A node has two children, which can be either nodes or leaves. Leaves have no children. You will be learning a lot more about trees in the Filesystem assignment!
Note: The paper makes reference to a decision tree. As you can see, the root of the tree contains all possible permutations of the input (as we don't know what the sorted ordering of elements is). As we perform comparisons we eliminate permutations based on the results until we end up with just one permutation remaining at a leaf, which represents the correct, sorted ordering of elements if the comparisons made along the path to the leaf hold.

Before continuing, call over a TA to check your answer

1.1 More MPRs

As you have seen, sometimes it's impossible to improve the runtime of an algorithm any further. However, constructing a lower bound on the runtime can be tricky; you have to understand the problem thoroughly and often need to perform in depth mathematical analysis. Here, we will present a number of simpler problems, for which you will find absolute minimum lower bounds. You do not need to implement a solution, but be prepared to explain your reasoning to a TA.

Task: Determine the minimum possible runtime complexity to sum all elements in a list of integers of size n.

Task: Determine the minimum possible runtime complexity to sum all elements in a list of integers of size n and a list of size m.

Task: Consider two lists of integers, of size m and n. Determine the minimum possible runtime complexity to find the sum of all possible products between an element from the first list and an element from the second list. For example, if the two lists were [list: 1, 2] and [list: 1] we would compute

$$1 \cdot 2 + 1 \cdot 1 = 3$$

Task: What if in the previous problem instead of taking the product of the element a from the first list and b from the second list, we used a^b? Would the minimum possible runtime complexity change? Using the examples from the previous problem, this would look like

$$1^1 + 2^1 = 3$$

Before continuing, call over a TA to check that your answer is right.

2 Formal Analysis Writeups

By now you should be fairly comfortable performing Big O analysis, but you may not yet be accustomed to writing up these findings in a more formal way.

When writing up this and similar analyses, it is important that you do so in a sufficiently structured way so that your solution and its correctness is easily understood. We have provided below a template for how this might look with a simple function.

Algorithm we will be analyzing

```
fun sols-silly-flipper<A>(lst :: List<A>) -> List<A>:
  doc: "A silly function that reorders lists."
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      link(f, sols-silly-flipper(reverse(r)))
  end
where:
  sols-silly-flipper(empty) is empty
  sols-silly-flipper([list: 1]) is [list: 1]
  sols-silly-flipper([list: 1, 2]) is [list: 1, 2]
  sols-silly-flipper([list: 1, 2, 3]) is [list: 1, 3, 2]
  sols-silly-flipper([list: 1, 2, 3, 4]) is [list: 1, 4, 2, 3]
  sols-silly-flipper([list: 1, 2, 3, 4, 5]) is [list: 1, 5, 2, 4, 3]
end
```

Our analysis:

Sol's flipping algorithm runs in $O([n \to n^2])$, where n represents the length of the input list, lst.

Each time the algorithm moves through the central cases structure, it reverses the portion of the list it has not yet handled. This has a cost of n - x, where x is the number of elements left in the list.

No value of x will be skipped, since the function handles only one element at a time: the list will be passed through with every integer length less than n and greater than 0.

When the length of the list is 0, we will incur some flat cost, r.

So the total cost of the function is

$$n + (n-1) + (n-2) + ... + (n-(n-2)) + (n-(n-1)) + r$$

This becomes n summorial + r, but since we are using Big O notation the addition and subtraction of smaller powers of n at the end drop off, leaving us with

$$O([n \to n^2])$$

Now, you will perform your own analysis for a few simple functions.

Task: Perform and write out a Big O analysis like above for the following implementation of the unique function from your summer work:

```
import Lists as l

fun my-reverse<elt>(loa :: List<elt>) -> List<elt>:
  doc: "outputs input list with elements in reverse order"
  l.foldl(lam(acc, cur): link(cur, acc) end, empty, loa)
where:
  my-reverse(empty) is empty
  my-reverse([list: 1]) is [list: 1]
  my-reverse([list: 1, 2]) is [list: 2, 1]
  my-reverse([list: 1, 2, 1]) is [list: 1, 2, 1]
  my-reverse([list: 2, 1, 1, 2, 3]) is [list: 3, 2, 1, 1, 2]
  my-reverse([list: 1, 2, 3, 4, 5]) is [list: 5, 4, 3, 2, 1]
end

fun unique<elt>(loa :: List<elt>) -> List<elt>:
  doc:"implementation of unique using hofs, outputs input list with no duplicate eleme
  unique-builder = lam(acc, cur):
      if not(l.member(acc, cur)):
          link(cur, acc)
      else:
          acc
      end
    end
  my-reverse(l.foldl(unique-builder, empty, loa))
where:
  unique(empty) is empty
  unique([list: 1, 2]) is [list: 1, 2]
  unique([list: 1, 2, 2, 1]) is [list: 1, 2]
  unique([list: 3, 3, 1, 3]) is [list: 3, 1]
  unique([list: 1, 2, 3, 4, 1, 2]) is [list: 1, 2, 3, 4]
  unique([list: 3, 3, 3, 3, 3]) is [list: 3]
end
```

Before continuing, call over a TA to check that your answer is right.

Task: Sol wants to restore the original ordering of lists he applied the flipper function to. Perform and write out a Big O analysis on his function below intended to repair the order of his lists:

```
fun undo-sol<A>(lst :: List<A>) -> List<A>:
  doc: ‘‘‘Function to restore the original ordering of lists
     reordered by sols-silly-flipper‘‘‘
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      undo-helper(lst, empty, empty)
  end
where:
  undo-sol(sols-silly-flipper(empty)) is empty
  undo-sol(sols-silly-flipper([list: 1])) is [list: 1]
  undo-sol(sols-silly-flipper([list: 1, 2])) is [list: 1, 2]
  undo-sol(sols-silly-flipper([list: 1, 2, 3])) is [list: 1, 2, 3]
  undo-sol(sols-silly-flipper([list: 1, 2, 3, 4])) is [list: 1, 2, 3, 4]
  undo-sol(sols-silly-flipper([list: 1, 2, 3, 4, 5])) is [list: 1, 2, 3, 4, 5]
  undo-sol(sols-silly-flipper([list: 1, 2, 3, 4, 5, 6])) is
  [list: 1, 2, 3, 4, 5, 6]
end

fun undo-helper<A>(original-list :: List<A>, start-piece :: List<A>,
    end-piece :: List<A>) -> List<A>:
    doc: "helper function for undo-sol"
  cases (List) original-list:
    | empty => my-reverse(start-piece).append(end-piece)
    | link(f, r) =>
      cases (List) r:
        |empty => undo-helper(empty, link(f,start-piece), end-piece)
        |link(f2,r2) => undo-helper(r2, link(f,start-piece), link(f2,end-piece))
      end
  end
end
```

Before continuing, call over a TA to check that your answer is right.

Extra fun: If you have extra time, you may attempt to implement the original silly-flipper function in linear time. If you feel this is impossible, provide a proof!
Super Extra fun: If you want to read about some interesting sorting algorithms, finish the paper or look up ”radix sort”!