1 Administrative Topics

- Quiz 5 will be graded tomorrow and I will hand it back on Friday. Sorry for the delay!
- I am sending out HW6 today.
- There will be a quiz on Friday. It will be about binary search on a sorted list.

2 Binary Search

We talked about search algorithms in the previous class. One advantage of the linear search we used last week was that it did not matter how the function behaved (it could have several peaks and several minima – it didn’t even need to be continuous).

But there are methods that can run much more quickly, if the objective function has certain properties. Our goal in the next project is to find the darting percentage that causes the elephant population to remain stable. So the objective function $c$ is:

$$c(\text{dartingPercent}) = \text{carryingCapacity} - \text{averagePopulationFromSimulation}(\text{dartingPercentage}, \text{otherParams})$$
As the darting percentage increases, the average population size decreases. Therefore, as the darting percentage increases, $c$ increases. This is a monotonically increasing function. Therefore, we can use a binary search:

Binary search is possible when the function has certain properties

- The objective function is continuous and a monotonic function of the parameter
- The sign of the error function tells us how to adjust the parameter

The idea behind a binary search is that we can eliminate half of the search space at every step. Suppose we test a darting percentage of 0.5. If the objective function is bigger than zero and is increasing, then we know that the ideal darting percentage must be less than 0.5. We can therefore ignore all values between 0.5 and 1.0.

The best way to learn a binary search is in a simpler context. Let’s first write a binary search that looks for a particular value in a sorted list of integers.

### 2.1 Binary search on a sorted list

The basic idea behind a binary search is that we divide the list into smaller and smaller sections. We begin with the entire list, then test the value at the middle index. If it is too small, then we confine the next step of the search to the upper half of the list. Otherwise, we confine our search to the lower half. We then test the middle value of the half of the list. If we haven’t found it yet, we confine ourselves to one quarter of the list. Then we test again. We keep going, until the section of the list is too small or we find the value we were looking for.
The algorithm for the binary search is given here (this is copied directly from the lab instructions)

```python
def searchSortedList( mylist, value ):
    # assign to the variable done, the value False
    # assign to the variable found, the value False
    # assign to the variable count, the value 0
    # assign to the variable maxIdx, the one less than the length of mylist
    # assign to the variable minIdx, the value 0

    # start a while loop that executes while done is not True
    # increment count (which keeps track of how many times the loop executes

    # assign to testIndex the average of maxIdx and minIdx (use integer math)

    # if the myList value at testIndex is less than value
    # assign to minIdx the value testIndex + 1
    # elif the myList value at testIndex is greater than value
    # assign to maxIdx the value testIndex − 1
    # else
    # set done to True
    # set found to True

    # if maxIdx is less than minIdx
    # set done to True
    # set found to False

    return ( found, count )
```

Consider the list

```python
nums = [3, 5, 7, 9, 11, 13, 15]
```

First, we search for 9 in the list. Since minIndex is 0 and maxIndex is 7, testIndex is 3. nums[3] is 9, so we find it right away (after having tested only 9).

Now, we search for 4 in the list. We test the values 9, 5, and 3 before determining 4 is not in the list.
2.2 Binary search on a continuous function

If we have a function instead of a list, then we will need to make some adjustments. In particular,

- We need to adjust how we choose the values for the optimization parameter. We specify the bounds, then begin at a midpoint. We adjust the value by adjusting the bounds and recomputing the midpoint with the new bounds.

- We need a tolerance for an exit strategy. We can’t assume we are going to find the exact value. Instead, we need to stop either when the value of the objective function is close enough to our objective, or when the bounds of our search space have gotten too small.

In the project, you will write code to test the parameter value between the lower and upper parameter bounds. You will decrease the bounds (restricting yourself to the upper half or the lower half of the current section). You will stop when

- you find the exact value you are looking for (unlikely to happen, but it could), or

- the difference between your lower and upper bounds are your parameters are too small, or

- you have looped too many times.

The algorithm for the project will always return a parameter value. That parameter value will be the best one your algorithm could find. That does not mean it is a good value.