COMP 161 - Lecture Notes 17 - Analyzing Search and Sort

In these notes we analyze search and sort.

Counting Operations

When we analyze the complexity of procedures we're determine the order of the number of **primitive operations** performed in the worst case of the procedure. Primitive operations are the operations we call on primitive data types. We count one such operation as one unit of work[1]. Almost all the operators we've encountered are primitive, but in C++ programmers can provide new definitions for operators so we must be careful about the assumption that all operators are primitive.

[1] The reality is that they are $O(1)$ hardware operations, but we ignore that because we're drawing the line at primitive C++

Search is Linear Time

So how good is our search and is there a difference between the iterative and recursive implementations. To answer this we first turn to complexity analysis to see how each implementation is classified.

Iterative Search

Let's start by re-introducing ourselves to the search code shown in figure 1.

```
int search(const std::vector<int>& data,int fst, int lst, int key){

  for(unsigned int i{0}; i < data.size() ; ++i){
    if( data[i] == key ){
      return i;
    }
  }

  return -1;
}
```

Figure 1: Search: Iterative Implementation

The first question of complexity analysis is **what's the worst case scenario for this code?** By this we mean, for what inputs will we maximize the work done? Looking a the code we see that either the loop runs to completion and the function returns -1 or it will terminate early and return the current value of *i*. Computational work will, therefore, be maximized when the loop completes and this occurs when the key is not in the vector. From here we can see that when the key is in the vector, then we'll do less work the closer the key is to the 0 location.

Now that we know the worst case occurs when the key isn't found, we can start counting up the work. This is where Big-O steps in. We'll count up work in pieces, determine the order of each piece, and then use the sum rule to reduce the total order of the procedure. All of the work done by this procedure is done by the loop, so this is really an example of loop analysis. To understand the work carried out by the loop we address three questions:

1. How much, if any, work is done but not repeated?
2. How many times does the loop repeat itself?
3. How much work is done per repetition?

If a loop repeats itself n times, does k work per repetition and c work outside of that, then the total work is $kn + c$. If k, n, and c are all constants then we're looking at some constant amount of work. For example, exactly 5 repetitions of 10 operations with 3 other operations on the side is just $O(53) = O(1)$ work. If, however, some of these values vary, then the loop is more complexity then constant.

A *for* loop has two parts that are done regardless of the number of repetitions: the initialization statement and one instance of the loop continuation check. The continuation check is the one check that must be false in order to terminate the loop. For *search* this is the initialization of i and one check that i is less than the size of the vector *data*. The initialization and the comparison itself are both primitive, so that's a constant number of operations[2]. Determining the size of a vector requires a std::vector method. A quick check of the std::vector reference tells us that the complexity of the size method is constant. Maybe it's 5 operations, maybe 500. Either way, the actual size of the vector doesn't influence the cost of looking up its size. All told, the operations not repeated by the loop itself but part of the loop structure are constant, or $O(1)$.

[2] $2 = O(1)$

Now, let's address the work repeated by the loop before we figure out the number of operations. Every time the *for* loop repeats it will do the continuation check and the update. We already know the continuation check is $O(1)$ operations. The update is just $i++$ which is one assignment and one addition[3], or just a constant, $O(1)$ number of operations. So far we know that the repeated code used by the loop itself is $O(1)$ operations per repetition. Now we look at the body of the loop. Every time the loop repeats the conditional check is done. This requires selecting the i^{th} integer with the std::vector *operator[]* and then comparing it to another integer with primitive integer $==$. The comparison is a single operation and if we look up

[3] $i = i + 1$

operator[] in the vector reference we see it too is $O(1)$. Combining the $O(1)$ operation inside the loop with the $O(1)$ operations done to control the loop gives us $O(1) + O(1) = O(1 + 1) = O(2) = O(1)$ operations.

Finally, we must determine how many times the loop repeats. If we start at 0, count up by 1, and stop after $i = data.size()$ then we'll do one iteration of the loop for every element of the vector. This was maybe obvious because that was our goal for the loop, to traverse and visit every single location in the vector. If n is the size of the vector, then this loop repeats $n = O(n)$ times in the worst case scenario when the key is not contained in the vector. Each repetition requires $O(1)$ work for a total of $O(n) * O(1) = O(n)$ work. *Iterative search has a complexity that is linear in the size of the vector we are searching.*

Recursive search

The recursive search is spread across two procedures as shown in figure 2.

Figure 2: Search: Recursive Implementation

```
int search(const std::vector<int>& data,int key){
  return search(data,0,data.size(),key);
}

int search(const std::vector<int>& data,int fst, int lst, int key){
  if( fst >= lst ){
    return -1;
  }
  else if( data[fst] == key ){
    return fst;
  }
  else{
    return search(data,fst+1,lst,key);
  }
}
```

Insertion Sort is Quadratic Time