COMP 161 — Lecture Notes 16
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);
  }
}
```

Analyzing the recursion itself requires a combination of knowing how to handle the use of helper procedures and knowing how a recursive process is carried out by the computer. The top level *search* function makes a call to the more general, recursive search. While we can't know the exact complexity of the process carried out by this search without knowing how the recursive helper behaves, we can fully understand the work done by the top-level search and not a part of the recursion. Before the recursive call is made we determine the size of the vector data, that's it. This means the two argument, top-level search does a constant, $O(1)$, amount of work *plus* whatever the recursive call with will do given its parameters. That's it.

In general, it's important to be able to say something definitive about a procedure's complexity in the face of uncertainty about helper complexity. As the search example demonstrates, it's simply a matter of identifying the work done outside the helper. We can actually go further with this search. If the recursive search is anything other than constant complexity, then we know that the work done by that recursion will dominate the $O(1)$ work done outside the recursion.[4] If the recursion itself is constant, then the sum total is constant[5]. This means that the top level search complexity will be as complex as the recursive helper, no more no less.

[4] the sum of two different classes is the max of the classes

[5] the basic sum rule of Big-O

Now we must manage the recursive version of search. Let's begin with the exact strategy we used for the top-level function and identify the complexity of all the work done outside of the recursive call. Under what circumstances is this work maximized? To answer this we need to look carefully at the conditional. The more conditions that fail, the more work done by the conditional because every test must be carried out. So the conditional itself will do the maximum amount of work with the *if* and *else if* condition are false. In that case each comparison requires $O(1)$ work[6]. The body of the *else* then requires another addition before the recursion, so we're still looking at some constant, $O(1)$, work. All told, each time search is called in the course of the recursion, it will perform $O(1)$ work. Figuring this out leaves us with one question: how many times does this procedure call itself?[7]

[6] so in complexity terms best and worst are the same $O(1)$

[7] notice the overall strategy here is the same as the loops: separate work repeated from repetition then combine that information

At this point we need to be clear about the mechanics of recursion. On one hand, it behaves no differently than any other procedure call, but on the other hand the procedure is calling itself and managing this sameness and otherness can take practice. Let's say we start with a vector containing $\{3, 1, 7, 5\}$ and we're searching for the key 2. This is an example of the worst case where the search key will not be found. The first call to the recursive search has $fst = 0$ and $lst = 4$. This instance of search makes a call with $fst = 1$ and $lst = 4$, that instance make call with $fst = 2$ and $lst = 4$, and so on until finally the base case is reached where$fst = lst = 4$. At this point each search returns -1 to the search in which it was called until finally the first instance returns -1 and the recursive chain is done as seen in figure 3.

The important thing to notice here is that for any value of $fst > lst$, search will call itself recursively while increasing fst by one such that fst will reach lst eventually and cause the procedure to begin returning back to the original call to search. If each call adds one to fst then n calls adds n and we need to determine when $fst + n = lst$ because at that point the recursion stops. It's clear that $n = lst - fst$. This is exactly the size of the region being searched. We now

```
search(data,2,0,4)
      ↪          search(data,2,1,4)
                       ↪          search(data,2,2,4)
                                        ↪          search(data,2,3,4)
                                                         ↪          search(data,2,4,4)
                                                   search(data,2,3,4)     ← -1 ↩
                                  search(data,2,2,4)     ← -1 ↩
                 search(data,2,1,4)     ← -1 ↩
search(data,2,0,4)     ← -1 ↩
    ← -1 ↩
```

Figure 3: An example sequence of recursive search calls for $data = \{3, 1, 7, 5\}$

know that in the worst case, recursive search calls itself once for every element in the search region. For each of those calls a constant amount of work will be done. For a search region of size n[8], search will do a total of $O(n) * O(1) = O(n)$ work. So, to search the entire vector for the key we first do $O(1)$ work then call the recursive search which does linear in the size of the vector work. Recursive search is no more or less complex than iterative search.

[8] again: $n = lst - fst$

Recurrence Relations (OPTIONAL)

To determine the complexity of the recursive search we analyzed the recursion in the same way that we analyzed loops: determine the work done per repetition, determine the amount of repetition, then combine the results as a product. There is a formal way of managing the analysis of recursive procedures using recursive mathematical functions. You'll learn this in Discrete Math and use it more advanced CS courses but I'll give you a quick taste of it here.

Let's say that T is the time function of our search procedure such that for searching a vector region size $n = lst - fst$, $T(n)$ is the time complexity of search. The base case occurs when $n = 0$ and in that case we find that search does a constant amount of work, or $T(0) = O(1)$. For $n > 0$, we see that the procedure does a constant amount of work then makes a recursive call to search a vector region of size $n - 1$. We express this with the RECURRENCE RELATION

$$T(n) = T(n-1) + O(1)$$

Our goal now is to solve this equation in such a way as to remove the recursion of T on the right hand side. One way to do this is simply to unwind the recursion in order to discover the pattern.

$$\begin{aligned}
T(n) &= T(n-1) + O(1) \\
&= (T(n-2) + O(1)) + O(1) = T(n-2) + 2(O(1)) \\
&= (T(n-3) + O(1)) + 2 * O(1) = T(n-3) + 3(O(1)) \\
&= (T(n-4) + O(1)) + 3 * O(1) = T(n-4) + 4(O(1)) \\
&= \ldots \\
&= T(n-k) + k * O(1)
\end{aligned}$$

At this point we see that after k recursive calls we've accumulated k units of constant work. When $k = n$ then we reach $T(0)$ and find that $T(n) = T(0) + O(n) = O(n)$ work is done. Many techniques exist for solving recurrence relations. The one demonstrated here is really just a more formalized version of the analysis that we did when analyzing recursive search.

Insertion Sort is Quadratic Time

Given that we're utilizing the same techniques to solve sort that we did to solve search, it should come as no surprise then that the analysis of sort follows the same path as that of search. The key difference will be that sort requires a non-trivial helper *insert*. Just like we did when managing the recursive version of search, we'll avoid getting overwhelmed by analyzing *insert* and *sort* independently and then when that's done we'll combine what we know to get a complete picture of the complexity of insertion sort.

Iterative Insertion Sort

Take a look at figure **??** and reacquaint yourself with the iterative implementations of both sort and insert.

The analysis of the sort procedure itself is pretty strait-forward if we set aside the work done by insert. The only work that is local to sort is the loop and that loop is a now familiar linear time loop. In this case there is no possibility of returning prior to violating the loop continuation condition, so for a vector of size n, this loop will always produce a $O(n)$ work. Put another way, there is only a single case for sort: call insert $n-1$ times. As a whole, this means sort has complexity no better than linear in the size of the vector because it takes at least that to drive the loop for insert.

The insert procedure works by systematically inserting the element that begins at *data[lst]* into the region of data running from *fst* to *lst-1* by swapping down from *lst-1* to *fst*. The loop and the procedure can stop early if *data[i+1]*, the item originally at *data[lst]*, is greater than or equal to *data[i]*. Otherwise the loop goes to completion and

```
void iter::sort(std::vector<int>& data){

  for(unsigned int i{1}; i < data.size(); i++){
   iter::insert(data,0,i);
  }
  return;
}

void iter::insert(std::vector<int>& data,
      unsigned int fst, unsigned int lst){

  for(unsigned int i{lst-1}; i >= fst && i < data.size(); i--){
   if( data[i+1] < data[i] ){
   std::swap(data[i],data[i+1]);
   }
   else{
   return;
   }
  }
  return;
}
\label{code:isortiter}
\caption{Sort \& Insert: Iterative Implementations}
```

traverses the entire insertion region. This means that the worst case occurs whenever the loop goes to completion which happens when the element being inserted is less than all the values in the insertion region. We can simplify the discussion of the complexity of insert by recognizing that the total work here is driven by the size of the insertion region which is $n = lst - fst + 1$[9].

[9] the +1 this time is due to the inclusion of *lst*

When the worst case of insert occurs, the loop traverses the entire insertion region of size $n = lst - fst + 1 = O(lst - fst) = O(n)$ time. We know this because it's the now all to familiar traversal by steps of size one loop that we've used time and time again[10]. The work on the inside of the loop will, in the worst case, require the comparison of two vector elements, $O(1)$ work, and a swap of those two elements. A quick check of the C++ reference tells us that this form of *std::swap* requires constant time[11]. Thus, the total work done on each iteration of the loop is $O(1)$, the loop does $O(n)$ iterations, and the total work needed by insert is linear in the size of the insertion region.

[10] Notice the continuation condition of this loops is more expensive than usual but only by a constant amount, which washes out in Big-O

[11] temp = a; a = b; b = temp;

We now know that insert does work that is linear in the size of the insertion region and that sort calls insert once for all but one of the vector elements. The real trick here is that each time sort calls insert the region of insertion grows by one. The work done by insert is dependent on the value of sort's loop counter variable i. At this point we need to look for a pattern in the work done by *insert* and use it to determine the total work done by all the inserts. The key observation is that the size of the insertion region grows by 1 each time. It starts a size 1 and stops after inserting into a region of size $n-1$ for a vector of size n. The total work done by every call to insert will then be sum of a series of insertions with an incrementally increasing work load, namely:

$$O(1)+O(2)+\cdots+O(n-2)+O(n-1) = O(1+2+\cdots+(n-2)+(n-1))$$

We must now be very careful as this isn't like the other sums we've dealt with while doing complexity analysis. The number of terms in this sum varies with n, and, in fact, it has exactly $n-1$ terms to it.

This kind of increasing sum is incredibly common in complexity analysis and in mathematics generally. It's what's called an arithmetic series. In mathematics it can be written very compactly using summation notation as shown in equation 1.

$$\sum_{i=0}^{n} i = 0+1+\cdots+(n-1)+n \tag{1}$$

The problem we currently have is that the series we're dealing with is open expression[12] and we don't know it's exact value. What we need is a closed-from expression[13] for this series that would let us calculate it's exact value and order.

[12] variable number of terms

[13] fixed number of terms

Before we go about solving for the closed form of this expression, we can intuit our way to an upper and lower bound. We know that we're dealing with a series of $n-1$ terms with the minimum of 1 and a maximum of $n-1$. What if each of them were just 1? Well then we'd just have a very long winded expression for $(n-1)$[14] which is $O(n)$. So certainly our series is no better than linear. Maybe we already figured that was the case. What if each term were $(n-1)$? Then we'd have $(n-1)$ occurrences of $(n-1)$ or n^2-2n+1 which is $O(n^2)$. So this series is no worse than quadratic. This worst case, quadratic upper-bound logic actually gives a real clue as to the exact nature of the series.

[14] $(n-1)$ 1's added together is $(n-1)$

Imagine an $(n-1)$ by $(n-1)$ square. This square represents the sum of $(n-1)$ terms each equal to $(n-1)$ where each column is a term and the height of the column is the term's value. The area of this square *is* the total of the sum. Now, what would our series look like? It'd be a triangle where the first column is height 1, the second is 2 and on to the $(n-1)^{th}$ column of height $n-1$. Once again, the area of this triangle is the total of our sum. This triangle just so happens to be the right triangle that comprises half of our

$(n-1) \times (n-1)$ square. It's area is, therefore, $\frac{n^2-2n+1}{2}$ which is $O(n^2)$. Returning back to our sort, we now know that the total work done by all the calls to insert is $O(n^2)$, quadratic, for a vector of size n. Adding this to the linear work done by the loop that drives the inserts, we find that our iterative insertion sort is quadratic in the size of the vector.

Before we move on let's go back to that series and see if we can't tease out a closed form using something similar to our triangle logic. Let's say we take the basic series that sums from 0 up to n. It has $n+1$ terms. We don't know what the exact value of the sum is but let's call it $\mathcal{S}$. What happens if we add it to itself. Obviously we have $2\mathcal{S}$, but there is something interesting we can see about $2\mathcal{S}$ if we flip one of the series around. As we see in figure 4, each term is equal to n and the total must then be $n(n+1)$.

$$\begin{array}{cccccccccccccccccc}
 & (& 0 & + & 1 & + & 2 & + & \cdots & + & (n-2) & + & (n-1) & + & n &) & \\
+ & (& n & + & (n-1) & + & (n-2) & + & \cdots & + & 2 & + & 1 & + & 0 &) & \\
\hline
 & (& n & + & n & + & n & + & \cdots & + & n & + & n & + & n &) & = n(n+1)
\end{array}$$

Figure 4: Adding $\mathcal{S}$ to itself

Of course, this $n(n+1)$ is twice as much as we want so we simply divide by two and get $\mathcal{S}$. We now have a closed form for the series as well as the Big-O of the series as shown in equation 2.

$$\sum_{i=0}^{n} i = 0 + 1 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = O(n^2) \qquad (2)$$

For insertion sort we had an initial value of 1, not 0, and a terminal value of $n-1$, not n. Not including 0 doesn't really change anything and we can simply plug $n-1$ in for n.

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

We got this result another way, but now that we know the closed form for arithmetic series we can use it whenever we encounter it in complexity analysis.

Recursive Insertion Sort

Figure 5 lists all the code for the recursive implementation of insertion sort.

This analysis is based entirely on ideas we've encountered before. The top-level sort does a small constant amount of work. The recursive sort will do a constant amount of work per recursive call and

```cpp
void recur::sort(std::vector<int>& data){
  recur::sort(data,0,data.size());
  return;
}

void recur::sort(std::vector<int>& data,int fst, int lst){
  if( fst >= lst-1 ){
    return;
   }

   recur::sort(data,fst+1,lst);
   recur::insert(data,fst,lst-1);
   return;
}

void recur::insert(std::vector<int>& data,
       unsigned int fst, unsigned int lst){
   if( fst >= lst ){
    return;
   }

   if( data[fst] > data[fst+1] ){
    std::swap(data[fst],data[fst+1]);
    recur::insert(data,fst+1,lst);
   }

}
```

Figure 5: Sort & Insert: Recursive Implementations

will call itself recursively $n-1$ times for a vector of size n. We can tell because the recursive call increments fst by 1 and recursion terminates when $fst = lst - 1$. The initial values for fst and lst were 0 and the size of the vector respectively. Just like with recursive search, this means one recursive call to sort per element in the vector[15]. This means that all the work done by sort but excluding the work done by insert must be linear in the size of the vector.

[15] this is the intent of basic recursion after all, to manage/sort one element at a time

Insert exhibits the same kind of linear recursive pattern we saw from sort. In the worst case, it will call itself once per element in the insertion region by incrementing fst by one with each call and pushing it towards lst. With each call to insert a constant amount of work is done. Thus we see that recursive insert is, like it's iterative counterpart, linear in the size of the insertion range.

Finally, we must account for all the work done by all the calls to insert. Each of the n calls is done on regions of decreasing[16] size. The first insert has a region of size $n-1$ and the last has a region of size 1. We know this pattern. We just solved this pattern. It's the arithmetic series and it's $O(n^2)$. So, just like the iterative insertion sort, recursive insertion sort requires work that is quadratic in the size of the vector.

[16] or increase if you're thinking about the actual order in which they get called as opposed to the order in which sort gets called