COMP 161 - Lecture Notes - 17 - Search

April 28, 2015

In these notes we look at the design and analysis of search procedures.

Searching in the C++ STL

The C++ standard template library provides two general purpose searches: *std::find*[1] and *std::binary_search*[2]. Both procedures are found in the *algorithm* library. Both search via the use of iterators for the range *[first,last)*.

[1] `http://www.cplusplus.com/reference/algorithm/find/`

[2] `http://www.cplusplus.com/reference/algorithm/binary_search/`

The procedure *std::find* is an linear[3] time complexity procedure. It is possible to use it for custom data, but we'll just assume data for which *operator==* is defined. Where v is the vector[4] containing our data and *key* is the value for which we're searching, then

[3] $O(n)$

[4] any STL container really

```
std::find(std::begin(v), std::end(v), key)
```

will return an iterator to the location where the first occurrence of *key* is found otherwise we get *std::end(v)*.

The procedure *std::binary_search* is a logarithmic[5] time complexity procedure. This makes it a whole order of magnitude faster than *std::find*[6]. The catch is that our data must first be sorted. By default, std::binary_search assumes data is sorted in least to greatest order. Thus,

[5] $O(\log n)$

[6] in the worst case of course

```
std::binary_search(std::begin(v), std::end(v), key)
```

will return true if *key* is in the vector/container v and false otherwise. If we want to search data sorted in greatest to least order then we can use the *functional* library's *std::greater* to specify the proper comparison to use in the search. For example, if v contains doubles in descending order, then

```
std::binary_search(std::begin(v), std::end(v), key, std::greater<double>() )
```

will search them for *key* using std::binary_search.

The underlying algorithms for std::find and std::binary_search are well know and classic examples of procedures in the linear and logarithmic complexity classes respectively. In what follows we'll write our own, constrained, versions and analyze them to gain insight in to the logical fingerprints of the respective complexity classes.

Linear Search

The algorithm underlying std::find is called LINEAR SEARCH. Let's begin with a version of linear search for vectors of integers in C++

and then walk through the complexity and logical analysis. Here is the documentation, declaration, tests, and full definition.

```
/**
  * Compute the location of the first occurence of the integer key
  * in the vector data.
  * @param data vector of integers
  * @param key search value
  * @return -1 if key is not found, otherwise the index where key
  * is first found
  * @pre none
  * @post none
  */
 int linsearch(const std::vector<int>& data,int key);

 TEST(linsearch,all){

   EXPECT_EQ(-1,linsearch(std::vector<int>({}),1));
   EXPECT_EQ(-1,linsearch(std::vector<int>({2}),1));
   EXPECT_EQ(0,linsearch(std::vector<int>({1}),1));
   EXPECT_EQ(1,linsearch(std::vector<int>({1,3,5}),3));
   EXPECT_EQ(0,linsearch(std::vector<int>({1,3,1}),1));
   EXPECT_EQ(2,linsearch(std::vector<int>({1,3,5}),5));

 }

 int linsearch(const std::vector<int>& data,int key){

   for(unsigned int i{0}; i < data.size() ; ++i){
     if( data[i] == key )
     return i;
   }
   return -1;
 }
```

Linear Search Complexity

All of the rules pertaining to Big-O that we listed in the previous lecture notes tell us that we can analyze procedures piece by piece, so let's start by looking at two pieces separately: the loop and the loop body. Seperating the act of repetition from the action being repeated is almost always a good way to start your analysis[7].

The body of the loop is a simple conditional. We check *data[i]==key* and either return or don't. For now, we ignore the effect of returning

[7] notice it's often how we start our design as well. this is not a coincidence

from within the loop and focus on everything else. All the operations carried out here are $O(1)$ or elementary operations regardless of the result of *data[i]==key*. Adding constant time chunks of work is still constant time so we can say that *the body of the loop carries out* $O(1)$ *operations every time it is executed.*

The *for* loop begins by executing the initialization code *unsigned int io*. This is just simple variable initialization and is therefore $O(1)$ work. Now, every time the loop executes the body it will also carry out the continuation check $i <$ *data.size()* and the update operation *++i*. These are all $O(1)$ operations. The only one we might give pause to is the vector *size* method. However, consulting the documentation will confirm that vectors can report their size in constant time. The loop will also carry out one more continuation check that terminates the loop. We don't know how many times the loop will actually loop yet, but if we assume it's some number k then we know the total work done by the loop is the sum of the initialization, the k loops, and the final continuation check that fails:

$$O(1) + k * O(1) + O(1) = O(k) + O(1)$$

The initialization and the final continuation check both contribute to an $O(1)$ term[8]. If k is some fixed constant for all vectors, then the whole thing becomes constant. If, however, it's some function of the vector size, then it's likely to be the dominant term. Let's see what happens.

[8] $O(1) + O(1) = O(2) = O(2 * 1) = O(1)$

This loop starts at 0 counts up in steps of size 1 to *data.size*() by steps of 1 and terminates when it reaches *data.size*(). If our *key* value is found, then the procedure terminates, which ends our loop early. If we don't find our key, then the loop runs to completion and it's clear then that the loop will execute exactly *data.size*() times. Let n be the size of the vector, then we have a complexity of $O(n) + O(1) = O(n)$. The constant term is the work we do outside the loop. The linear term is the work we do each time the loop body executes. It's important to remember that the $O(n)$ term comes from a loop that repeats $O(1)$ work n times.

Designing Linear Search

Linear search seems like something you come up with if you just apply the iterative recipe. Let's write up a template for iterative accumulation:

```
int linsearch(const std::vector<int>& data,int key){

   int fst_loc{...}
   for(unsigned int i{0}; i < data.size() ; ++i){
```

```
   ... key ... data[i]... fst_loc
    }
    return fst_loc;
  }
```

We can determine the value of *fst_loc* by decided what we should return for empty vectors: -1. It also seems clear that we need to build a conditional predicated on the equality of *key* and *data[i]*. That condition should, as usual, be responsible for updating the accumulator variable *fst_loc*.

```
int linsearch(const std::vector<int>& data,int key){

    int fst_loc{-1}
    for(unsigned int i{0}; i < data.size() ; ++i){
   if( key == data[i] ){
     fst_loc = ...;
   }
   else{
     fst_loc = ...;
   }
    }
    return fst_loc;
  }
```

The first thing we might notice is that the else condition can be dropped. If I didn't find the key value, then I don't really need to do anything. Now, we might think to assign *i* to *fst_loc* when *data[i]* is the same as *key*. However, doing so will mean that we get the last location containing key, not the first. To ensure that we get location of the first key value, we need to either ensure that *fst_loc* is -1[9] or we could just stop early. Let's just stop early as that clearly seems to be a more efficient choice. One way to do this is to work with the loop and add a second continuation condition.

[9] we haven't found key yet

```
int linsearch(const std::vector<int>& data,int key){

    int fst_loc{-1}
    for(unsigned int i{0}; i < data.size() && fst_loc == -1; ++i){
   if( key == data[i] ){
     fst_loc = i;
   }
    }
    return fst_loc;
  }
```

This is nice because it works with our existing template. However, it adds more work to the loop. Every time we execute the loop body we now do two comparisons: i to the data.size() and fst_loc to -1. The additional work is $O(1)$, so it doesn't change the complexity, but it is is more work. Rather than do more work, we can, in this case, safely return from the procedure[10]. This also negates the need for the accumulator variable.

[10] always be cautious when returning inside loops

```
int linsearch(const std::vector<int>& data,int key){

   for(unsigned int i{0}; i < data.size(); ++i){
  if( key == data[i] ){
    return i;
  }
   }
   return -1;
 }
```

Iteration and Linearity

It should be clear that our basic iterative logic template forces us into linear complexity procedures because this loop is, itself, linear:

```
for(unsigned int i{0}; i < data.size(); ++i){
  ...
}
```

Early termination doesn't change anything in terms of complexity. Let's say you could guarantee that you stop after doing $O(1)$ work with half the data in the vector? Then you'd be doing $O(\frac{n}{2}) = O(\frac{1}{2}n) = O(n)$ work still. The same thing happens if you count in steps. Counting by 2[11] means you visit half the data. Counting by 5 visits $\frac{n}{5}$ elements in the vector. Any fixed step size s will visit $O(\frac{n}{s})$ steps and if we're doing $O(1)$ work at each step then we're still stuck in the $O(n)$ complexity class. So, in the event that we can count in steps greater than 1[12] or terminate early, we should for practical reason. Just remember that larger steps and early termination do not break your loop out of the linear complexity class.

[11] i+=2

[12] this clearly isn't possible for linear search

Non-Linear Counting

Let's consider different ways of counting. More specifically, let's look at taking steps of variable size. What if we counted in steps that weren't fixed amounts but instead were sized as a function of the vector's size? For example,

```
unsigned int step = std::max(1,v.size()/5);
for(unsigned int i{0}; i < v.size(); i += v.size()/5){
 ...
}
```

We start by ensure that we have a non-zero step size. For vectors with a size less than 5, the step will be 1, otherwise we get steps equal to one fifth the vector size, rounded down. We're thinking about complexity, so we can ignore what happens with those small vectors and think about sizes greater than 5, maybe much greater. The step size will clearly get bigger as the vector gets bigger. If the size of the vector is 25 then we'll count through $0, 5, 10, 15, 20$ and terminate on 25. It the size were 100, we'd count in 20s. This loop is dangerous.For sufficiently large vectors, this loop always does 5 executions of the loop body. By stepping in increments that are proportional to the vector size we've created an $O(1)$ loop. While this is a good demonstration of how constant complexity loops can arise even when working with variable sized vectors, it's probably not something we'll run into often.

What if we vary the size of the step based on the previous step size? Imagine starting with a big step, then taking increasingly smaller steps. For example,

```
for(unsigned int i{1}; i <= v.size(); i *= 2){
  ...v[(i-1)]
}
```

Clearly something different is going on because we're counting by multiplying, not adding. Let's see what happens on some concrete vector sizes. For a vector of size 128 this loop would count through $1, 2, 4, 8, 16, 32, 64, 128$ and terminate on 256. This would visit locations $0, 1, 3, 7, 15, 31, 63$ and 127. What you might notice is that we're counting in powers of 2. If we replaced *i**=2 with *i**=10, then we'd count through $1, 10, 100, 1000, \ldots$, i.e. powers of 10. We can do the same thing counting down by dividing rather than multiplying.

```
for(unsigned int i{v.size()}; i > 0; i /= 2){
  ...v[(i-1)]
}
```

The question now is, how many times will this loop execute it's body for a vector of size n? Let's start by teasing out the pattern to our counting:

step	i
0	$1 = 2^0$
1	$1 * 2 = 2 = 2^1$
2	$1 * 2 * 2 = 4 = 2^2$
3	$1 * 2 * 2 * 2 = 8 = 2^3$
$\vdots$	$\vdots$
k	2^k

Let's just assume for a second that our vector size is exactly a power of two n. Now we can ask a different question: for what k is $2^k = n$? Why? Well this is exactly the last step of our loop and the number of time our loop body will execute! To get the k out of the exponent we need the inverse to the exponential. We need a function f such that $f(2^k) = k$. This function is the LOGARITHM function, base 2.

$$2^k = v.size() \rightarrow k = \log_2 n$$

If we were counting down we'd see the following pattern with a similar solution. Once again, if we assume the vector's size is a power of two:

step	i
0	$n = \frac{n}{2^0}$
1	$\frac{n}{2} = \frac{n}{2^1}$
2	$\frac{n}{4} = \frac{n}{2^2}$
3	$\frac{n}{8} = \frac{n}{2^3}$
$\vdots$	$\vdots$
k	$1 = \frac{n}{2^k}$

A little algebra later and we once again find that $k = \log_2 n$.

What happens if the size isn't a power of two? Well we know that the size, n, must be between two powers of 2, namely 2^k and 2^{k+1}. This means that $\log_2 n$ must be some real number between the integers k and $k+1$. The loop doesn't do partial evaluations, so either it executes the body k times or $k+1$ times. Given that 2^{k+1} is greater than the vector size, it must execute k times. This number, the largest integer smaller than $\log_2 n$, is what we get form the FLOOR function, i.e. by rounding down to the nearest integer[13].

[13] $k = \lfloor \log_2 n \rfloor$

Changing the multiple from 2 to something larger than two just changes the base of the logarithm, and in the world of Big-O, $O(log_a(n)) = O(log_b(n))$. This is a bit analogous to how changing the step size of our counted loop from 1 to some other fixed value didn't kick us out of the linear complexity class. The really important thing here is what kind of looping process leads to logarithmic loops. Fixed sized additive steps leads to linear time. Fixed sizes determined as

a portion of the vector size led to constant time loops. What these loops are doing is growing exponentially with each loop. By doubling or halving the current i value, rather than adding or subtracting to it, we've discovered a pattern for Logarithmic loops.

Before moving on let's generalize things a bit. We've been thinking in terms of steps and counting. A more recursive way of thinking about loop complexity is in terms of the size of the problem left unsolved. If I need to search through n items, then a step size of 1 leaves me with a space of $n - 1$ items to search. A step of size s leaves me with $n - s$. For this perspective, the question of complexity is one of determining the number of steps needed until there is no more vector data. What happens with or logarithmic loops? To put some context on this we should look at a classic example of logarithmic complexity: binary search.

Binary Search

Let us once again begin with the complete code.

```
/**
 * Compute the location of an occurence of the integer key
 * in the vector data.
 * @param data vector of integers
 * @param key search value
 * @return -1 if key is not found, otherwise the index where key
 * is first found
 * @pre data is sorted in greastest to least order
 * @post none
 */
int binsearch(const std::vector<int>& data,int key);

TEST(binsearch,all){

  EXPECT_EQ(-1,binsearch(std::vector<int>({}),1));
  EXPECT_EQ(-1,binsearch(std::vector<int>({2}),1));
  EXPECT_EQ(0,binsearch(std::vector<int>({1}),1));
  EXPECT_EQ(1,binsearch(std::vector<int>({5,3,1}),3));
  EXPECT_EQ(2,binsearch(std::vector<int>({5,3,1}),1));
  EXPECT_EQ(0,binsearch(std::vector<int>({5,3,1}),5));
  EXPECT_EQ(3,binsearch(std::vector<int>({5,4,3,2,1}),2));

}

int binsearch(const std::vector<int>& data,int key){
```

```
    int fst{0},mid{0};
int lst = data.size()-1;


    while ( fst < lst ){
      mid = (lst+fst+1)/2;
      if( data[mid] < key ){
    lst = mid-1;
      }
      else if( data[mid] > key ){
       fst = mid+1;
      }
      else{
      return mid;
      }
    }
    return -1;
  }
```

The Complexity of Binary Search

Let's begin with everything but the loop. Before and after the while loop we clearly do some constant, $O(1)$ complexity, work regardless of the vector's size. Inside the loop we do $O(1)$ work to compute the value of *mid* and then carry out a conditional. If we examine each branch of the conditional we notice that only constant complexity, elementary operations are ever carried out. This means that the worst case must be $O(1)$ because all cases are $O(1)$.[14] The while loop carries out one $O(1)$ continuation check every time it executes the body of the loop and one more to terminate the loop. Putting this together we see $O(1)$ work outside the loop and $O(1)$ work every time the loop body executes. Once again, for a loop that executes k times, we're looking at $O(1) + O(k)$ total complexity.

Understanding how many times this loop will execute is trickier than our simple additive counting loops. For starters, it's not clear how this drives to termination. Let's start by looking at mid. A little bit of algebra shows us that this is the index of the point in the

[14] In practice the worst case is probably when the second condition is true. This means we evaluate the first condition $data[mid] < key$ and notice its false, then evaluate the second condition $data[mid] > key$ see that it's true, and finally compute the new fst value.

middle of *[fst,lst]*.

$$\begin{aligned} mid &= \frac{lst - fst + 1}{2} + fst \\ &= \frac{lst - fst + 1 + 2 * fst}{2} \\ &= \frac{lst + fst + 1}{/} 2 \end{aligned}$$

So what's happening? Lets fix some values and look at the pattern knowing that every loop starts by setting mid to the point half-way between fst and lst. If we assume that $data[mid] > key$, then fst and mid change where lst stays the same. Let's assume a vector of size 128 and look at the relationship between fst,mid,lst, and the size the step we take from one iteration to the next.

step number	fst	mid	lst	step size
0	0	0	127	NA
1	0	63	127	64
2	63	95	127	32
3	95	111	127	16
4	111	119	127	8
5	119	123	127	4
6	123	125	127	2
7	125	126	127	1
8	126	127	127	1
9	127	127	127	0

The pattern is pretty clear from the step size, we're halving the step size with each step. We've already established that this is a fingerprint for logarithmic loops. As this is the worst case for binary search, we can now determine that for vectors of size n, the loop will execute $O(\log n)$ times and the total complexity is $O(\log n)$. It's worth pointing out that when we flip flop between $data[mid] > key$ and $data[mid] < key$ the absolute value of the step size continues on the logarithmic path. To better understand why this happens we should look at how one might come-up with this search in the first place.

The Design of Binary Search

I find this loop based version to less easy to understand then the basic recursive picture of binary search. Here's some pseudo-code that states binary search recursively. Selecting with a single index like *v[k]* is single element selection. When two indexes are separated by two periods like *v[k..j]*, then we're selecting the region of vector v that starts at k and ends at j. If $j < k$ for some reason, then assume

that region is empty. Let v be a vector that's sorted in greatest to least order and k be the key value we're searching for within v.

```
binary_search( v, k )

  if v is empty
     return -1
  else
    mid = size(v)/2
    if( v[mid] == k )
      return mid
    else if( v[mid] < k )
      return binary_search(v[mid+1..size(v)-1],k)
    else
      return binary_search(v[0..mid-1],k)

end binary_search
```

First we check to see if the mid-point of the vector is the item we're looking for[15]. If this is not the key value, then because the vector is sorted, we'll know which half of the vector contains the key. If the key is greater (less) than the mid point value, then if its in the vector it must be before (after) the mid point.We can therefore restrict out next binary search to that *half* of the original data. The key is that each recursive call deals with portion of the data that is half the size of the previous portion.

[15] in our loop version we check the other two cases first for efficiency reasons

Our loop based version manages the selection of a half iteratively by modifying the current starting and ending pointers to region being search as needed. If the item is to the left of the mid, then we move *lst* to just to the left of *mid*. If it's to the right of mid, then we move *fst* to the right of *mid*. This effectively selects one half or another of the previous region of the vector v.

Our previous encounters with recursion worked by recursing on a simple first/rest structure in the data. This recursion on half marks a new way to recursively decompose our data. The vector is either empty or it's the vector $v[fst..mid-1]$ followed by the single element $v[mid]$, and finally by the vector $v[mid+1..lst]$. In the case of binary search, it's only necessary to recurse on one of the sub-vectors. When we look at sorting, we'll use this same strategy to develop a sorting algorithm.

This strategy is so pivotal that it has it's own name Divide and Conquer. With divide and conquer algorithms we recursively solve the problem on vectors that are a fraction of the size of the original input and then combine those solutions. For binary search we invert

this, first check a singular, non-recursive element, then recurse on a vector that is a fraction of the original size. The key here is that the repeated reduction of the vector size generates the same kind of logarithmic behaviors we saw when exploring non-linear counting.

Binary Search all the Data

Now you might think that the order of magnitude improvement we gain from binary search versus linear search means we should always use binary search. However, we should not forget that our ability to employ this strategy came at the cost of a pretty serious precondition: our vector must be sorted. For binary search to negate the need for linear search then we must be able to sort with $O(\log n)$ complexity. If the complexity of sorting is some class $O(f(n))$ that's greater than $O(\log n)$, then we know that the total complexity, $O(f(n)) + O(\log n)$, would be the dominant $O(f(n))$ term. Sorting in logarithmic time seems highly unlikely. We'd have either do $O(\log n)$ work relative to a single element or $O(1)$ work on $O(\log n)$ elements. Either way, we pretty much preventing ourselves from even looking at all the data let alone manipulating it.

What if we're doing $O(n)$ searches? If we stick with linear search then the total cost for all the searching would be $O(n^2)$. The cost of $O(n)$ binary searches, on the other hand, is only $O(n * \log n)$[16]. So, if we can sort in $O(n * \log n)$ complexity and then do all $O(n)$ of our searches without resorting to a resort, then we'll actually be the complexity of linear searching. A quick look at the C++ *std::sort* documentation tells us that we can, in fact, sort in $O(n * \log n)$ complexity.

On thing to take away from this discussion is to watch your preconditions. It's not uncommon to have optimized procedures for special cases of a larger problem just like we have binary search to solve search with sorted data. The other thing to notice is how we used the basic Big-O theorems to better understand our algorithms in a different context and to help us decide on an efficient course of action when doing multiple searches. It's this ability to focus on the big picture before you dive into the implementation that really makes Big-O and complexity an indispensable tool for the working programmer.

[16] note that if locations in the original vector are a concern, then you'll need to build in a second vector to track those values

Recap

Understanding linear and binary search takes us a long way to understanding the linear and logarithmic complexity classes. The real insight comes from understanding how the traverse along the vector.

Linear search moves in simple fixed size steps from one iteration of the loop to the next. This pattern, regardless of step size, is going to induce worst case linear complexity at best. On the other hand, if the step size varies with the vector size, but is fixed from one iteration to the next, then we end up with constant complexity loops. Unfortunately, we haven't seen a practical use for this strategy yet.

The key to breaking the linearity barrier was varying the step size from one iteration in an exponentially growing or decreasing fashion. In the case of binary search we start with a large step to the middle of the vector and then take exponentially decreasing smaller steps[17]. Another way to understand this analysis is from the effect on the amount of data left to work with after the current iteration. Once you check the mid, you'll be able to reduce the search space to half of the current space.

[17] $\frac{n}{2^k} \rightarrow \frac{n}{2^{k+1}}$