CSI 1430
Week 12 Resource

Colin Burdine

4/04/21

Major Topics:

1. Searching
2. Sorting

Keywords: *searching, sorting, bubblesort, binary search*

Reminder: Group tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CDT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Searching

Last week we discussed Objects and how they are useful in aggregating multiple data types and functions into a single, independent entity. We also discussed some of the streams that C++ provides in the standard library for I/O operations. This week we will be pivoting our discussion to talk about something a bit more abstract- *algorithms*. An algorithm is simply a scalable step-by-step procedure we can follow to perform some computation in an input of any size. This week we will be discussing *searching*, a process by which we can find a data entry in some large collection of data. We will also show how searching can be made much easier through *sorting* data.

Linear Searching

Suppose we have the list of unordered values:

```
{ 2, 33, -4, 5, 20, 33, 40, -20, 7, 43 }
```

If we want to find the index of a particular value (if it exists) or otherwise determine if a particular value is not in the array, we can do this quite easily with a for loop. This method of finding the index of a value in an unordered list is called a *linear search*, and we can do it in C++ with the following code:

```
int linearSearch(int[] list, int listLength, int value){

    // traverse the list sequentially:
    for(int i = 0; i < listLength ; ++i){

        // check if we have found the value:
        if(list[i] == value){
            return i;
        }
    }

    // if value is not found, return -1:
    return -1;
}
```

Binary Searching

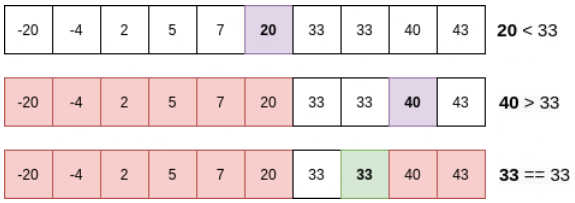
Now, suppose we have the same list of number, but the list of numbers has been sorted for us in ascending value:

```
{ -20, -4, 2, 5, 7, 20, 33, 33, 40, 43 }
```

Notice that in the worst case scenario for our linear search, the value we are looking for doesn't exist in the list, which means for a list of size n we would need to traverse all n elements in the list. If our list happens to be sorted, is there a faster way by which we can find the value we are looking for. As it turns out, we can use the method of *binary search*. Binary search only works if our list is sorted; in general you can think of it like a "guess the number game" where our value is the number we are trying to guess the position of (if it even exists in the list). Our initial position guess will be that the number is in the center of the sorted list. If we do not find the number there, we ask *is the value greater than or less than the value in the center?* If the value we are looking for is greater, we can know certain that the index of the value cannot be in the lower half of the list. Otherwise, if our value is less than the value in the center, we know that the index of our value cannot be in the upper half of the list. In either case, we eliminate *half* of the total indexes in the list

that we have to search. In fact, if we repeat this process by picking a new "center" to be the center of the set of indexes we have not yet eliminated, we can continue this process until we either find the index of the value we are looking for or we eliminate all indexes in the list.

To provide a concrete example, suppose we want to find the index of the value 33 in the list above. (note there are two possible indexes that would both work. We observe that because the list is sorted, we can do this with *only three* comparisons! In fact, for a list of size n we need only approximately $\log_2(n)$ comparisons, while our linear searching method required us to search up to the entire list (n comparisons):



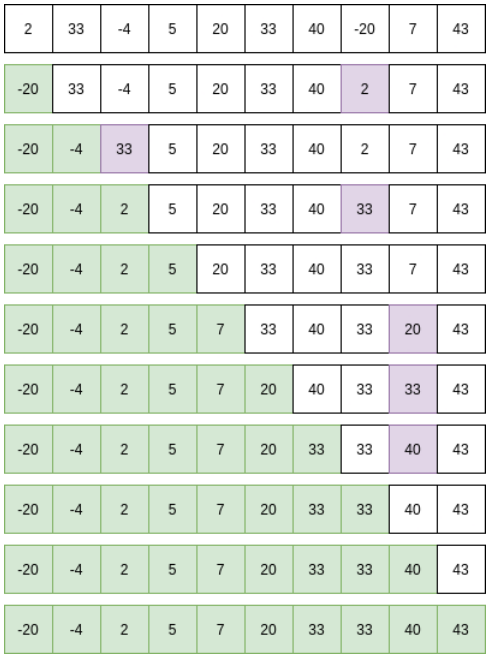
For the code the perform binary search, see the programming exercise below.

1.2 Sorting

We just discussed that sorting a list prior to performing searches gives us significant time savings when we are trying to look up a value (in fact, this is how most database systems work). This week we will discuss two basic sorting algorithms that you can use to sort data- *Selection Sort* and *Bubble sort*.

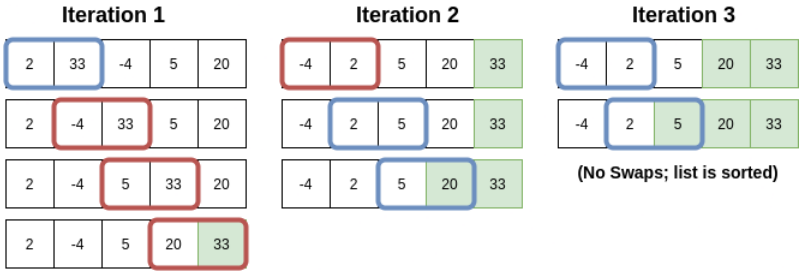
Selection Sort

Selection sort works by taking a list of values and builds the sorted list from left to right. It does this by dividing up the list into two sub-lists, a sorted sub-list (on the left) and an unsorted sub-list (on the right). On each iteration of selection sort, the smallest element is found in the unsorted sub-list, and is swapped with the leftmost element in the unsorted sub-list. Since the selected element that we swapped must the next element in the sorted order of the entire list, the size of the sorted sub-list increases by 1, while the size of the unsorted sub-list decreases by 1. An example of this algorithm in action is given below:



Bubble Sort

The second sorting method we will discuss is *bubble sort*, which is similar to insertion sort but does two things differently. First, bubble sort sorts the list from right to left, instead of left to right. Second, bubble sort works by swapping any two adjacent elements if they are out of order in the unsorted portion of the list. This has the effect (take a moment to think *why*) of moving the maximal element to the rightmost position in the list, much like insertion sort moves the minimal element to the leftmost position of the unsorted sub-list. However, bubble sort also has the benefit of knowing when the unsorted sub-list is sorted; that is, *if no swaps are made while traversing the unsorted sub-list, we know that the entire list must be sorted*. This means that is some cases, we can "quit early" because we know when the list is sorted. Consider the example below, in which we sort the a list of five elements:



2 Programming Example Problem

To illustrate how we can use searching and sorting in C++, consider the following example programming problem:

Create a header and C++ source file called `vectorUtil.h/.cpp` with a function called `bubbleSortList` that sorts a vector of elements and a function called `searchSortedList` that binary searches for a given integer value. o keep things simple, we will assume that the vectors are storing integers.

The declaration of your two functions should look like:

```
// bubble sorts a list in ascending order
void bubbleSortList(vector<int>& list);

// returns an index of the sorted list
// where the indicated value can be found. If
// the value is not in the list, -1 is returned
int searchSortedList(vector<int>& sortedList, int value);
```

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week.

3 Closing Remarks

I hope that the resources I shared with you this semester are helpful and efficient in addressing your academic needs. As we approach the end of the fall semester, I would like to let you know that all resources for the course content this semester can be found here: https://www.baylor.edu/support_programs/index.php?id=967950

Group tutoring sessions for this course will be held online every Monday evening from 7:00pm to 8:00pm CT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

```h
#ifndef VECTORUTIL_H
#define VECTORUTIL_H
#include <vector>
using namespace std;

void bubbleSortList(vector<int>& list);
int searchSortedList(vector<int>& sortedList, int value);
#endif /* VECTORUTIL_H */
```

```cpp
#include <vector>
#include "vectorUtil.h"
using namespace std;

void bubbleSortList(vector<int>& list){
    bool swp = true;
    for(int i = 0; i < list.size() && swp; ++i){
        // traverse list and swap out-of-order pairs:
        swp = false;
        for(int j = 0; j < (list.size()-(i+1)); ++j){
            // swap each out-of-order pair:
            if(list[j] > list[j+1]){
                swap(list[j],list[j+1]);
                swp = true;
            }
        }
    }
}

int searchSortedList(vector<int>& sortedList, int value){
    // initialize left and right pointers:
    int l = 0, r = sortedList.size()-1;

    // continue searching until all possible
    // elements in the sorted list are eliminated:
    while(l <= r){
        // check the center of the current sub-list:
        int center = (l+r)/2;
        if(sortedList[center] < value){
            l = center+1;
        } else if (value < sortedList[center]){
            r = center-1;
        } else {
            // return center index if a match is found:
            return center;
        }
    }
    return -1;
}
```