Edinburgh Napier University

SET09117 Algorithms & Data Structures

Lab 6 - Searching

Dr Simon Wells

1 Aims

At the end of the practical portion of this topic you will be able to:

- Implement a variety of basic search algorithms in a language of your choice.
- Implement a *test harness* for your algorithms
- Create a plot of your algorithm's performance on datasets of various size
- Consider the factors that might affect the performance of an algorithm
- Understand the circumstances that lead to the best, worst, and average performance charateristics for your algorithms

2 Activities

2.1 Implement a linear search algorithm

The Linear search, also known as a sequential search, algorithm moves through your dataset, starting with the first element, and stepping through, examining one element at a time. At each step the element is compared to the search term (or key), if they match then the seach is complete as you've found your item, otherwise you must continue until you find a match or reach the end of your data.

This is the only kind of search that can be performed on unsorted data. If you don't know the order of your data, or your data has no order, then there is no alternative but to check every item to see if it is what you're looking for. In the worst case this means that you have to check every member of the collection before determining whether the item is or is not in the collection.

Think of finding a particular card in a deck of playing cards, the Queen of Hearts for instance. If the deck has been properly shuffled then, unless you are Ricky Jay[1] who could find a single card anywhere in a deck, you're probably going to have to do a linear search, check each card as rapidly as you can until your find your target.

The Linear search *algorithm* is as follows (given a collection of unsorted/unorered data and a search term or key):

1. Compare the key to the first element of data
2. If key matches element then the search is successful
3. If there are more elements of data: get the next element of data & return to the previous step
4. If there is no more data & no matching key so search the the search is unsuccessful

We actually have already written linear searches many of the earlier labs, look back through your code for any function that you've written that takes a particular value to look for then tells you where it is in the collection. Make sure that you understand the process that we've followed and how our earlier code implements the algorithm above.

We are going to concentrate on seaching for values within arrays so that we can work with a simple and straightforward environment rather than applying our learning in lots of contexts. This will be useful over the next couple of weeks as we study various ways to sort data so that it can be searched more efficiently.

[1] An awesome magician whose legerdemain was particularly amazing when doing card manipulation `https://en.wikipedia.org/wiki/Ricky_Jay`.

Our search function from the arrays parts of Lab 03 looked like this[2]:

```
void search(int* array, int num)
{
    int idx;
    for(idx=0; idx<MAX; idx++)
    {
        if(array[idx] == num)
        {
            printf("%d found in position %d\n", num, idx+1);
            return;
        }
    }
    if(idx == MAX)
        printf("%d not found in array\n", num);
}
```

Now retrieve your code from lab 02, the program you wrote to time things, You're going to use that code to investigate how linear search performs. It should look something like this[3]:

```
#include <time.h>
#include <stdio.h>

void code()
{
    for(int i=0; i<10000; i++)
    {
        printf(".");
    }
    printf("\n");
}

int main()
{
    clock_t t;
    printf("start: %d \n", (int) (t=clock()));

    code();

    printf("stop: %d \n", (int) (t=clock()-t));
    printf("Elapsed: %f seconds\n", (double) t / CLOCKS_PER_SEC);

    return 0;
}
```

Your first task is to implement an integer array in the main function, populate the array with random values, then run the linear search against it looking for a particular value. You'll have to call the linearsearch function from within *code()* because we want to measure how long it takes. Repeat this a number of times, say 100, keeping track of the times, then work out the average time for the search. Now increase the size of your array and repeat the procedure the same number of times, keeping track of the timings and working out an average at the end. You can either do this manually, or else, get your program to automate this[4]It's also worth keeping a count of the number of comparisons performed as this makes a nice proxy for absolute timings. Consider why each might be a useful value to have when doing analysis of the performance of an algorithm.

Now repeat this several more time, increasing the number of items in the collection at each repetition. You're welcome to see just how large you can make the array before your program chokes[5]. Finally, take the figures from your experiments and plot them to draw a chart. Excel, or any other spreadsheet is a useful tool for this[6]. The x-axis should record the size of the collection, and the y-axis the time taken to search for a value. What do you notice about the shape of the plot? Does the search implementation behave as you'd expect it to? You might want to repeat the experiments twice, once doing a search where you are finding an element that does exist in the

[2]Refresh your understanding by re-doing parts of that lab if necessary before proceeding
[3]Although I did suggest some enhancements that you could make to make your own life easier at this point
[4]As programmers this is exactly the kind of thing that we should be automating.
[5]Which might be interesting to do in it's own right.
[6]Those of you who do Python have no need of ever using a spreadsheet again, you just need to find the right Python libs for this particular type of analysis

collection, and once where the element is not in the collection. Under what circumstance do you appear to get the best, worst, and average performances?

Consider repeating this same experiment using various of other data structures that we've constructed. This should given you an insight into the relative performance of a linear search across various data structures[7].

2.2 Implement a binary search algorithm

This algorithm still has to do comparisons but is more efficient than a linear search because, after each comparison, half the search space is subsequently ignored. This greatly reduces the potential number of comparisons that need to be performed.

If we return to our card deck again, but this time, if the card deck is brand new then the cards are ordered, first by suit, and within each suit from the two to the Ace. It is straightforward to cut the deck, decide whether the card at the cut is higher or lower in the ordering and decide which half of the remaining cards will contain our lady. We can repeat this until we are left holding the Queen of Hearts. This, in essence is the binary search. We rely on the data being ordered, so that we know roughly where to search. Even in the worst case scenario we would have to do many fewer comparison than for the linear search.¡D-d¿

The Binary search algorithm[8] is as follows:

1. Compare search key to middle element of the data:
2. If the middle element matches the search key then the search is successful & complete
3. If the middle element is > search key then discard the larger half of the data & return to step #1
4. If the middle element is < search key then discard the smaller half of the data & return to step #1
5. If no more data & no matching key so search is unsuccessful & complete

Let's look at an implementation of this algorithm:

```
1  #include <stdio.h>
2
3  #define SIZE 10
4  int main(void)
5  {
6      int arr[SIZE] = {1, 2, 3, 9, 11, 13, 17, 25, 57, 90};
7      int mid = 0, lower = 0, upper = SIZE-1, key = 0, found = 0;
8
9      printf("Enter number to search for: ");
10     scanf("%d", &key);
11
12     for(mid=(lower+upper)/2; lower <= upper; mid=(lower+upper)/2)
13     {
14         if(arr[mid] == key)
15         {
16             printf("Your number is at position %d of our array\n", mid);
17             found = 1;
18             break;
19         }
20         if(arr[mid] > key)
21             upper = mid-1;
22         else
23             lower = mid+1;
24     }
25     if(!found)
26         printf("%d is not in the array\n", key);
27         return 0;
28     return(0);
29 }
```

[7] Trees & graphs have their own specialist construction, search, and traversal algorithms which we'll see in a few weeks so don't worry about them just yet

[8] If this description is insufficient then this is an opportunity to do some research for a description that works for you. Sometimes a different voice or use of words can make things more understandable

We've created and populated a small array with integer values that are in ascending numerical sorted order[9]. We then ask our user for a number to look for. To look for it, we calculate the mid-point of the array, check the value, if it's correct then we can print a message and exit. Otherwise, we adjust our upper or lower bounds depending upon whether our search term is larger or smaller than the value at the mid point. We then repeat the search, examining the mid point of the remaining half of the array, and discarding the half that doesn't contain our search term. Eventually we shall either find our search term or else run out of numbers to check.

Repeat the experiment that you ran for the linear search, but now using the binary search. You will need to adjust our binary search program to work with our timing program. Plot the results on the same chart that has your linear search and compare the results. What do you notice? Which algorithm has the steepest curve? Why do you think this?

2.3 Challenge: Other Search Algorithms

If you think you're happy with your understanding of the previous two search algorithms then you should research and consider the following (less common) search algorithms. Again they all rely on sorted data and behave with slightly different characteristics:

- Jump Search
- Interpolation Search
- Exponential Search
- Ternary Search

Find a description of the algorithm for each, implement a method to use that search strategy on your array, then run the function in your timing program and plot the results.

The key point to take away is that the data can be as important as the algorithm when it comes to real world performance. If you know your data well, and have carefully considered whether your data fits the case, then you can, perhaps, get a slight edge in performance.

2.4 Coursework

Finally, if you're happy with your understanding of search algorithms then feel free to use the lab time to work on your coursework assignment.

[9] In a real program we would either construct the array as a sorted collection or else run a sorting algorithm, but that is for next week, so, for now, let's assume that our array is magically sorted ready for us to search it