8.1 Sorting

It is common to search through an array to find a certain value.

Search: locate an item in a list of information

Two algorithms we will examine:

- Linear search
- Binary search

What do we want to return?
- The value we are looking for
- An indication that it was found or not
- The location where it was found

If we want to create a function that will search through an array, what do we need to send it?

- The array
- The size
- The value we are looking for

What do we want it to return?
- Something that indicates whether it was found or not
- Its location in the array
- The index where it was found

The last two can be the same thing: the index is its location in the array.

If we want to create a function that will search through an array, steps are:

1. Starting at the first element, the algorithm sequentially steps through the array examining each element until it locates the value it is searching for.
2. Also called the sequential search.

A Linear Search Function

```c
int searchList(int list[], int numElems, int value)
{
    for (int i=0; i<numElems; i++)
        if (list[i] == value)
            return i; // return position i if found
    return -1; // return -1 if not found
}
```

Linear Search - Example

```
17 23 5 11 2 29 3
```

Array numList contains:

- 17
- 23
- 5
- 11
- 2
- 29
- 3

Linear Search - Example

- Linear search
- Binary search
- Linear search

- Two algorithms we will examine:
  - Search: locate an item in a list of information
  - Find a certain value

- It is common to search through an array to
Linear Search

• Benefits:
  – Easy algorithm to understand
  – Array can be in any order

• Disadvantages:
  – Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array. For example if you have 250,000 elements, then the average number of comparisons is 125,000.

Binary Search

• Another type of search is the binary search.

1. Divides the array into three sections:
   – middle element
   – elements on one side of the middle element
   – elements on the other side of the middle element

2. If the middle element is the correct value, done.

3. Otherwise, go to step 1. using only the half of the array that may contain the correct value.

4. Continue steps 1. and 2. until the value is found or there are no more elements to examine.

How do we find the middle?

(firstIndex+lastIndex)/2

Look at the element at the middle index
– if it is the value – return the index
– otherwise - decide which part of the array to look at

1. < look left – move lastIndex to middle-1
2. > look right – move firstIndex to middle+1

Continue steps 1. and 2. until the lastIndex is smaller than the firstIndex.

A Binary Search Function

```c
int binarySearch(int array[], int size, int value)
{
    int first = 0,             // First array element
        last = size - 1,       // Last array element
        middle;                // Mid point of search

    while ( first <= last)
    {
        middle = (first + last) / 2;     // Calculate mid point

        if (array[middle] == value)      // If value is found at mid
            return middle;
        else if (array[middle] > value)  // If value is in lower half
            last = middle - 1;
        else                              // If value is in upper half
            first = middle + 1;
    }

    return -1;                   // Return -1 if value not found.
}
```

Binary Search

- If we want to create a function that will search through an array, what do we need to send it?
  • the array
  • the size
  • the value we are looking for

- What do we want it to return?
  – something that indicates whether it was found or not
  – its location in the array

- The last two can be the same thing, the index where it was found or –1 if it was not found.

The last two can be the same thing, the index

- is location in the array
- something that indicates whether it was found or not

What do we want it to return?
- the value we are looking for
- the size
- the array

- The array

Binary Search Example

<table>
<thead>
<tr>
<th>Array</th>
<th>value</th>
<th>outcomes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 5 11 17 23 27 33 40</td>
<td>17</td>
<td>found</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>found</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>found</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>not found</td>
</tr>
</tbody>
</table>

This way our search is cut in half each time.

- There are no more elements to examine
- Continue steps 1. and 2. until either the value is found or there are no more elements to examine
- Otherwise stop if step 1. is true
- Move the mid element to the middle of the array
- Repeat steps 1. to 3. until step 1. is true

Disadvantages:
- Array can be in any order
- No guarantee to understand elements

Benefits:
- N/2 elements for the array N elements
- Comparisons is 125,000
- Have 250,000 elements, then the average number of elements for value not in array.

Another type of search is the binary search.

Linear Search

• Benefits:
  – Easy algorithm to understand
  – Array can be in any order

• Disadvantages:
  – Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array. For example if you have 250,000 elements, then the average number of comparisons is 125,000.
### Binary Search - Tradeoffs

**Benefits:**
- Much more efficient than linear search. For array of N elements, performs at most \( \log_2 N \) comparisons.
- If you have 250,000 elements to look through, it would take a maximum of 15 compares.

**Disadvantages:**
- Requires that array elements be sorted, this is not trivial.

---

### Sorting

**8.3 Sorting**

- **Sort**: arrange values into an order:
  - Alphabetical
  - Ascending numeric
  - Descending numeric

**Two algorithms considered here:**
- **Bubble sort**
- **Selection sort**

**Requirements:**
- Demonstration of selection sort
- Understanding how the data is moving

**Concept:**
- Repeal until pass made with no exchanges
- Pass through array again, exchanging as necessary
- Sort until end
- Compare 1st two elements
  - If out of order, exchange them to put in order
- Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end
- Repeat until pass made with no exchanges

**Example – First Pass**

<table>
<thead>
<tr>
<th>Array numlist3 contains:</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 23 5 11</td>
</tr>
</tbody>
</table>

- **Example – Second Pass**

<table>
<thead>
<tr>
<th>Array numlist3 contains:</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 5 11 23</td>
</tr>
</tbody>
</table>

---

### Comparison

<table>
<thead>
<tr>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average of 500 comparisons</td>
<td>(1000) comparisons</td>
</tr>
<tr>
<td>Average of 2,000 elements</td>
<td>(5000) elements</td>
</tr>
<tr>
<td>Max of 10 comparisons</td>
<td>Max of 10 comparisons</td>
</tr>
</tbody>
</table>

**Benefits:**
- Much more efficient than linear search, for any array of N elements.

**Disadvantages:**
- Requires that array elements be sorted. This is not trivial.
After second pass, array numlist3 contains:

5 11 17 23

- In correct order, no exchange

Next pass:

5 11 17 23

- In correct order, no exchange

No exchanges, so array is in order

Bubble Sort Function

```c
void bubbleSort(int array[], int size)
{
    for (int pass = 1; pass < size; pass++)
        for (int i = 0; i < size - pass; i++)
            if (array[i] > array[i + 1])
                swap(array[i], array[i + 1]);
}
```

- Benefit: Easy to understand and implement
- Disadvantage: Inefficient, slow for large arrays

Add Optimization 1

```c
void bubbleSort(int array[], int size)
{
    for (int pass = 1; pass < size; pass++)
        for (int i = 0; i < size - pass; i++)
            if (array[i] > array[i + 1])
                swap(array[i], array[i + 1]);
}
```

Add Optimization 2

```c
bool didSwap = true;
for (int pass = 1; pass < size && didSwap; pass++)
{
    didSwap = false;
    for (int i = 0; i < size - pass; i++)
    {
        if (array[i] > array[i + 1])
        {
            swap(array[i], array[i + 1]);
            didSwap = true;
        }
    }
}
```

- Benefit: Easy to understand and implement
- Disadvantage: Inefficient, slow for large arrays

Example – Third Pass

Array is in order after two passes.
Selection Sort

- Concept for sort in ascending order:
  - Locate smallest element in array. Exchange it with element in position 0.
  - Continue until all elements are arranged in order.

Example (Continued)

2. Next smallest element is 3. Exchange 3 with element in position 1 in array:

   3 2 29 11

Example (Continued)

3. Next smallest element is 11. Exchange 11 with element in position 2 in array:

   2 3 11 29

Selection Sort

- still use the swap function
- also need a find minimum function
- returns index of where the minimum value is in array
- has a third parameter that identifies the starting place in the array

Selection Sort

```c
int minimum(int array[], int size, int start)
{
    int min = array[start];
    int minIndex = start;
    for (int index = start; index < size; index++)
    {
        if (array[index] < min)
        {
            min = array[index];
            minIndex = index;
        }
    }
    return minIndex;
}
```

Selection Sort

```c
void selectionSort(int array[], int size)
{
    int start = 0;
    int minIndex;
    for (int pass = 1; pass < size; pass++) // size - 1 passes
    {
        minIndex = minimum(array, size, start);
        swap(array[start], array[minIndex]);
        start++;
    }
}
```

Selection Sort

```c
void swap(int& num1, int& num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```
Selection Sort

• **Benefit:**
  – More efficient than Bubble Sort, since fewer exchanges

• **Disadvantage:**
  – May not be as easy as Bubble Sort to understand