Definitions of Search and Sort

- **Search**: find a given item in an array, return the index of the item, or -1 if not found.
- **Sort**: rearrange the items in an array into some order (smallest to biggest, alphabetical order, etc.).
- There are various methods (algorithms) for carrying out these common tasks.
- Which ones are better? Why?

Linear Search

- Very simple method.
- Compare first element to target value, if not found then compare second element to target value . . .
- Repeat until: target value is found (return its index) or we run out of items (return -1).

Linear Search in C++ first attempt

```c++
int searchList (int list[], int size, int target) {
    int position = -1;           //position of target
    for (int i=0; i<size; i++)
    {
        if (list[i] == target)    //found the target!
            position = i;        //record which item
    }
    return position;
}
```

Is this algorithm correct (does it calculate the right value)?

Is this algorithm efficient (does it do unnecessary work)?
Linear Search in C++
second attempt

```cpp
int searchList (int list[], int size, int target) {
    int position = -1;  // position of target
    bool found = false;  // flag, true when target is found
    for (int i=0; i < size && !found; i++)
    {
        if (list[i] == target)  // found the target!
        {
            found = true;       // set the flag
            position = i;       // record which item
        }
    }
    return position;
}
```

Is this algorithm correct (does it calculate the right value)?

Is this algorithm efficient (does it do unnecessary work)?

Program that uses linear search

```cpp
#include <iostream>
using namespace std;

int searchList(int[], int, int);

int main() {
    const int SIZE=5;
    int idNums[SIZE] = {871, 750, 988, 100, 822};
    int results, id;
    cout << "Enter the employee ID to search for: ";
    cin >> id;
    results = searchList(idNums, SIZE, id);
    if (results == -1) {
        cout << "That id number is not registered\n";
    } else {
        cout << "That id number is found at location ";
        cout << results+1 << endl;
    }
}
```

Evaluating the Algorithm

- Does it do any unnecessary work?
- Is it time efficient? How would we know?
- We measure time efficiency of algorithms in terms of number of main steps required to finish.
- For search algorithms, the main step is comparing an array element to the target value.
- Number of steps depends on:
  - size of input array
  - whether or not value is in array
  - where the value is in the array

Efficiency of Linear Search

how many main steps (comparisons to target)?

<table>
<thead>
<tr>
<th></th>
<th>N=50,000</th>
<th>In terms of N</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best Case:</strong></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Average Case:</strong></td>
<td>25,000</td>
<td>N/2</td>
</tr>
<tr>
<td><strong>Worst Case:</strong></td>
<td>50,000</td>
<td>N</td>
</tr>
</tbody>
</table>

Note: if we search for many items that are not in the array, the average case will be greater than N/2.
Binary Search

- Works only for SORTED arrays
- Divide and conquer style algorithm
- Compare target value to middle element in list.
  - if equal, then return its index
  - if less than middle element, repeat the search in the first half of list
  - if greater than middle element, repeat the search in last half of list
- If current search list is narrowed down to 0 elements, return -1

Binary Search Algorithm example

We use first and last to indicate beginning and end of current search list

<table>
<thead>
<tr>
<th>target</th>
<th>first</th>
<th>mid</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>is 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt; 50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list</td>
<td>2 4 7 10 11 45</td>
<td>50</td>
<td>59 60 66 69 70 79</td>
</tr>
<tr>
<td>target &gt; 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list</td>
<td>2 4 7 10 11 45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>target == 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list</td>
<td>10 11 45</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Binary Search in C++

```cpp
int binarySearch (int array[], int size, int target) {
    int first = 0,        //index of beginning of search list
    last = size - 1,     //index of end of search list
    middle,              //index of midpoint of search list
    position = -1;       //position of target value
    bool found = false;  //flag
    while (first <= last && !found) {
        middle = (first + last) /2;    //calculate midpoint
        if (array[middle] == target) {
            found = true;
            position = middle;
        } else if (target < array[middle]) {
            last = middle - 1;    //search list = lower half
        } else {
            first = middle + 1;  //search list = upper half
            position = middle;
        }
    }
    return position;
}
```

Program using Binary Search

```
#include <iostream>
using namespace std;

int binarySearch(int[], int, int);

int main() {
    const int SIZE=5;
    int arr[SIZE] = {100, 750, 822, 871, 988};
    int results, id;
    cout << "Enter the employee ID to search for: " ;
    cin >> id;
    results = binarySearch(arr, SIZE, id);
    if (results == -1) {
        cout << "That id number is not registered\n" ;
    } else {
        cout << "That id number is found at location ";
        cout << results+1 << endl;
    }
}
```

What if first + last is even?
What if first + last is odd?
Efficiency of Binary Search

Calculate worst case (target not in list) for N=1024

<table>
<thead>
<tr>
<th># Items left to search</th>
<th># Comparisons so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
</tr>
<tr>
<td>128</td>
<td>3</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

1024 = $2^{10}$ $\iff$ $\log_2 1024 = 10$

Goal: calculate this value from N

Is $\log_2 N$ better than $N$?

Is binary search better than linear search?

Compare values of $N/2$, $N$, and $\log_2 N$ as $N$ increases:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N/2$</th>
<th>$\log_2 N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2.5</td>
<td>2.3</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>5.6</td>
</tr>
<tr>
<td>500</td>
<td>250</td>
<td>9</td>
</tr>
<tr>
<td>5,000</td>
<td>2,500</td>
<td>12.3</td>
</tr>
<tr>
<td>50,000</td>
<td>25,000</td>
<td>15.6</td>
</tr>
</tbody>
</table>

$N$ and $N/2$ are growing much faster than $\log N$!

slower growing is more efficient (fewer steps).

Efficiency of Binary Search

If $N$ is the number of elements in the array, how many comparisons (steps)?

<table>
<thead>
<tr>
<th>$N$=50,000</th>
<th>In terms of $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case:</td>
<td>1</td>
</tr>
<tr>
<td>Worst Case:</td>
<td>$16 \log_2 N$</td>
</tr>
</tbody>
</table>

Rounded up to next whole number

8.3 Sorting Algorithms

• Sort: rearrange the items in an array into ascending or descending order.

• Bubble Sort

• Selection Sort
The Bubble Sort

On each pass:
- Compare first two elements. If the first is bigger, they exchange places (swap).
- Compare second and third elements. If second is bigger, exchange them.
- Repeat until last two elements of the list are compared.
- Repeat this process (keep doing passes) until a pass completes with no exchanges.

Example: first pass

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 3 8 9 1</td>
<td>7 &gt; 2, swap</td>
</tr>
<tr>
<td>2 7 3 8 9 1</td>
<td>7 &gt; 3, swap</td>
</tr>
<tr>
<td>2 3 7 8 9 1</td>
<td>!(7 &gt; 8), no swap</td>
</tr>
<tr>
<td>2 3 7 8 9 1</td>
<td>!(8 &gt; 9), no swap</td>
</tr>
<tr>
<td>2 3 7 8 9 1</td>
<td>9 &gt; 1, swap</td>
</tr>
<tr>
<td>2 3 7 8 1 9</td>
<td>finished pass 1, did 3 swaps</td>
</tr>
</tbody>
</table>

Note: largest element is now in last position
Note: This is one complete pass!

Example: second and third pass

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 3 8 9 1</td>
<td>2&lt;3&lt;7&lt;8, no swap, !(8&lt;1), swap</td>
</tr>
<tr>
<td>2 3 7 8 9 1</td>
<td>(8&lt;9) no swap</td>
</tr>
<tr>
<td>2 3 7 8 9 1</td>
<td>finished pass 2, did one swap</td>
</tr>
<tr>
<td>2 3 7 1 8 9</td>
<td>2&lt;3&lt;7, no swap, !(7&lt;1), swap</td>
</tr>
<tr>
<td>2 3 1 7 8 9</td>
<td>7&lt;8&lt;9, no swap</td>
</tr>
<tr>
<td>2 3 7 8 1 9</td>
<td>finished pass 3, did one swap</td>
</tr>
</tbody>
</table>

Note: 2 largest elements in last 2 positions

Example: passes 4, 5, and 6

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 1 7 8 9</td>
<td>2&lt;3, !(3&lt;1) swap, 3&lt;7&lt;8&lt;9</td>
</tr>
<tr>
<td>2 3 7 8 9</td>
<td>finished pass 4, did one swap</td>
</tr>
<tr>
<td>2 1 3 7 8 9</td>
<td>!(2&lt;1) swap, 2&lt;3&lt;7&lt;8&lt;9</td>
</tr>
<tr>
<td>1 2 3 7 8 9</td>
<td>finished pass 5, did one swap</td>
</tr>
<tr>
<td>1 2 3 7 8 9</td>
<td>1&lt;2&lt;3&lt;7&lt;8&lt;9, no swaps</td>
</tr>
<tr>
<td>1 2 3 7 8 9</td>
<td>finished pass 6, no swaps, list is sorted!</td>
</tr>
</tbody>
</table>

Note: 3 largest elements in last 3 positions
Bubble sort
how does it work?

- At the end of the first pass, the largest element is moved to the end (it’s bigger than all its neighbors)
- At the end of the second pass, the second largest element is moved to just before the last element.
- The back end (tail) of the list remains sorted.
- Each pass increases the size of the sorted portion.
- No exchanges implies each element is smaller than its next neighbor (so the list is sorted).

Program using bubble sort

```cpp
#include <iostream>
using namespace std;

void bubbleSort(int array[], int size) {
    bool swap;
    int temp;
    do {
        swap = false;
        for (int i = 0; i < (size-1); i++) {
            if (array[i] > array[i+1]) {
                temp = array[i];
                array[i] = array[i+1];
                array[i+1] = temp;
                swap = true;
            }
        }
    } while (swap);
}

void showArray(int array[], int size) {
    for (int i=0; i<size; i++)
        cout << array[i] << " ";
    cout << endl;
}

int main() {
    int values[6] = {7, 2, 3, 8, 9, 1};
    cout << "The unsorted values are: \n";
    showArray(values, 6);
    bubbleSort(values, 6);
    cout << "The sorted values are: \n";
    showArray(values, 6);
}
```

Output:
The unsorted values are: 7 2 3 8 9 1
The sorted values are: 1 2 3 7 8 9

Selection Sort

- There is a pass for each position (0..size-1)
- On each pass, the smallest (minimum) element in the rest of the list is exchanged (swapped) with element at the current position.
- The first part of the list (the part that is already processed) is always sorted
- Each pass increases the size of the sorted portion.
**Selection sort**

*Example*

- **7 2 3 8 9 1**  
  1 is the min a[5], swap with a[0]
- **1 2 3 8 9 7**  
  2 is the min a[1], self-swap a[1]
- **1 2 3 8 9 7**  
  3 is the min a[2], self-swap a[2]
- **1 2 3 8 9 7**  
  7 is the min a[5], swap with a[3]
- **1 2 3 7 9 8**  
  8 is the min a[5], swap with a[4]
- **1 2 3 7 8 9**  
  sorted

Note: underlined portion of list is sorted.

**Selection Sort in C++**

*My version*

```cpp
// Returns the index of the smallest element, starting at start
int findIndexOfMin (int array[], int size, int start) {
    int minIndex = start;
    for (int i = start+1; i < size; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}

// Sorts an array, using findIndexOfMin
void selectionSort (int array[], int size) {
    int temp;
    int minIndex;
    for (int index = 0; index < (size -1); index++) {
        minIndex = findIndexOfMin(array, size, index);
        //swap
        temp = array[minIndex];
        array[minIndex] = array[index];
        array[index] = temp;
    }
}
```

Note: saving the index

We need to find the index of the minimum value so that we can do the swap

**Program using Selection Sort**

```cpp
#include <iostream>
using namespace std;

int findIndexOfMin (int [], int, int);
void selectionSort(int [], int);
void showError(int [], int);

int main() {
    int values[6] = {7, 2, 3, 8, 9, 1};
    cout << "The unsorted values are: \n";
    showError (values, 6);
    selectionSort (values, 6);
    cout << "The sorted values are: \n";
    showError(values, 6);
}

void showError (int array[], int size) {
    for (int i=0; i<size; i++)
        cout << array[i] << " " ;
    cout << endl;
}
```

Output:

```
The unsorted values are: 7 2 3 8 9 1
The sorted values are: 1 2 3 7 8 9
```

**Analysis of Algorithms**

*using Big O notation*

- Which algorithm is better, linear search or binary search?
- Which algorithm is better, bubble sort or selection sort?
- How can we answer these questions?

**Analysis of algorithms** is the determination of the amount of resources (such as time and storage) necessary to execute them.
Time Efficiency of Algorithms

- To classify the time efficiency of an algorithm:
  - Express “time” (using number of main steps), as a mathematical function of input size (or n below).
    - Binary search: \( f(n) = \log_2(n) \)
  - Need a way to be able to compare these math functions to determine which is better.
    - We are mostly concerned with which function has smaller values (# of steps) at very large data sizes.
    - We compare the growth rates of the functions and prefer the one that grows more slowly.

Classifications of (math) functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>( f(x) = b )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>( f(x) = \log_b(x) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Linear</td>
<td>( f(x) = ax + b )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Linearithmic</td>
<td>( f(x) = x \log_b(x) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Quadratic</td>
<td>( f(x) = ax^2 + bx + c )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Exponential</td>
<td>( f(x) = 2^x )</td>
<td>( O(2^n) )</td>
</tr>
</tbody>
</table>

- Last column is “big O notation”, used in CS.
- It ignores all but dominant term, constant factors

Comparing growth of functions

Time Efficiency of Algorithms

- To classify the time efficiency of an algorithm:
  - Express “time” (using number of main steps), as a mathematical function of input size.
  - Determine which classification the function fits into.

- Nearer to the top of the classification chart (on slide 30) is slower growth, and more efficient (constant is better than logarithmic, etc.)
**Efficiency of Searches**
(Assuming the array is already sorted)

- **Linear Search, worst case:**
  - Linear search: \( f(n) = n \) \(O(N)\)

- **Binary Search, worst case:**
  - Binary search: \( f(n) = \log_2(n) \) \(O(\log N)\)

- Which is slower growing (and thus fewer steps at large input sizes)?
  - \(O(\log N)\)

- Which search algorithm is more time efficient?
  - Binary search

**Efficiency of Selection Sort**

- \(N\) is the number of elements in the list
- Outer loop executes \(N-1\) times
- Inner loop executes \(N-1,\) then \(N-2,\) then \(N-3,\) ... then once. One comparison per loop iteration.
- Total number of comparisons (in inner loop):
  \[
  f(N) = (N-1) + (N-2) + \ldots + 2 + 1 = \text{sum of 1 to } N-1
  \]
  \[
  \text{sum of 1..N: } N + (N-1) + (N-2) + \ldots + 2 + 1 = N(N+1)/2
  \]
  Subtract \(N\) from each side:
  \[
  (N-1) + (N-2) + \ldots + 2 + 1 = N(N+1)/2 - N \\
  = (N^2+N)/2 - 2N/2 \\
  = (N^2+N-2N)/2 \\
  = N^2/2 - N/2
  \]
  \(O(N^2)\)

**Efficiency of Bubble Sort**

- Each pass makes \(N-1\) comparisons
- There will be (at most) \(N\) passes
- So worst case it’s:
  \[
  f(N) = (N-1)^*N = N^2 - N \] \(O(N^2)\)
- If you change the algorithm to look at only the **unsorted** part of the array in each pass, it’s exactly like the selection sort:
  \[
  (N-1) + (N-2) + \ldots + 2 + 1 = N^2/2 - N/2 \\
  \text{still } O(N^2)
  \]
- Neither algorithm is more efficient in the worst case.