Return to Searching

- Recall the linear search
  - Access the array from left to right
  - Compare each element with the specified value
  - If there is a match, perform some action
    - Return the index
    - Print a message

"Mary" "Eric" "Nancy" "John" "Jacob"
Another Implementation

```c
const int NOT_FOUND = -1; // must be an invalid index

// From CS1044 Notepack by Barnette, et al
int SearchStringArray(const string array[], int usage,
const string& val)
{
    int loc = 0; // location of the value

    // search until reach the end of the array or value found
    while ( (loc < usage) && (array[loc] != val) )
        loc++;

    // If the location is valid, the target was found.
    if (loc < usage)
        return loc;
    else
        return NOT_FOUND;
}
```

What If?

- Suppose the elements in our array are ordered.
- Let's play a game
  - Price is Right
  - The value of the prize is between 1 and 100 (cents)
  - How do you figure it out?
What Steps Did You Take?

- In class discussion only…

Binary Search

- Problem
  - Find a specific value in an ordered array (list)
  - Could use a linear search, but we can do better
- Input
  - An ordered array of elements (we'll use integers)
- Output
  - The index in the array where the value is found or NOT_FOUND if the value isn't in the array
**Binary Search Code**

```c
const int NOT_FOUND = -1; // must be invalid index

int BinarySearch(const int array[], int usage, int val) {
    int mid; // midpoint of the array
    int low = 0; // lowest index
    int high = usage - 1; // highest index

    while (low <= high) {
        mid = (low + high) / 2; // midpoint
        if (array[mid] == val)
            return mid;
        else if (val < array[mid])
            high = mid - 1; // look in lower half
        else
            low = mid + 1; // look in upper half
    }
    return NOT_FOUND; // value not in array
}
```

**Binary Search Trace**

```c
const int MAX_PRICE = 8;
int BinarySearch(const int array[], int usage, int val);
void main() {
    int loc;
    int prices[MAX_PRICE] = {
        21, 35, 47, 82, 110, 144, 171, 180
    };

    loc = BinarySearch(prices, MAX_PRICE, 110);
    cout << "Location of 110 is " << loc << endl;
    loc = BinarySearch(prices, MAX_PRICE, 38);
    cout << "Location of 38 is " << loc << endl;
}
```
Binary Search Trace

- Trace done in class…

Cost of Searching

- Number of comparisons for an $n$ element array

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Search</td>
<td>1</td>
<td>$n$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>Binary Search</td>
<td>1</td>
<td>$\log_2 n$</td>
<td>$\log_2 n$</td>
</tr>
</tbody>
</table>

- What does that amount to?

<table>
<thead>
<tr>
<th></th>
<th>$n/2$</th>
<th>$\log_2 n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>512</td>
<td>10</td>
</tr>
<tr>
<td>1048576</td>
<td>524288</td>
<td>20</td>
</tr>
</tbody>
</table>
Many problems and computer applications require data ordered in some way
  - Binary search
We need to have a total ordering on elements for sorting to work
  - One of
    a < b
    a > b
    a = b
must be true whenever you compare two elements.

We need to choose the order of the elements
- Ascending
  - Place smaller elements before larger elements
    20 35 40 76 83 90
- Descending
  - Place larger elements before smaller elements
    90 83 76 40 35 20
Let's Try It Ourselves

- Sort the following array in ascending order...

  60 35 94 82 23 71 59 48

Bubble Sort

- Basic Idea (Ascending order)
  - Compare two adjacent elements $a[i]$ and $a[i+1]$
  - If $a[i] > a[i+1]$, swap $a[i]$ and $a[i+1]$
    - because they are out of order

- A single pass loops over the unsorted part of the array one time
  - Largest element will be *bubbled* up to location of the last unsorted element
Bubble Sort (First Pass)

- During the first the entire array is unsorted.
- Go through array from left to right
- Stop at element size-2
- Largest element ends up at size-1

Bubble Sort (Remaining Passes)

- Suppose we are on the nth pass
- Elements from size-1, size-2, ..., size-(n-1) are already sorted
- Only need to sort 0 through size-n
  - Go through the array from 0 to (size-n)-1
  - Swap adjacent elements that are out of order.
- We need size-1 passes to sort the entire array.
**Bubble Sort Code**

```c
void Swap(int& a, int& b);

void BubbleSort(int list[], int size) {
    for (int pass = 1; pass <= size-1; pass++) {
        int last = size-pass; // index of last unsorted element
        // sort elements for one pass
        for (int idx=0; idx < last; idx++) {
            if (list[idx] > list[idx+1]) {
                Swap(list[idx], list[idx+1]);
            }
        }
    }
}

void Swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

**Bubble Sort Trace (In Class)**

```plaintext
60 35 94 82 23 71 59 48
```
Selection Sort

- Basic Idea
  - Find minimum element and place it in the left most position
  - After \( n \) passes, first \( n \) elements are sorted
- Name comes from selecting a position to place an element into

Selection Sort Trace

- Find the minimum element in unsorted elements
- Swap minimum with selected location
- Executes for size-1 passes
Selection Sort (Code)

- First, you give it a try...

Selection Sort Code

```c
void Swap(int& a, int& b);

void SelectionSort(int list[], int size) {
    int start, minElem, check;
    for (start = 0; start < (size - 1); start++) {
        // initially the starting element is smallest.
        minElem = start;
        // Find the minimum in the rest of the list
        for (check = start + 1; check < size; check++) {
            if (list[check] < list[minElem]) {
                minElem = check;
            }
        }
        Swap(list[start], list[minElem]);
    }
}
```
Bubble vs. Selection Sort

- Execute both sorts on

  | 72 | 58 | 34 | 96 | 12 | 27 |

<table>
<thead>
<tr>
<th>Sort</th>
<th>Comparisons</th>
<th>Swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Suppose we have an array of $n$ elements
- Assume worst case for both algorithms

<table>
<thead>
<tr>
<th>Sort</th>
<th>Comparisons</th>
<th>Swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>$(n^2 - n) / 2$</td>
<td>$(n^2 - n) / 2$</td>
</tr>
<tr>
<td>Selection</td>
<td>$(n^2 - n) / 2$</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>
Sorting

- There exist algorithms with $n \log_2 n$ growth in comparisons
- What does this imply for 1024 elements?

<table>
<thead>
<tr>
<th>Growth</th>
<th>Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(n^2 - n) / 2$</td>
<td>523776</td>
</tr>
<tr>
<td>$n \log_2 n$</td>
<td>10240</td>
</tr>
</tbody>
</table>