Definitions of Search and Sort

- **Search**: find a given item in a list, return the position of the item, or -1 if not found.
- **Sort**: rearrange the items in a list into some order (smallest to biggest, alphabetical order, etc.).
- “list” could be: array, linked list, string, etc.
- There are various methods (algorithms) for carrying out these common tasks.

Linear Search

(week 6)

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

```c
int searchList (int list[], int size, int value) {
    for (int i=0; i<size; i++)
        if (list[i] == value)
            return i;
    return -1;
}
```
**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**

<table>
<thead>
<tr>
<th>Target is 11</th>
<th>First</th>
<th>Mid</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>list: 2 4 7 10 11 45</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target &lt; 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>list: 2 4 7 10 11 45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target &gt; 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>list: 2 4 7 10 11 45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target == 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>list: 10 11 45</td>
</tr>
</tbody>
</table>

---

**Binary Search in C++**

**Iterative version**

```cpp
int binarySearch (int array[], int size, int target) {
    int first = 0,          //index to (current) first elem
        last = size - 1,    //index to (current) last elem
        middle,             //index of (current) middle elem
        position = -1;      //index 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 lower half
        } else {
            first = middle + 1;      //search upper half
        }
    }
    return position;
}
```

**Recursive version**

```cpp
int binarySearchRec(int array[], int first, int last, int value) {
    int middle; // Mid point of search
    if (first > last)           //check for empty list
        return -1;
    middle = (first + last)/2;  //compute middle index
    if (array[middle]==value)
        return middle;
    if (value < array[middle])    //recursion
        return binarySearchRec(array, first,middle-1, value);
    else
        return binarySearchRec(array, middle+1,last, value);
}

int binarySearch(int array[], int size, int value) {
    return binarySearchRec(array, 0, size-1, value);
}
```
What is sorting?

• Sort: rearrange the items in a list into ascending or descending order
  - numerical order
  - alphabetical order
  - etc.

55 112 78 14 20 179 42 67 190 7 101 1 122 170 8
1 7 8 14 20 42 55 67 78 101 112 122 170 179 190

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

• 36 24 10 6 12  pass 1: minimum is 6, swap
• 6 24 10 36 12  pass 2: minimum is 10, swap
• 6 10 24 36 12  pass 3: minimum is 12, swap
• 6 10 12 36 24  pass 4: minimum is 24, swap
• 6 10 12 24 36  sorted

Note: first n elements are sorted after pass n

Selection sort: code

```c
// 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 minIndex;
    for (int index = 0; index < (size -1); index++) {
        minIndex = findIndexOfMin(array, size, index);
        swap(array[minIndex],array[index]);
    }
}
```
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 until a pass completes with no exchanges

Bubble sort Example

• 7 2 3 8 9 1
  7 > 2, swap

• 2 7 3 8 9 1
  7 > 3, swap

• 2 3 7 8 9 1
  !(7 > 8), no swap

• 2 3 7 8 9 1
  !(8 > 9), no swap

• 2 3 7 8 9 1
  9 > 1, swap

• 2 3 7 8 1 9
  finished pass 1, did 3 swaps

Note: largest element is in last position

2 largest elements in last 2 positions

• 2 3 7 8 1 9
  2<3<7<8, no swap, !(8<1), swap

• 2 3 7 1 8 9
  (8<9) no swap

• finished pass 2, did one swap

3 largest elements in last 3 positions

• 2 3 7 8 1 9
  2<3<7, no swap, !(7<1), swap

• 2 3 1 7 8 9
  7<8<9, no swap

• finished pass 3, did one swap

1 largest element in last position

• 2 3 1 7 8 9
  2<3, !(3<1) swap, 3<7<8<9

• 2 1 3 7 8 9

• finished pass 4, did one swap

• 2 1 3 7 8 9
  !(2<1) swap, 2<3<7<8<9

• 1 2 3 7 8 9

• finished pass 5, did one swap

• 1 2 3 7 8 9
  1<2<3<7<8<9, no swaps

• finished pass 6, no swaps, list is sorted!
**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).

**Quick sort**

- Divide and conquer!
- 2 (hopefully) half-sized lists sorted recursively
- the algorithm:
  - If list size is 0 or 1, return. otherwise:
  - partition into two lists:
    - pick one element as the pivot
    - put all elements less than pivot in first half
    - put all elements greater than pivot in second half
  - recursively sort first half and then second half of list.

**Bubble sort: code**

```c++
template<class ItemType>
void bubbleSort (ItemType a[], int size) {
    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < (size-1); i++) {
            if (a[i] > a[i+1]) {
                swap(a[i],a[i+1]);
                swapped = true;
            }
        }
    } while (swapped);
}
```

**Quicksort**

Example.
Quicksort: partitioning

- Goal: partition a sub-array so that:
- 4 8 5 6 3 19 12 pick middle as pivot, swap to front
- compare each elem to pivot, if elem<6, increment pivotIndex, swap
  - 6 8 5 4 3 19 12 8>6, continue
  - 6 8 5 4 3 19 12 5<6, pivotIndex++ and swap
  - 6 5 4 8 3 19 12 4<6, pivotIndex++ and swap
  - 6 5 4 3 8 19 12 3<6, pivotIndex++ and swap
  - 6 5 4 3 8 19 12 19 and 12 not less than 6, don't swap
- After the scan, swap A[start] with A[pivotIndex], return pivotIndex
- 3 5 4 6 8 19 12 return 3 as index of pivot (6)

Quicksort: code

```c
void quickSort(int set[], int start, int end) {
    if (start < end) {
        // Get the pivot point (and partition the set).
        int pivotPoint = partition(set, start, end);
        // Sort the first sub list.
        quickSort(set, start, pivotPoint - 1);
        // Sort the second sub list.
        quickSort(set, pivotPoint + 1, end);
    }
}

void quickSort(int set[], int size) {
    quickSort(set, 0, size-1);
}
```

Sample Problem

**Sorting Benchmarks:** Write a program that uses three identical arrays of at least 20 integers. It should call a different function to sort each array in ascending order, one that uses the bubble sort algorithm, one that uses the selection sort algorithm, and one that uses the quicksort algorithm. Each function should keep a count of the number of exchanges (swaps) it makes. Display these values on the screen.