Search Algorithms

- The most important operation on a list is the search algorithm
- Using a search algorithm you can:
  - Determine whether a particular item is in the list
  - If the data are specially organized, find the location where a new item can be inserted
  - Find an item in the list and remove it
- The search algorithm’s performance is crucial
  - If the search is slow, it takes a large amount of computer time to accomplish your task
  - If the search is fast you can accomplish your task quickly

The Sequential Search (Linear Search)

- Always starts at the first element in the list
- The search continues until either the item is found in the list or the entire list is searched
- If the search item is found, its index is returned
- If the search is unsuccessful, -1 is returned
- When analyzing a search algorithm, we count the number of key comparisons
The Sequential Search (Linear Search)

Example

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

Task 1: search item 8 in the list

Step 1: compare 8 and list[0]

Step 2: compare 8 and list[1]
Task 1: search item 8 in the list

Step 3: compare 8 and list[2]

Step 4: compare 8 and list[3]

Step 5: compare 8 and list[4]

Step 6: compare 8 and list[5]

return index=5
The Sequential Search

**Best Case:** If the search item is the first element in the list, the algorithm makes one comparison.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

yes

6

**Worst Case:** If the search item is the last element in the list, the algorithm makes \( n \) comparisons, where \( n \) is the number of elements in the list.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

no no no no no no no yes

Sequential Search Algorithm

The best and worst cases are not likely to occur every time we apply the sequential search, so we determine the average.

To determine the average number of comparisons the successful case of the sequential search algorithm:
- Consider all possible cases
- Find the number of comparison for each case
- Add the number of comparisons and divide by the number of cases

Assuming there are \( n \) elements in the list, the following expression gives the average number of comparisons:

\[
1+2+\ldots+n \quad \frac{n(n+1)}{2}
\]
**Sequential Search Algorithm**

- Therefore the expression that gives the average number of comparisons made by the sequential search in the successful case is:
  \[
  \frac{1+2+\ldots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}
  \]
- This expression shows that on average the sequential search algorithm searches half the list.
- The sequential search is not good for large lists.

**Ordered Lists**

- A list is ordered if its elements are sequenced according to some criteria.
- The elements of a list are usually put in ascending order.
- The operations determining whether the list is empty or full, determining the length of the list, printing the list, and clearing the list are the same for an ordered list as an unordered list.

**Binary Search**

- A sequential search is not good for large lists because, on average, the sequential search searches half the list.
- A binary search is very fast.
- A binary search can only be performed on ordered lists.
- The binary search uses the “divide and conquer” technique to search the list.
- First the search item is compared with the middle element of the list.
- If the search item is less than the middle element of the list, restrict the search to the upper half of the list otherwise, search the lower half of the list.
- Because we need to determine the middle element of the list frequently, the binary search algorithm is typically implemented for array-based lists.
Binary Search

- To determine the middle element of the list, add the starting index to the ending index and divide by 2 to calculate its index. That is: mid = \( \frac{\text{first} + \text{last}}{2} \)
- If the item is found, its location is returned and if it is not in the list, -1 is returned
- In the binary search algorithm, each time through the loop we make two comparisons
- The last time through the loop only one comparison is made

Example: Search 75 in the ordered list
list={4, 8, 19, 25, 34, 39, 45, 48, 66, 75, 89, 95}

First, check 75 with the middle element: list[mid], where mid=(first+last)/2, first=0, last=11

75 > 39, so the right half of list should be searched
Binary Search

- Search 75 in the ordered list
  list=[4, 8, 19, 25, 34, 39, 45, 48, 66, 75, 89, 95]

  0 1 2 3 4 5 6 7 8 9 10 11
  4 8 19 25 34 39 45 48 66 75 89 95

  \textbf{first} \quad \textbf{last}

75 > 66, so the new right half of list should be searched

Binary Search

- Search 75 in the ordered list
  list=[4, 8, 19, 25, 34, 39, 45, 48, 66, 75, 89, 95]

  0 1 2 3 4 5 6 7 8 9 10 11
  4 8 19 25 34 39 45 48 66 75 89 95

  \textbf{first} \quad \textbf{last}

compare 75 with list[mid], list[mid], where mid=(first+last)/2, first=6, last=11
Binary Search

Search 75 in the ordered list
list={4, 8, 19, 25, 34, 39, 45, 48, 66, 75, 89, 95}

0 1 2 3 4 5 6 7 8 9 10 11
4 8 19 25 34 39 45 48 66 75 89 95

75 < 89, so go to search the left half

Search 75 in the ordered list
list={4, 8, 19, 25, 34, 39, 45, 48, 66, 75, 89, 95}

0 1 2 3 4 5 6 7 8 9 10 11
4 8 19 25 34 39 45 48 66 75 89 95

first last

compare 75 with list[mid], list[mid], where
mid=(first+last)/2, first=9, last=9

We got it!!! 7 comparisons

int binarySearch(const int list[], int listLength, int searchItem)
{
    int first=0, last=listLength - 1, mid;
    bool found = false;
    while((first <= last) && !found)
    {
        mid = (first+last)/2;
        if (list[mid] == searchItem)
            found = true;
        else if (list[mid] > searchItem)
            last = mid - 1;
        else
            first = mid + 1;
    } // end of while
    if(found)
        return mid;
    else
        return -1;
} // end of binarySearch
Binary Search

Exercise

Write a function

```cpp
int binarySearch(const int list[], int listLength, int searchItem)
```

to perform the binary search on a non-ascending list

Performance of Binary Search

Every iteration of the while loop cuts the search list in half

Because \(1000 \approx 2^{10}\), the while loop has at most 11 iterations to determine if \(x\) is in \(L\)

Each iteration of the loop makes two comparisons, i.e. (1) == and (2) >

The binary search makes at most 22 comparisons to determine whether \(x\) is in \(L\)

By comparison the sequential search algorithm, on average makes 500 comparisons, and 1000 comparison in the worst case

Performance of Binary Search

Suppose \(L\) is of size 1,000,000

\(1,000,000 \approx 2^{20}\), therefore the loop has approximately 21 iterations to determine whether an element is in \(L\)

Each iteration makes 2 comparisons

Therefore to determine whether an element is in \(L\), a binary search makes at most 42 comparisons

By contrast, on average the sequential search makes 500,000 comparisons to determine whether an element is in \(L\), and 1,000,000 comparisons in the worst case

If \(L\) is a sorted list of size \(n\), to determine whether or not an element is in \(L\), a binary search makes at most \(2*\log_2 n + 2\) comparisons
Searches

- **Unsuccessful Search**
  - In the case of an unsuccessful search of a list of length $n$, the number of comparisons made by the binary search is approximately: $2 \times \log_2 n + 2$ comparisons

- **Successful Search**
  - In the case of a successful search it can be shown that for a list of length $n$, on average, a binary search makes:
    $$\frac{2(n + 1) \log_2 (n + 1) - 3}{n}$$

Insertion into an Ordered List

- The algorithm to insert an item in an ordered list is:
  - Use an algorithm similar to a binary search to find the place where the item is to be inserted
  - if item is already in this list
    - output an appropriate message
  - else
    - use the function `insertAt` to insert the item in the list

Asymptotic Notations

- For a list of size $n$, the function $f(n)$ gives the number of comparisons done by the search algorithm
- If we know how the function $f(n)$ grows as the size of the problem grows, we can determine the efficiency of the algorithm. Consider the following tables.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2 n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>256</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>1024</td>
<td>4,294,967,296</td>
</tr>
</tbody>
</table>
Asymptotic Notations

- If the number of basic operations is a function of $f(n) = n^2$, then the number of operations is quadrupled when $n$ is doubled.
- If the number of basic operations is a function of $f(n) = 2^n$, then the number of operations is squared when $n$ is doubled.
- If the number of operations is a function of $f(n) = \log_2 n$, then the change in the number of basic operations is insignificant.

Common Big-O functions that appear in algorithm analysis

- Let $f(n) = O(g(n))$
- $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(2^n)$

Algorithm analysis of the sequential and binary search algorithm

<table>
<thead>
<tr>
<th>Function $g(n)$</th>
<th>Successful Search</th>
<th>Unsuccessful Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(n) = 1$</td>
<td>$\frac{N+1}{2} = O(n)$</td>
<td>$n = O(n)$</td>
</tr>
<tr>
<td>$g(n) = \log_2 n$</td>
<td>$2(n+1) \log_2 (n+1) - 3 = O(\log_2 n)$</td>
<td>$2^\log_2 n + 2 = O(\log_2 n)$</td>
</tr>
<tr>
<td>$g(n) = n$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$g(n) = n \log_2 n$</td>
<td>$O(n \log_2 n)$</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>$g(n) = n^2$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$g(n) = 2^n$</td>
<td>$O(2^n)$</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>
Selection Sort

- The selection sort algorithm sorts a list by selecting the smallest element in the (unsorted portion of the) list and then moving this element to the top of the list.
- The first time we locate the smallest item in the entire list.
- The second time we locate the smallest item in the list starting from the second item and so on.

Example

```
5 4 9 0 1 6 8 2 7 3
0 1 2 3 4 5 6 7 8 9
```

smallest

swap
Selection Sort

Example

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
```

```
0 7 2 8 6 1 3 9 4 5
```

**Selection Sort**

Example

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
```

```
0 7 2 8 6 1 3 9 4 5
```

smallest

**Selection Sort**

Example

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
```

```
0 1 2 8 6 7 3 9 4 5
```

smallest

**Selection Sort**

Example

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
```

```
0 1 2 8 6 7 3 9 4 5
```

smallest

**Selection Sort**

Example

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
```

```
0 1 2 8 6 7 3 9 4 5
```
Selection Sort

- Example

```
0 1 2 8 6 7 3 9 4 5
```

Selection Sort

- Example

```
0 1 2 8 6 7 3 9 4 5
```

Selection Sort

- Example

```
0 1 2 3 6 7 8 9 4 5
```

Selection Sort

- Example

```
0 1 2 3 6 7 8 9 4 5
```
Selection Sort

Example

unsorted

0 1 2 3 4 7 8 9 6 5

smallest

swap

0 1 2 3 4 7 8 9 6 5

unsorted

0 1 2 3 4 5 8 9 6 7

smallest

swap

0 1 2 3 4 5 8 9 6 7
Selection Sort

Example

Selection Sort

Example

Selection Sort

Example

Selection Sort

Example

Finished!
Selection Sort

```c
void selectionSort (int a[], int n) {
    int minIndex;
    for (int beginning = 0; beginning < n-1 ; beginning++) {
        // find index of the smallest in unsorted subarray
        minIndex = getMinIndex(a, beginning, n-1);
        swap (a[beginning], a[minIndex]);
    }
}
```

Selection Sort

Exercise:
The sequence is: 3 2 5 1 4

Write the rest sequence changes by selection sort until it is fully sorted.

(1) 3 2 5 1 4
(2) 1 2 5 3 4
(3) 1 2 3 5 4
(4) 1 2 3 4 5

Insertion Sort

- The insertion sort algorithm sorts a list by inserting each element one after the other. At each insertion, the element is inserted at its correct position by shifting the larger elements.
Insertion Sort

Try to insert 2 into the sorted portion.

2 is the first element in the unordered portion.

temp

3 7 2 8 6 1 0 9 4 5
Insertion Sort

```
  0 1 2 3 4 5 6 7 8 9
3 7 2 8 6 1 0 9 4 5
```

```
  0 1 2 3 4 5 6 7 8 9
3 7 2 8 6 1 0 9 4 5
```

```
  0 1 2 3 4 5 6 7 8 9
3 7 2 8 6 1 0 9 4 5
```

```
  0 1 2 3 4 5 6 7 8 9
3 7 2 8 6 1 0 9 4 5
```
Insertion Sort

```
0 1 2 3 4 5 6 7 8 9
3 7 7 8 6 1 0 9 4 5
```

Insertion Sort

```
0 1 2 3 4 5 6 7 8 9
3 3 7 8 6 1 0 9 4 5
```

Insertion Sort

```
0 1 2 3 4 5 6 7 8 9
2 3 7 8 6 1 0 9 4 5
```

Insertion Sort

```
0 1 2 3 4 5 6 7 8 9
2 3 7 8 6 1 0 9 4 5
```
Insertion Sort

Exercise:
The sequence is: 3 2 5 1 4

Write the rest sequence changes by insertion sort until it is fully sorted.

(1) 3 2 5 1 4
(2) 2 3 5 1 4
(3) 2 3 5 1 4
(4) 1 2 3 5 4
(5) 1 2 3 4 5
Bubble Sort

- The Bubble sort algorithm sorts a list by moving the current maximum element to the top of the list. On the fly, it will swap its position with each element that is smaller.
- The process is repeated with the second largest element, then the third,...

Bubble Sort

Iteration 1

<table>
<thead>
<tr>
<th>10</th>
<th></th>
<th>compare and swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bubble Sort

Iteration 1

<table>
<thead>
<tr>
<th>7</th>
<th></th>
<th>compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bubble Sort

Iteration 1

7
10
19
5
6

compare and swap

Iteration 1

7
10
5
19
6

compare and swap

Iteration 1

7
10
5
6
19

Iteration 2

7
10
5
6
19

compare
**Bubble Sort**

**Iteration 2**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>10</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*compare and swap*

**Iteration 2**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*compare and swap*

**Iteration 3**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*compare and swap*
Bubble Sort

**Iteration 3**

- 5
- 7
- 6
- 10
- 19

Compare and swap

**Iteration 4**

- 5
- 6
- 7
- 10
- 19

Compare

**finished**

---

Bubble Sort

**Iteration 3**

- 5
- 6
- 7
- 10
- 19

**Iteration 4**

- 5
- 6
- 7
- 10
- 19

**finished**

---

Bubble Sort

**Iteration 3**

- 5
- 6
- 7
- 10
- 19

**Iteration 4**

- 5
- 6
- 7
- 10
- 19

**finished**

---

Bubble Sort

**Iteration 3**

- 5
- 6
- 7
- 10
- 19

**Iteration 4**

- 5
- 6
- 7
- 10
- 19

**finished**

---
**Bubble Sort**

- A special case

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Analysis: Insertion Sort**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Comparisons</th>
<th>Number of Swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>