Sorting and Searching Algorithms

Tessema M. Mengistu
Department of Computer Science
Southern Illinois University Carbondale
tessema.mengistu@siu.edu
Room - Faner 3131
Outline

• Introduction to Sorting
• Sorting Algorithms
  – Selection Sort
  – Insertion Sort
  – Shell Sort
  – Merge Sort
  – Quick Sort
• Searching Algorithms
  – Sequential Search
  – Binary search
Introduction to Sorting

• Arranging things into either ascending or descending order

• For example
  – Arranging a group of numbers from lowest to highest or from highest to lowest
  – Ordering strings in alphabetical order

• Many sorting algorithms exist
  – Selection, Insertion, Bubble, Merge, Radix, Shell, ....
Comparable interface

- For an array to be sortable, objects must be comparable
  - Must implement interface `Comparable`

```java
<T extends Comparable<T>>
```

- We could begin our class with

```java
public class SortArray
{
    public static <T extends Comparable<T>> void sort(T[] a, int n)
    {
        ...
    }
```
Selection Sort

- Sorting books by height on the shelf
  - Take all books off shelf
  - Select shortest, replace on shelf
  - Continue until all books

- Alternative
  - Look down shelf, select shortest
  - Swap first element with selected shortest
  - Move to second slot, repeat process
### Original array

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

### Pass one

**Original array:**

|   | 15 | 8 | 10 | 2 | 5 |

- **min = 15**
- **minIndex = 0**

|   | 15 | 8 | 10 | 2 | 5 |

- **min = 8**
- **minIndex = 1**

|   | 2  | 8 | 10 | 15 | 5 |

- **min = 2**
- **minIndex = 3**

### Pass Two

**Original array:**

|   | 2  | 8 | 10 | 15 | 5 |

- **min = 8**
- **minIndex = 1**

|   | 2  | 8 | 10 | 15 | 5 |

- **min = 5**
- **minIndex = 4**
Pass three

2 5 10 15 8

Pass Four

2 5 10 15 8

2 5 8 15 10

Output

2 5 8 10 15
Selection Sort

• Pseudocode for algorithm

\[\text{Algorithm selectionSort}(a, n)\]

\[\text{// Sorts the first } n \text{ entries of an array } a.\]

\[\text{for (index = 0; index < n - 1; index++)}\]

\[\{\]

\[\text{indexOfNextSmallest = the index of the smallest value among}\]
\[a[\text{index}], a[\text{index} + 1], \ldots, a[n - 1]\]

\[\text{Interchange the values of } a[\text{index}] \text{ and } a[\text{indexOfNextSmallest}]\]

\[\text{// Assertion: } a[0] \leq a[1] \leq \ldots \leq a[\text{index}], \text{ and these are the smallest}\]
\[\text{// of the original array entries. The remaining array entries begin at } a[\text{index} + 1].\]
\[\}\]
Selection Sort

• Example
  – Trace the steps that the method selection takes when sorting the following array into ascending order: 9 6 2 8 4 7 5 3 1.
public static <T extends Comparable<? super T>> void selectionSort(T[] a)
{
    int IndexOfNextSmallest;
    T temp;
    for (int index = 0; index < a.length-1; index++)
    {
        IndexOfNextSmallest = index;
        T min = a[IndexOfNextSmallest];
        for (int scan = index+1; scan < a.length; scan++)
            if (a[scan].compareTo(min) < 0) {
                IndexOfNextSmallest = scan;
                min=a[scan];
            }
        // Swap the values
        temp = a[index];
        a[index]= a[IndexOfNextSmallest];
        a[IndexOfNextSmallest] = temp;
    }
}
Recursive Selection Sort

Algorithm selectionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
    indexOfNextSmallest = the index of the smallest value among
                           a[first], a[first + 1], ..., a[last]

    Interchange the values of a[first] and a[indexOfNextSmallest]
    // Assertion: a[0] ≤ a[1] ≤ ... ≤ a[first] and these are the smallest
    // of the original array entries. The remaining array entries begin at a[first + 1].
    selectionSort(a, first + 1, last)
}
Recursive Selection Sort

```java
public static <T extends Comparable<?>> void selectionSort ( T[] a, int first, int last) {
    if (first < last) {
        int IndexOfNextSmallest = first;
        T temp;
        T min = a[IndexOfNextSmallest];
        for (int scan = first+1; scan < a.length; scan++) {
            if (a[scan].compareTo(min) < 0) {
                IndexOfNextSmallest = scan;
                min = a[scan];
            }
        }
        // Swap the values
        temp = a[first];
        a[first] = a[IndexOfNextSmallest];
        a[IndexOfNextSmallest] = temp;
        selectionSort(a, frist+1, last);   }         }
```
Efficiency of Selection Sort

• Efficiency of selection sort is $O(n^2)$ for all cases
• The inner loop executes
  $$(n-1)+(n-2)+(n-3)+\ldots\ldots+1 = \frac{n(n-1)}{2}$$
• In addition to comparisons, it requires $O(n)$ swaps.
Insertion Sort

• When book found taller than one to the right
  – Remove book to right
  – Slide taller book to right
  – Insert shorter book into that spot

• Compare shorter book just moved to left
  – Make exchange if needed

• Continue ...
Insertion Sort Algorithm

• Partitions the array into sorted and unsorted
  – Initially the sorted part contains only the first element
  – The unsorted contains all the other elements

• At each pass
  – Removes the first entry from the unsorted part and inserts it into its proper sorted position within the sorted part.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pass 1</th>
<th>15</th>
<th>8</th>
<th>10</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pass 2</th>
<th>8</th>
<th>15</th>
<th>10</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Pass 3</th>
<th>8</th>
<th>10</th>
<th>15</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Pass 4</th>
<th>2</th>
<th>8</th>
<th>10</th>
<th>15</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>
Insertion Sort

8  2  6  4  9  7  1

8  2  6  4  9  7  1

2  8  6  4  9  7  1

2  6  8  4  9  7  1

2  4  6  8  9  7  1

2  4  6  8  9  7  1

2  4  6  7  8  9  1

1  2  4  6  7  8  9
**Algorithm** insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] iteratively.

for (unsorted = first + 1 through last)
{
    nextToInsert = a[unsorted]
    insertInOrder(nextToInsert, a, first, unsorted - 1)
}

**Algorithm** insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted entries a[begin] through a[end].

index = end  // index of last entry in the sorted portion
// make room. if needed. in sorted portion for another entry

while ( (index >= begin) and (anEntry < a[index]) )
{
    a[index + 1] = a[index]  // make room
    index--
}
// Assertion: a[index + 1] is available.

a[index + 1] = anEntry  // insert
Insertion Sort

• Example

  – Trace the steps that the method insertion takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3 1.
public static <T extends Comparable<? super T>> void insertionSort (T[] a, int start, int end) {
    for (int index = start+1; index < end; index++) {
        T min = a[index];
        int position = index;
        // Shift larger values to the right
        while (position > start && min.compareTo(a[position-1]) < 0) {
            a[position] = a[position-1];
            position--;
        }
        a[position] = min; // put min in its final location
    }
}
Insertion Sort

• Efficiency
  – Loop executes at most
    \[1 + 2 + \ldots (n - 1)\] times

• Sum is \[\frac{n \cdot (n - 1)}{2}\]
  – Which gives \(O(n^2)\)

• Best case – array already in order, \(O(n)\)
Shell Sort

- Previously mentioned sorts are simple, often useful
  - However can be inefficient for large arrays
  - Array entries move only to adjacent locations

- Shell sort
  - Variation of Insertion sort - but faster
  - Moves/compares entries beyond adjacent locations
    - Sort sub arrays of entries at equally spaced indices
Shell Sort

• Example
  – Consider n/2
    • Every 6\textsuperscript{th} element

\begin{center}
\begin{tabular}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
10 & 16 & 11 & 4 & 15 & 3 & 9 & 6 & 1 & 17 & 8 & 12 & 7 \\
\end{tabular}
\end{center}

– Gives 6 sub arrays
– Sort the sub arrays separately using insertion sort
– Combine the output
Shell Sort

7 9 10
6 16
1 11
4 17
8 15
3 12

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>3</th>
<th>9</th>
<th>16</th>
<th>11</th>
<th>17</th>
<th>15</th>
<th>12</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Shell Sort

- Form a new sub array
  - Reduce the separation again by n/2
  - Apply insertion sort on the sub array
Shell Sort

• Continue till the separation is 1
  – The ordinary insertion sort of the entire array
Shell Sort

• Example
  – Apply the Shell sort to the array 9 8 2 7 5 4 6 3 1. What are the intermediate steps?
public static <T extends Comparable<? super T>> void shellSort ( T[] a)
{
    int pos;
for (int gap = a.length/2; gap>0; gap/=2)
    {
        for (int i = gap; i<a.length; i++)
            {
                T min = a[i];
                pos = i;
        for (pos>=gap && (min.compareTo(a[pos-gap]) < 0) )
            {
                a[pos] = a[pos-gap];
                pos-=gap; }
        a[pos] = min;
    }
}
Shell Sort

• Uses insertion sort several times
  – The problem size is much smaller than the original one
  – The last one is on an array that is entirely sorted
• Worst case - $O(n^2)$ (not $O(n^3)$)
• Average case – $O(n^{1.5})$
Comparison of Insertion, Selection and Shell Sort

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n)$</td>
<td>$O(n^{1.5})$</td>
<td>$O(n^2)$ or $O(n^{1.5})$</td>
</tr>
</tbody>
</table>
Faster Sorting Algorithms

• Insertion, selection, and shell are sufficient when the problem size is small arrays
• However, when we need to sort very large arrays frequently, those methods take much time
• We need better sorting algorithms
Merge Sort

• Divide array into two halves
  – Sort the two halves
  – Merge them into one sorted array

• Uses strategy of “divide and conquer”
  – Divide problem up into two or more distinct, smaller tasks

• Good application for recursion
The major steps in a merge sort

1. Divide the array into two halves

2. Sort the two halves

3. Merge the sorted halves into another array

4. Copy the merged array back into the original array
Merging Step

First array

3 5 7 9

Second array

0 2 4 6

3 > 0, so copy 0 to new array

3 5 7 9

3 > 2, so copy 2 to new array

3 5 7 9

3 < 4, so copy 3 to new array

3 5 7 9

5 > 4, so copy 4 to new array

3 5 7 9

5 < 6, so copy 5 to new array

3 5 7 9

7 > 6, so copy 6 to new array

3 5 7 9

The entire second array has been copied to the new array
Copy the rest of the first array to the new array
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
    mid = (first + last) / 2
    mergeSort(a, tempArray, first, mid)
    mergeSort(a, tempArray, mid + 1, last)
    // Merge the sorted halves a[first..mid] and a[mid+1..last] using the array tempArray
}
Algorithm to Merge

\textbf{Algorithm} \texttt{merge(a, tempArray, first, mid, last)}
\hfill // Merges the adjacent subarrays a[first..mid] and a[mid+1..last].

beginHalf1 = first
endHalf1 = mid
beginHalf2 = mid + 1
endHalf2 = last

// While both subarrays are not empty, compare an entry in one subarray with
// an entry in the other; then copy the smaller item into the temporary array
index = 0 // next available location in tempArray
\textbf{while} ( (beginHalf1 <= endHalf1) \textbf{and} (beginHalf2 <= endHalf2) )
\{ 
  \textbf{if} (a[beginHalf1] <= a[beginHalf2])
  \{ 
    tempArray[index] = a[beginHalf1]
    beginHalf1++
  \}
  \textbf{else}
  \{ 
    tempArray[index] = a[beginHalf2]
    beginHalf2++
  \}
  index++
\}

// Assertion: One subarray has been completely copied to tempArray.

\textit{Copy remaining entries from other subarray to tempArray}
\textit{Copy entries from tempArray to array a}
Effect of recursive calls to mergeSort

Merge steps

Copy to original array
Merge Sort

• Merge sort rearranges the entries in an array during its merge steps not recursive calls

• Example
  – Trace the steps that a merge sort takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3.
public static <T extends Comparable<? super T>> void mergeSort(T[] a)
{
    T[] tmp = (T[]) new Comparable<?>[a.length];
    mergeSort(a, tmp, 0, a.length - 1);
}

private static void mergeSort(T[] a, T[] tmp, int left, int right)
{
    if (left < right)
    {
        int center = (left + right) / 2;
        mergeSort(a, tmp, left, center);
        mergeSort(a, tmp, center + 1, right);
        merge(a, tmp, left, center + 1, right);
    }
}
private static void merge(T[ ] a, T[ ] tmp, int left, int right, int rightEnd)
{
    int leftEnd = right - 1;
    int k = left;
    int num = rightEnd - left + 1;
    while(left <= leftEnd && right <= rightEnd) {
        if(a[left].compareTo(a[right]) <= 0)
            tmp[k++] = a[left++];
        else
            tmp[k++] = a[right++];
    }
    while(left <= leftEnd) // Copy rest of first half
        tmp[k++] = a[left++];
    while(right <= rightEnd) // Copy rest of right half
        tmp[k++] = a[right++];

    for(int i = 0; i < num; i++, rightEnd--)
        a[rightEnd] = tmp[rightEnd];
}
Efficiency of Merge Sort

• For \( n = 2^k \) entries
  – In general \( k \) levels of recursive calls are made
• Each merge requires at most \( 3n - 1 \) operations
  – \( n-1 \) comparisons
  – \( n \) moves to temp array
  – \( n \) moves back to the original array
• Calls to merge do at most \( 3n - 2^2 \) operations
• Can be shown that efficiency is \( O(n \log n) \)
• Disadvantage
  – Requires a temporary array – more memory
Quick Sort

• Like merge sort, divides arrays into two portions
  – Unlike merge sort, portions not necessarily halves of the array

• One entry called the “pivot”
  – Pivot in position that it will occupy in final sorted array
  – Entries in positions before pivot less than or equal to the pivot
  – Entries in positions after pivot are greater than or equal to the pivot
Quick Sort Algorithm

Algorithm quickSort(a, first, last)
   // Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
   Choose a pivot
   Partition the array about the pivot
   pivotIndex = index of pivot
   quickSort(a, first, pivotIndex - 1) // sort Smaller
   quickSort(a, pivotIndex + 1, last) // sort Larger
}
Partitioning Algorithm

• Swap the pivot element with the last element of the array
• Start from the first index and traverse the array to the left till an element that is greater than or equal to the pivot is found
  – Get the index - indexFromLeft
• Start from one from the last index and traverse the array to the right till an element that is less than or equal to the pivot is found
  – Get the index - indexFromRight
• If the indexFromLeft is less than indexFromRight, swap the values
• Continue till indexFromLeft is greater than indexFromRight
• Swap the pivot with indexFromLeft
fromleft = 0
fromright = 6

fromleft = 1
fromright = 6

fromleft = 3
fromright = 5

fromleft = 4
fromright = 3
(a) 

(b) 
indexFromLeft: 1 3 5 0 4 6 1 2 4
indexFromRight: 6

(c) 
indexFromLeft: 1 3 2 0 4 6 1 5 4
indexFromRight: 6

(d) 
indexFromLeft: 3 3 2 0 4 6 1 5 4
indexFromRight: 5

(e) 
indexFromLeft: 3 3 2 0 1 6 4 5 4
indexFromRight: 5

(f) 
indexFromLeft: 4 3 2 0 1 6 4 5 4
indexFromRight: 3

(g) 

(h) 
Smaller  Pivot  Larger

3 2 0 1 4 4 5 6
Pivot selection

- The median of three entries in the array: the first entry, the middle entry, and the last entry.
  - Sort only those three entries and use the middle entry of the three as the pivot.
Quick Sort

• Minor modification to partition algorithm
  – Pivot swapped with array[last-1]
  – indexFromLeft = first + 1
  – indexFromLast = last - 2
Pivot Calculation

```java
private static <T extends Comparable<? super T>>
    void sortFirstMiddleLast(T[] a, int first, int mid, int last)
{
    order(a, first, mid); // make a[first] <= a[mid]
    order(a, mid, last);  // make a[mid] <= a[last]
    order(a, first, mid); // make a[first] <= a[mid]
} // end sortFirstMiddleLast

private static <T extends Comparable<? super T>>
    void order(T[] a, int i, int j)
{
    if (a[i].compareTo(a[j]) > 0)
        swap(a, i, j);
} // end order

/** Swaps the array entries array[i] and array[j]. */
private static void swap(Object[] array, int i, int j)
{
    Object temp = array[i];
    array[i] = array[j];
    array[j] = temp;
} // end swap
```
private static <T extends Comparable<? super T>>
    int partition(T[] a, int first, int last)
{
    int mid = (first + last) / 2;
    sortFirstMiddleLast(a, first, mid, last);

    swap(a, mid, last - 1);
    int pivotIndex = last - 1;
    T pivot = a[pivotIndex];

    int indexFromLeft = first + 1;
    int indexFromRight = last - 2;

    boolean done = false;
    while (!done)
    {
        while (a[indexFromLeft].compareTo(pivot) < 0)
            indexFromLeft++;

        while (a[indexFromRight].compareTo(pivot) > 0)
            indexFromRight--;

        assert a[indexFromLeft].compareTo(pivot) >= 0 &&
        a[indexFromRight].compareTo(pivot) <= 0;
if (indexFromLeft < indexFromRight)
{
    swap(a, indexFromLeft, indexFromRight);
    indexFromLeft++;
    indexFromRight--;
}
else
    done = true;
} // end while

// place pivot between Smaller and Larger subarrays
swap(a, pivotIndex, indexFromLeft);
pivotIndex = indexFromLeft;

return pivotIndex;
} // end partition
public static <T extends Comparable<? super T>>
    void quickSort(T[] a, int first, int last)
{
    if (last - first + 1 < MIN_SIZE)
    {
        insertionSort(a, first, last);
    }
    else
    {
        // create the partition: Smaller | Pivot | Larger
        int pivotIndex = partition(a, first, last);

        // sort subarrays Smaller and Larger
        quickSort(a, first, pivotIndex - 1);
        quickSort(a, pivotIndex + 1, last);
    } // end if
} // end quickSort
Quick Sort

• Example
  – Trace the steps that the method quickSort takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3. Assume that MIN_SIZE is 4.
Efficiency of Quick Sort

• For $n$ items
  – $n$ comparisons to find pivot
• If every choice of pivot cause equal sized arrays, recursive calls halve the array
• Results in $O(n \log n)$
Java Class Library

• Class Arrays
  – Uses quick sort for primitive types
    • public static void sort(type[] a)
    • public static void sort (type[] a, int first, int after)
  – Uses merge sort for objects
    • public static void sort(Object[] a)
    • public static void sort (Object[] a, int first, int after)
Radix Sort

• Previously seen sorts on objects that can be compared
• Radix sort does not use comparison
  – Looks for matches in certain categories
  – Places items in “buckets”
• Radix sort is *not appropriate for all data.*
• *It considers the input as if they are strings with the same length*
Radix Sort

(a) 123 398 210 019 528 003 513 129 220 294  Unsorted array

Distribute integers into buckets according to the rightmost digit

210 220 0 1
123 003 513 3
294 4
398 528 5
019 129 6
Radix Sort

Distribute integers into buckets according to the middle digit

(b) 210 220 123 003 513 294 398 528 019 129
Radix Sort

(c) 003  210  513  019  220  123  528  129  294  398

Distribute integers into buckets according to the leftmost digit

003  019   123  129   210  220  294   398
  0       1       2       3       4

513  528
  5       6       7       8       9

(d) 003  019  123  129  210  220  294  398  513  528
Algorithm radixSort(a, first, last, maxDigits)
   // Sorts the array of positive decimal integers a[first..last] into ascending order;
   // maxDigits is the number of digits in the longest integer.
   for (i = 0 to maxDigits - 1)
   {
      Clear bucket[0], bucket[1], ..., bucket[9]
      for (index = first to last)
      {
         digit = digit i of a[index]
         Place a[index] at end of bucket[digit]
      }
      Place contents of bucket[0], bucket[1], ..., bucket[9] into the array a
Radix Sort

• Example
  – Trace the steps that the algorithm radixSort takes when sorting the following array into ascending order:
    
    6340 1234 291 3 6325 68 5227 1638
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Case</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix sort</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n^{1.5})$</td>
<td>$O(n)$</td>
<td>$O(n^2)$ or $O(n^{1.5})$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$n$</td>
<td>$n^1$</td>
<td>$n^{1.5}$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10^2</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>664</td>
<td>9966</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>10,000</td>
<td>31,623</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10^4</td>
<td>10^6</td>
<td></td>
</tr>
<tr>
<td>10^4</td>
<td></td>
<td>10^8</td>
<td></td>
</tr>
<tr>
<td>10^5</td>
<td></td>
<td>10^10</td>
<td></td>
</tr>
<tr>
<td>10^6</td>
<td></td>
<td>10^12</td>
<td></td>
</tr>
<tr>
<td>19,931,569</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Searching

• Like sorting, searching is one of the most common algorithms in programming
• We search a particular item — called target — from a collection of elements
• Different algorithms
  – Linear (sequential) search
  – Binary search
Searching an Unsorted Array

• Iterative search, unsorted array

```java
public boolean contains(T anEntry)
{
    boolean found = false;
    for (int index = 0; !found && (index < numberOfEntries); index++)
    {
        if (anEntry.equals(list[index]))
            found = true;
    } // end for
    return found;
} // end contains
```
(a) A search for 8

Look at 9:

\[
\begin{array}{cccc}
9 & 5 & 8 & 4 \\
\end{array}
\]

8 ≠ 9, so continue searching.

Look at 5:

\[
\begin{array}{cccc}
9 & 5 & 8 & 4 \\
\end{array}
\]

8 ≠ 5, so continue searching.

Look at 8:

\[
\begin{array}{cccc}
9 & 5 & 8 & 4 \\
\end{array}
\]

8 = 8, so the search has found 8.
(b) A search for 6

Look at 9:

```
9 5 8 4 7
```

6 ≠ 9, so continue searching.

Look at 5:

```
9 5 8 4 7
```

6 ≠ 5, so continue searching.

Look at 8:

```
9 5 8 4 7
```

6 ≠ 8, so continue searching.

Look at 4:

```
9 5 8 4 7
```

6 ≠ 4, so continue searching.

Look at 7:

```
9 5 8 4 7
```

6 ≠ 7, so continue searching.

No entries are left to consider, so the search ends. 6 is not in the array.
Recursive Sequential Search of an Unsorted Array

Algorithm to search $a[\text{first}]$ through $a[\text{last}]$ for desiredItem
if (there are no elements to search)
  return false
else if (desiredItem equals $a[\text{first}]$)
  return true
else
  return the result of searching $a[\text{first} + 1]$ through $a[\text{last}]$

private boolean search(int first, int last, T desiredItem)
{
  boolean found;
  if (first > last)
    found = false; // no elements to search
  else if (desiredItem.equals(list[first]))
    found = true;
  else
    found = search(first + 1, last, desiredItem);
  return found;
} // end search
Recursive Sequential Search of an Unsorted Array

- Efficiency of a sequential search of an array

The time efficiency of a sequential search of an array

Best case: $O(1)$
Worst case: $O(n)$
Average case: $O(n)$
Binary Search

• If the data to be searched is sorted, the searching can be made easier.

• For example

\[ a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[n-1] \]

  – We are looking for 7 and we know \( a[5] \) is 9
  – We can ignore elements beyond \( a[5] \) since they are greater than 7
  – This is how Binary search works
Binary Search of a Sorted Array

- Algorithm for binary search

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2  // approximate midpoint
if (first > last)
    return false
else if (desiredItem equals a[mid])
    return true
else if (desiredItem < a[mid])
    return binarySearch(a, first, mid - 1, desiredItem)
else  // desiredItem > a[mid]
    return binarySearch(a, mid + 1, last, desiredItem)
```
(a) A search for 8

Look at the middle entry, 10:

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>18</th>
<th>21</th>
<th>24</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

8 < 10, so search the left half of the array.

Look at the middle entry, 5:

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

8 > 5, so search the right half of the array.

Look at the middle entry, 7:

<table>
<thead>
<tr>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

8 > 7, so search the right half of the array.

Look at the middle entry, 8:

<table>
<thead>
<tr>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

8 = 8, so the search ends. 8 is in the array.
(b) A search for 16

Look at the middle entry, 10:

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>18</th>
<th>21</th>
<th>24</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

16 > 10, so search the right half of the array.

Look at the middle entry, 18:

<table>
<thead>
<tr>
<th>12</th>
<th>15</th>
<th>18</th>
<th>21</th>
<th>24</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

16 < 18, so search the left half of the array.

Look at the middle entry, 12:

<table>
<thead>
<tr>
<th>12</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

16 > 12, so search the right half of the array.

Look at the middle entry, 15:

<table>
<thead>
<tr>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.
Binary Search of a Sorted Array

- Implementation of `binarySearch`

```java
private boolean binarySearch(int first, int last, T desiredItem) {
    boolean found;
    int mid = first + (last - first) / 2;
    if (first > last)
        found = false;
    else if (desiredItem.equals(list[mid]))
        found = true;
    else if (desiredItem.compareTo(list[mid]) < 0)
        found = binarySearch(first, mid - 1, desiredItem);
    else
        found = binarySearch(mid + 1, last, desiredItem);
    return found;
} // end binarySearch
```
Efficiency of a Binary Search of an Array

• Given $n$ elements to be searched
• Number of recursive calls is of order $\log_2 n$

The time efficiency of a binary search

<table>
<thead>
<tr>
<th>Case</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best case</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Worst case</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Average case</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Iterative Sequential Search of an Unsorted Chain
Sequential Search of a Unsorted Chain

- Similar to sequentially searching a sorted array

```java
public boolean contains(T anEntry)
{
    boolean found = false;
    Node currentNode = firstNode;

    while (!found && (currentNode != null))
    {
        if (anEntry.equals(currentNode.getData()))
            found = true;
        else
            currentNode = currentNode.getNextNode();
    }  // end while

    return found;
}  // end contains
```
Recursive Search of a Unsorted Chain

```java
private boolean search(Node currentNode, T desiredItem)
{
    boolean found;

    if (currentNode == null)
        found = false;
    else if (desiredItem.equals(currentNode.getData()))
        found = true;
    else
        found = search(currentNode.getNextNode(), desiredItem);

    return found;
} // end search
```
<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential search (unsorted data)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sequential search (sorted data)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Binary search (sorted array)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>