Search and Sorting

Lecture 8a

Topics

- Search Algorithms
- The Sequential Search Algorithm
- The Binary Search
- Sorting Algorithms
- The Selection Sort
- The Insertion Sort

Search Algorithms

- Searching is the process of finding a target element within a group of items called the search pool
- The target may or may not be in the search pool
- We want to perform the search efficiently, minimizing the number of comparisons
- Let's look at two classic searching approaches: sequential search and binary search

The Sequential Search Algorithm (1 of 3)

- The simplest of all search algorithms
- Search a collection with *unsorted data* (values that occur in arbitrary order)
- The *sequential search algorithm* uses a loop to:
  - sequentially step through an array starting with the first element
  - compare each element with the search value, and
  - stop when
    - the value is found or
    - the end of the array is encountered.
  - Return the first index of the match or -1 if the value is not found

The Sequential Search Algorithm (2 of 3)

- Example: `SearchArray.java`

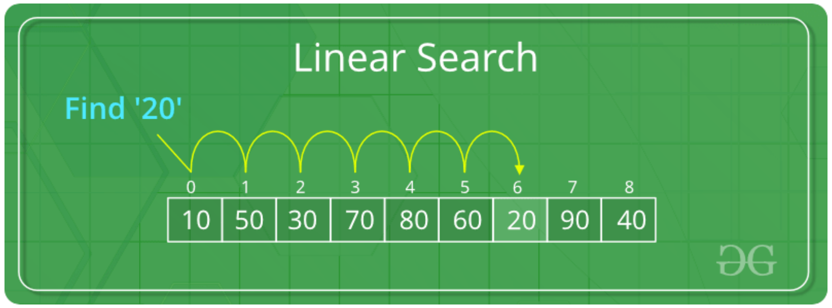
```
public static int sequentialSearch(int[] array, int value)
 // Element 0 is the starting point of the search.
   int index = 0;

   // Store the default values element and found.
   int element = -1;
   boolean found = false;

   // Search the array with a while loop.
   while (!found && index < array.length)
   {
     if (array[index] == value)
     {
       found = true;
       element = index;
     }
     index++;
   }
   return element;
```

```
public static int sequentialSearch(int[] array, int value)
 // Element 0 is the starting point of the search.
 int index = 0;

 // Store the default values element.
 int element = -1;

 // Search the array with a for loop.
 for (index = 0; index < array.length; index++) {
   if (array[index] == value)
   {
     element = index;      // you can return right here
     break;
   }
 }
return element;
```

The Sequential Search Algorithm (3 of 3)

- Point to ponder #1:

  What is the other popular name for the Sequential Search Algorithm?



  In the worst case, how many elements the algorithm will look through?

  n elements (all).

  Can we speed up this search process?

  No.

Binary Search (1 of 5)

- A *binary search* assumes the list of items in the search pool is sorted
- It eliminates a large part of the search pool with a single comparison
- A binary search first examines the middle element of the list -- if it matches the target, the search is over
- If it doesn't, only one-half of the remaining elements need to be searched
- Since they are sorted, the target can only be in that half of the array

Binary Search (3 of 5)

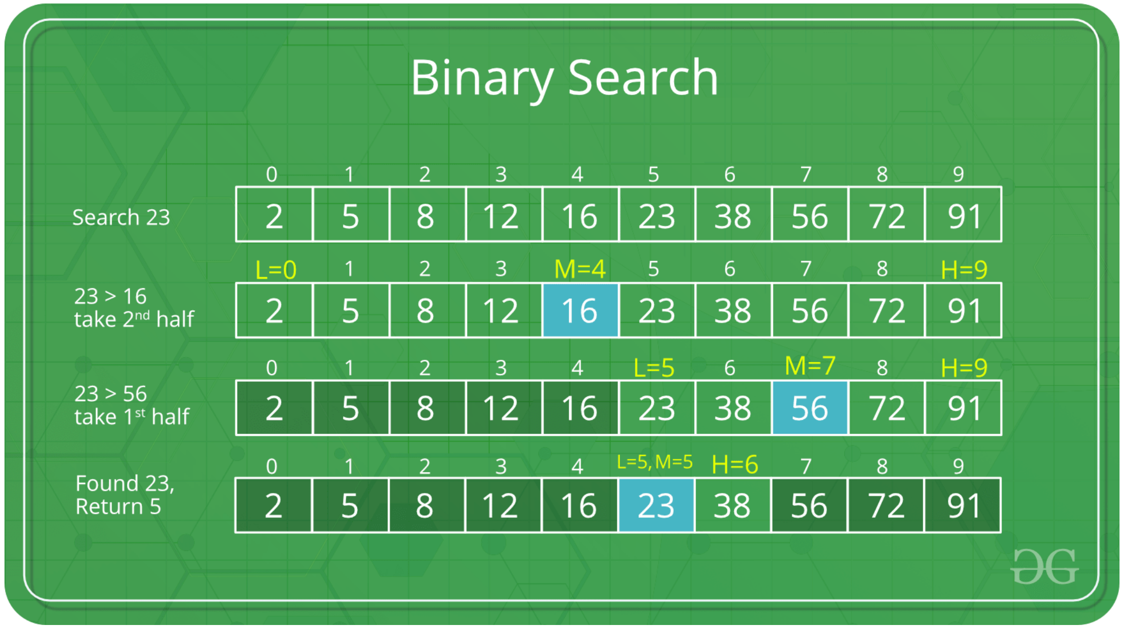
- The process continues by comparing the middle element of the remaining *viable candidates*
- Each comparison eliminates approximately half of the remaining data
- Eventually, the target is found, or the data is exhausted

Binary Search (4 of 5)

```
public static int binarySearch(int[] array,
int value)
{
 int first; // First array element
 int last; // Last array element
 int middle; // Midpoint of search
 int position; // Position of search value
 boolean found; // Flag
 // Set the inital values.
 first = 0;
 last = array.length - 1;
 position = -1;
 found = false;
 // Search for the value.
 while (!found && first <= last)
 {
      // Calculate midpoint
      middle = (first + last) / 2;
      // If value is found at midpoint...
      if (array[middle] == value)
      {
        found = true;
        position = middle;
      }
      // else if value is in lower half...
      else if (array[middle] > value)
         last = middle - 1;
      // else if value is in upper half....
      else
         first = middle + 1;
    }
    // Return the position of the item, or -1
    // if it was not found.
    return position;
}
```

Binary Search (5 of 5)

- The search process



- See example: BinarySearchDemo.java

Binary Search

- Search number = 32

Initialize variables
first = ?
last = ?

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

middle	**first**	**last**

Binary Search

- Search number = 32

Initialize variables
first = 0
last = 8

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

middle	first	last

Binary Search

- Search number = 32

first = 0
last = 8

Calculate middle and see the number is found
Otherwise, update the first and last values

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

middle	first	last

Binary Search

- Search number = 32

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

```
first = 0
last = 8

If 27 = 32 then
  number found (return middle)
else if 27 > 32
  last = middle - 1
else
  first = middle + 1
```

middle	first	last
4	5	8

Binary Search

- Search number = 32

```
first = 0
last = 8

If 27 = 32 then
  number found (return middle)
else if 27 > 32
  last = middle - 1
else
  first = middle + 1
```

middle	first	last
4	5	8

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

Binary Search

- Search number = 32

Calculate middle and see the number is found
Otherwise, update the first and last values

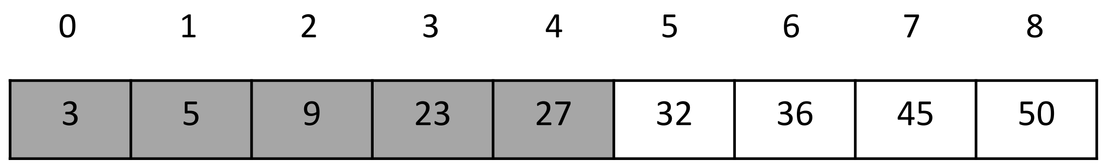
middle	first	last
4	5	8

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

Binary Search

- Search number = 32

```
If 36 = 32 then
  number found (return middle)
else if 36 > 32
  last = middle - 1
else
  first = middle + 1
```



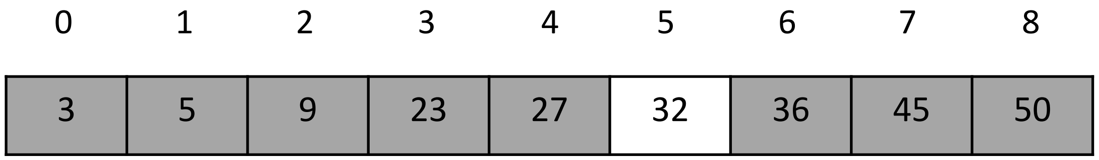
0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

middle	**first**	**last**
4	5	8
6	5	5

Binary Search

- Search number = 32

```
If 36 = 32 then
  number found (return middle)
else if 36 > 32
  last = middle - 1
else
  first = middle + 1
```



middle	first	last
4	5	8
6	5	5

Binary Search

- Search number = 32

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

Calculate middle and see the number is found
Otherwise, update the first and last values

middle	first	last
4	5	8
6	5	5

Binary Search

- Search number = 32

```
If 32 = 32 then
  number found (return middle)
else if 36 > 32
  last = middle - 1
else
  first = middle + 1
```

0	1	2	3	4	5	6	7	8
3	5	9	23	27	32	36	45	50

middle	first	last
4	5	8
6	5	5
5		

Point to ponder #1:

On average, how many elements the algorithm will look through?

log(n)

Search and Sorting

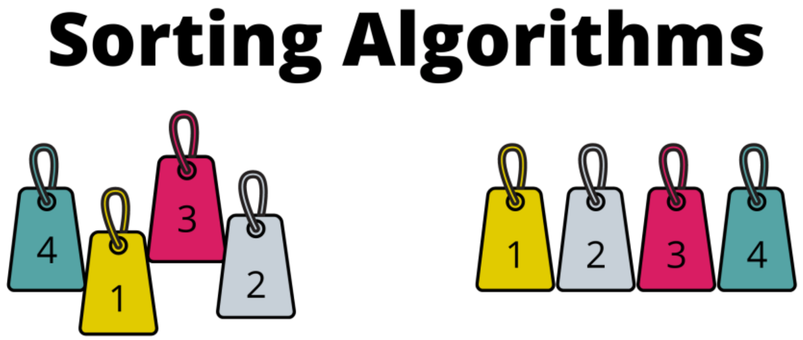
Lecture 8b

Topics

- Search Algorithms
- The Sequential Search Algorithm
- The Binary Search
- Sorting Algorithms
- The Selection Sort
- The Insertion Sort

Sorting (1 of 2)

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific value(s)
    - sorting a list of test scores in ascending numeric order
    - sorting a list of people alphabetically by last name



Sorting (2 of 2)

- There are many algorithms, which vary in efficiency, for sorting a list of items
- We will examine two specific algorithms:
  - Selection Sort
  - Insertion Sort

Selection Sort (1 of 13)

- Steps:
    - find the smallest value in the array
    - switch it with the value in the first position
    - find the next smallest value in the array
    - switch it with the value in the second position
    - repeat until all values are in their proper places

- See example: SelectionSortDemo.java

Selection Sort (2 of 13)

- An example:

Original	5	7	2	8	9	1

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Selection Sort (3 of 13)

- An example:

Original	5	7	2	8	9	1
smallest is 1:	1	7	2	8	9	5

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Selection Sort (4 of 13)

- An example:

Original	5	7	2	8	9	1
smallest is 1:	1	7	2	8	9	5
smallest is 2:	1	2	7	8	9	5

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Selection Sort (5 of 13)

- An example:

Original	5	7	2	8	9	1
smallest is 1:	1	7	2	8	9	5
smallest is 2:	1	2	7	8	9	5
smallest is 5:	1	2	5	8	9	7

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Selection Sort (6 of 13)

- An example:

Original	5	7	2	8	9	1
smallest is 1:	1	7	2	8	9	5
smallest is 2:	1	2	7	8	9	5
smallest is 5:	1	2	5	8	9	7
smallest is 7:	1	2	5	7	9	8

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Selection Sort (7 of 13)

- An example:

Original	5	7	2	8	9	1
smallest is 1:	1	7	2	8	9	5
smallest is 2:	1	2	7	8	9	5
smallest is 5:	1	2	5	8	9	7
smallest is 7:	1	2	5	7	9	8
smallest is 8:	1	2	5	7	8	9

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Selection Sort (8 of 13)

- Another example:

Original	2	3	7	4

Selection Sort (9 of 13)

- Another example:

Original	2	3	7	4
smallest is 2:	2	3	7	4

Selection Sort (10 of 13)

- Another example:

Original	2	3	7	4
smallest is 2:	2	3	7	4
smallest is 3:	2	3	7	4

Selection Sort (11 of 13)

- Another example:

Original	2	3	7	4
smallest is 2:	2	3	7	4
smallest is 3:	2	3	7	4
smallest is 4:	2	3	4	7

Point to ponder #1:

What is the total number of machine operations this algorithm always takes regardless of how correctly are the numbers sorted at the beginning? (time spent on the sorting)?

It is of the order n^2.

Selection Sort (12 of 13)

- Swapping
  - The processing of the selection sort algorithm includes the *swapping* of two values
  - Swapping requires **3 assignment** statements and a temporary storage location:

	first	second	temp
`temp = first;`	9	3	9
`first = second;`	3	3	9
`second = temp;`	3	9	9

Selection Sort (13 of 13)

```
public static void selectionSort (int[] numbers)
{      int min, temp;
       for (int index = 0; index < numbers.length-1; index++) {
          min = index;
          for (int scan = index+1; scan < numbers.length; scan++)
             if (numbers[scan] < numbers[min])
                min = scan;
          // swap the values
          temp = numbers[min];
          numbers[min] = numbers[index];
          numbers[index] = temp;
      }
}  // method selectionSort
```

Insertion Sort (1 of 14)

- Steps:
  - consider the first item to be a sorted sublist (of one item)
  - insert the second item into the sorted sublist, shifting the first item as needed to make room to insert the new addition
  - insert the third item into the sorted sublist (of two items), shifting items as necessary
  - repeat until all values are inserted into their proper positions

Insertion Sort (2 of 14)

- An example:

Original	11	9	16	5	7	3

Insertion Sort (3 of 14)

- An example:

Original	11	9	16	5	7	3

- The first item is a sorted sublist

Insertion Sort (4 of 14)

- An example:

Original	11	9	16	5	7	3
insert 9:	9	11	16	5	7	3

- Insert items into the sorted sublists, shifting items as necessary

Insertion Sort (5 of 14)

- An example:

Original	11	9	16	5	7	3
insert 9:	9	11	16	5	7	3
insert 16:	9	11	16	5	7	3

- Insert items into the sorted sublists, shifting items as necessary

Insertion Sort (6 of 14)

- An example:

Original	11	9	16	5	7	3
insert 9:	9	11	16	5	7	3
insert 16:	9	11	16	5	7	3
insert 5:	5	9	11	16	7	3

- Insert items into the sorted sublists, shifting items as necessary

Insertion Sort (7 of 14)

- An example:

Original	11	9	16	5	7	3
insert 9:	9	11	16	5	7	3
insert 16:	9	11	16	5	7	3
insert 5:	5	9	11	16	7	3
insert 7:	5	7	9	11	16	3

- Insert items into the sorted sublists, shifting items as necessary

Insertion Sort (8 of 14)

- An example:

Original	11	9	16	5	7	3
insert 9:	9	11	16	5	7	3
insert 16:	9	11	16	5	7	3
insert 5:	5	9	11	16	7	3
insert 7:	5	7	9	11	16	3
insert 3:	3	5	7	9	11	16

Insertion Sort (9 of 14)

- Another example:

Original	2	3	4	7

Insertion Sort (10 of 14)

- Another example:

Original	2	3	4	7

Insertion Sort (11 of 14)

- Another example:

Original	2	3	4	7
insert 3:	2	3		

Insertion Sort (12 of 14)

- Another example:

Original	2	3	4	7
insert 3:	2	3		
insert 4:	2	3	4	

Insertion Sort (13 of 14)

- Another example:

Original	2	3	4	7
insert 3:	2	3		
insert 4:	2	3	4	
insert 7:	2	3	4	7

Point to ponder #2:

What is the total number of machine operations that this algorithm performs in general to sort an array?

Depends on how numbers sorted at the beginning: n for the best case and n^2 for the worst.

Insertion Sort (14 of 14)

```
public static void insertionSort (int[] numbers)
{
    for (int index = 1; index < numbers.length; index++)
    {
        int key = numbers[index]; //value to be added
        int position = index;
        // shift larger values to the right
        while (position > 0 && numbers[position-1] > key)
        {
          numbers[position] = numbers[position-1];
          position--;
        }
        numbers[position] = key;

   }
}  // method insertionSort
```

Comparing Sorts

- In general (what that means?), the Selection and Insertion sort algorithms are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts are of *order* n^2
- Other sorts are more efficient: *order* $n \log_2 n$
- Interesting visualization: https://www.youtube.com/watch?v=ZZuD6iUe3Pc