Searching and Sorting

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

Searching in an Array

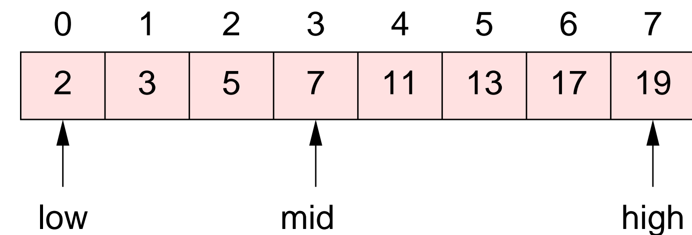
How can we find an element in an array?

- Sequential search:
  - Run through array from the first index to the last and compare each element with given key.
  - Array has n elements: at most n comparisons have to be performed.
  - Algorithm is linear in the length of the array.
    * If the array length doubles, the searching time doubles.
- Binary search:
  - Faster method that is based on the divide and conquer strategy.
    * Reduce the problem to a similar problem of smaller size.
  - Only applicable if array is sorted.
  - Array elements are arranged in some (e.g. ascending) order.

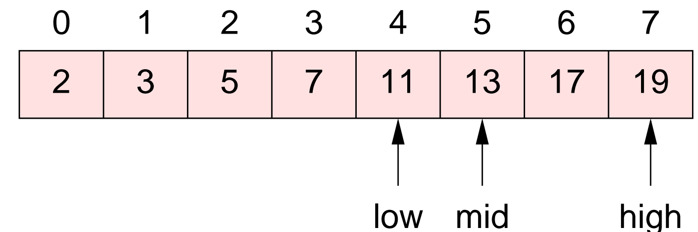
Two fundamental methods for searching in an array.

Binary Search

- We search for element 13 in index range low to $high$.
  - Initially: $low = 0$, $high = 7$.
  - $mid = (low + high)/2$.



- Take a look at element in the middle.
  - If element at mid equals 13, we are done.
  - If element is less than 13, the element can only be at the right of mid.
    - $low := mid + 1$.
  - Otherwise, element must be to the left.
    - $high := mid - 1$.



Process continues until element is found or $low > high$.

Applet

Applet Viewer: Ordered.class

Applet

New Fill Ins Find Del

Linear

Binary

Number: 738

Checking index 44, range = 30 to 59

0	51	12	262	24	395	36	595	48	767
1	71	13	265	25	408	37	603	49	776
2	82	14	276	26	415	38	634	50	789
3	89	15	277	27	459	39	666	51	792
4	91	16	300	28	463	40	672	52	803
5	135	17	306	29	481	41	680	53	807
6	140	18	308	30	512	42	687	54	831
7	163	19	310	31	523	43	710	55	842
8	183	20	311	32	535	44	713	56	941
9	200	21	340	33	560	45	737	57	991
10	206	22	358	34	590	46	738	58	993
11	232	23	381	35	594	47	765	59	998

Applet started.

Method Specification

```
// -----------------------------------------------
// i = binarySearch(a, x)
// 'i' is the index of 'x' in 'a'.
//
// Input:
//   'a' is not null and sorted in ascending order.
// Output:
//   'a' remains unchanged.
//   if 'i' == -1, then 'x' is not in 'a'.
//   if 'i' != -1, then 'a[i] == x'.
// -----------------------------------------------
public static int binarySearch(int[] a, int x)
```

Method Implementation

```
public static int binarySearch(int[] a, int x)
{
  int low = 0;
  int high = a.length-1;
  while (low <= high)
  {
    int mid = (low + high)/2;
    if (a[mid] == x) return mid;
    if (x > a[mid])
      low = mid+1;
    else
      high = mid-1;
  }
  return -1;
}
```

Analysis

How many comparison operations are needed at most?

- In every iteration, the size of the search range is halved.
  - The difference between $high$ and low is divided by a factor of 2.
- Iteration number is at most the binary logarithm of array length.
  - $\#\mathrm{iterations} = \log_2 a.\mathrm{length}$.
  - The algorithm is logarithmic in the length of the array.
  - If the array length doubles, the searching time only increases by a constant.
- Example: $a.\mathrm{length} = 1024 = 2^{10}$
  - Linear search: at most 1024 comparison operatons.
  - Binary search: at most 10 comparison operations.

Binary search has lower runtime complexity than linear search.

Sorting an Array

Rearrange the elements of an array according to some ordering.

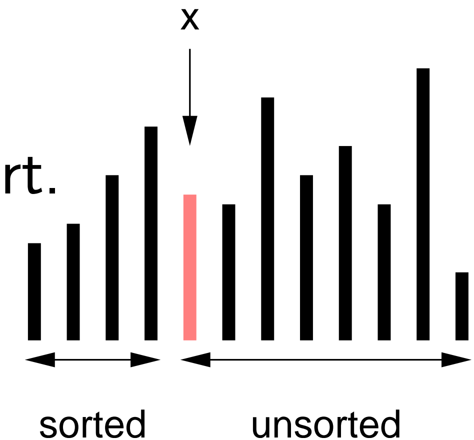
- Multiple orderings are possible.
  - Array elements may be objects containing personal data.
  - Array may be sorted according to last name or according to age.
- Sorting an array allows to lookup elements faster.
  - We can apply binary search rather than linear search.
- Sorting is a well-studied problem in computer science.
  - Various sorting algorithms developed until the 1970s.
  - A simple sorting algorithm: insertion sort.

We will study insertion sort now.

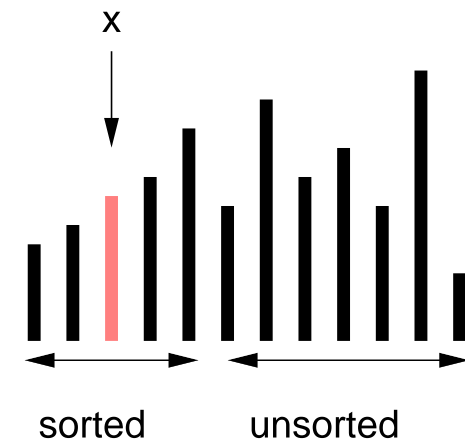
Insertion Sort

Sort an array like you would sort a deck of cards.

- Array has sorted left part and unsorted right part.
  - At iteration i, the first i elements are sorted (initially $i = 0$).



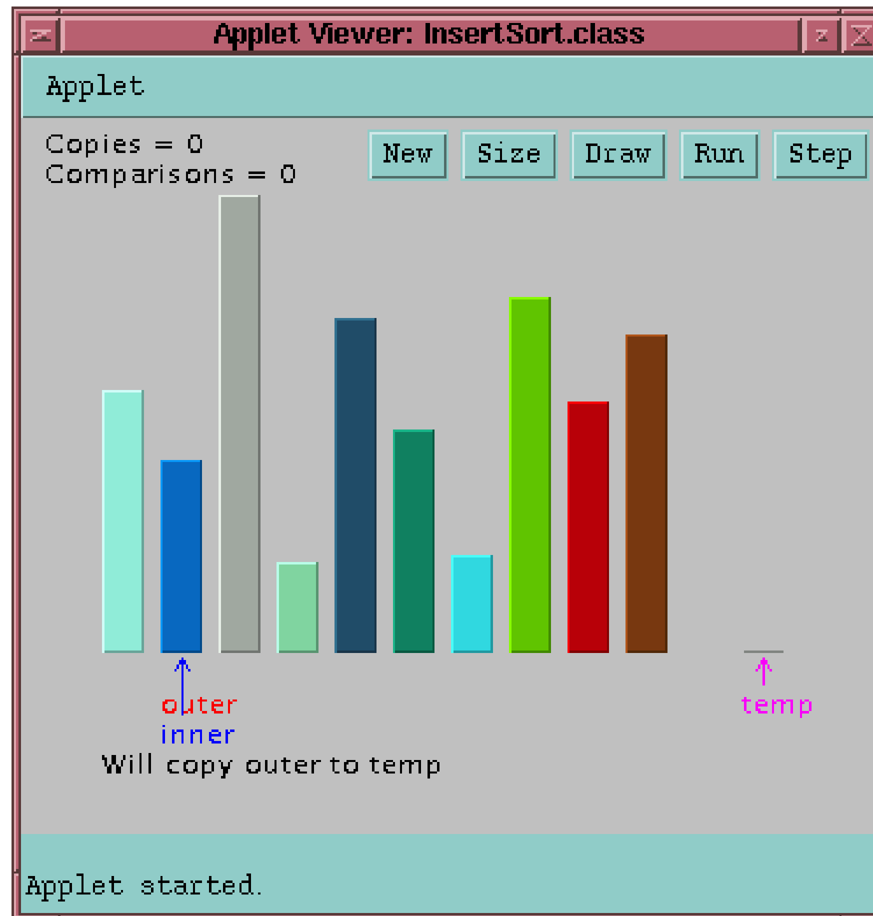
- Take element x at index $(i + 1)$.
  - Insert it into the sorted part to the left of x.
  - Shift elements larger than x one position to the right.



- First $i + 1$ elements sorted.
  - Process can be repeated with iteration $i + 1$.

After n iterations, an array of length n is sorted.

Applet



Method Specification

```
// ------------------------------------------------
// insertionSort(a)
// sort 'a' by the insertion sort algorithm.
//
// Input:
//   'a' is not null.
// Output:
//   'a' has the same elements as the old 'a'.
//   'a' is sorted.
// ------------------------------------------------
public static void insertionSort(int[] a)
```

Method Implementation

```
public static void insertionSort(int[] a)
{
  final int n = a.length;
  for (int i = 1; i < n; i++)
  {
    // a is sorted from 0 to i-1
    int x = a[i];
    int j = i-1;
    while (j >= 0 && a[j] > x)
    {
      a[j+1] = a[j];
      j = j-1;
    }
    a[j+1] = x;
 }
}
```

Analysis

How many comparison operations are needed at most?

- Iterations:
  - 1. iteration: 1, 2. iteration: 2, . . . , $n-1$. iteration: $n-1$ comparisons.
- Total: $\approx n^2/2$ comparisons.

$$1 + 2 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = n * (n-1)/2 \approx n^2/2$$

  - The algorithm is quadratic in the length n of the array.
    - If the array length doubles, the sorting time quadruples.
- Average: $\approx n^2/4$ comparisons at most.
  - In average, the element x is inserted in the middle of the unsorted part.

Sorting time grows much faster than array length.