Java code: look at class SearchElement array

return false;

You have unsuccessfully examined all elements of array:

and value v in array: return true, or

either

Algorithm: examine elements of array in some order. If you

find value v in array: return true, or

Input: unsorted array A, value v

Linear Search:

middle

| 6 | 14 | 11 | 1 | 7 | 7 | 7 | 14 |

Binary Search: sorted array

Search for 20:

| 7 | 14 | 11 | 1 | 7 | 7 |

Linear search: unsorted array

Organization

in Arrays

Sorting and Searching

Quicksort: divide and conquer algorithm, in-place sort

merge sort: divide and conquer algorithm, sort

Selection sort:

1. Searching in Arrays: Linear search and Binary search

2. Sorting Arrays:
Java code: Look at class SearcherArray

\[ \text{search left half of array} \]
\[ \text{search right half of array} \]
\[ \text{return}\]

Algorithm: Think about looking up element directory

1. Input: Sorted array A[0..n-1], value v

Binary search

We will discuss this format later.

We will discuss this format later.

We will discuss this format later.

We will discuss this format later.

We will discuss this format later.
Let us study the second option:

1. Write separate code for searching/sorting in a real, don't деле.
   What do we do if we want to search/sort integers? Two options:
   - In arrays of type `Comparable`
   - In arrays of type `Comparable`

How? Integers, reals etc are not objects, so they cannot be stored
in arrays of type `Comparable`.

Define a class called `BoxInteger`

```java
define a class called BoxInteger
... (Comparable) extends BoxInteger()
   if (Comparable) extends BoxInteger() {
      new BoxInteger(0); 
      if (Comparable) extends BoxInteger() {
         box the integer into an object
      } else {
         return (Comparable) extends BoxInteger() {
            return (Comparable) extends BoxInteger() {
               return (Comparable) extends BoxInteger() {
                  return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                    return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (Comparable) extends BoxInteger() {
                     return (C...
with the ith element of the array...

3. In general, at the ith step, examine array elements from i to $N - 1$, finding the smallest one in that range, and exchange it with the ith element of the array.

Algorithm: Assume $N$ is size of array.

1. Input: Some array sorted in ascending order

2. Examine all elements from 0 to ($N - 1$), finding the smallest one

3. Examine all elements from 1 to ($N - 1$), finding the smallest one

Input array of Comparables

<table>
<thead>
<tr>
<th>Question: can we find a way to speed up selection sort?</th>
</tr>
</thead>
</table>

We will see a formal definition of $O(N^2)$ in section later.

Both merge sort and quick sort have $O(N \log N)$ time.

Selection sort has $O(N^2)$ time and is not the fastest to be selected.

It is easy to show that selection sort requires $\frac{N}{2} \left( \frac{N-1}{2} \right)$ evaluations.

Sorting algorithms: the first phase:

Binary search works great but how do we create a sorted array in

<table>
<thead>
<tr>
<th>int[] arr = new int[N];</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int i = 0; i &lt; N; i++)</td>
</tr>
<tr>
<td>arr[i] = random();</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Selection sort:</td>
</tr>
<tr>
<td>Merge sort: $O(N \log N)$ time</td>
</tr>
</tbody>
</table>
Can we apply the divide and conquer approach to sorting?

Rationale: The partitioning and assembly process should not be independent; we can balance the runtime time.

If we divide problem into two subproblems then can be solved

Rough outline: Suppose you break problem into 2 process. Each

and assemble final solution

Any time you have a \( O(N^2) \) algorithm, it pays to break the

Merge sorted arrays \( a_i \) and \( a_j \)

- Keep three indices: \( p1 \), \( p2 \), \( p3 \)
- \( i \) into \( p1 \), \( j \) into \( p2 \), \( p1 \) into \( p3 \)
- Create an array in whose size is size of \( a_i \) + size of \( a_j \)

Merged array

Array 1

Array 2

Divide array into two equal parts. Sort each part and merge

Quicksort

Three cases:

4 6 7 7 4

p1

p2

p3

Array 1

Array 2

How do we merge sorted arrays? Let us write some code

How do we sort the parts? Call merge-sort recursively

How do we divide array into two equal parts? Use indices into


Asymptotic complexity of merge sort: $O(n \cdot \log(n))$

Disadvantage of merge sort: need extra storage for temporary arrays. In practice, this can be a disadvantage even though merge sort is

an asymptotically optimal algorithm for sorting.

Good sorting algorithm that does not use extra array storage:

QuickSort

Intuitive idea:

Given an array $A$ and a pivot value $v$:

- Partition array elements into two subarrays $S_X$ and $S_Y$.
- $S_X$ contains only elements less than or equal to $v$.
- $S_Y$ contains only elements greater than $v$.
- Sort $X$ and $Y$ separately.
- Concatenate sorted $X$ and sorted $Y$ to produce result.

Main advantage of QuickSort: sorting can be done in-place (no extra array needs to be created).

Key problem to be solved: how do we partition array in place?

If we can do this, we can have a QuickSort method of form:

```
void quicksort(int[] A, int i, int j) { /sort values between i and j
    // implementation
}
```

where $i$ is lower bound of subarray to be sorted, and $j$ is upper bound.
Heuristic: use first array value as pivot. However, median is expensive to compute exactly.

Some people use middle element, and others use median of first, middle and last element.

We need to keep track of two indices: \( \text{LEFT} \) and \( \text{RIGHT} \). We swap the read and blue balls under the blue ball, and make progress again.

If ball is read or blue, then we advance \( \text{LEFT} \) index.

If ball is red, then we advance \( \text{RIGHT} \) index.

How can we get all blue balls to the left of all red balls?

Our partition code is somewhat more inefficient than it needs to be.

Note our QuickSort code.

Sentinel:

Choices for any one of bounds can be done more efficiently with

Inplace Partitioning
Many applications however require more dynamic data structures.

Key points
- **Comparable interface** to permit handling of more general types
- Started with arrays of int
- 3. Write generic code for linear search:
  `((u)(v))O
  ; 2. Sort by array: selection sort `O
  `((u)(log))O
  ; 1. Search by array: linear search `O
  `((u)(log))O

**Summary**

java class library: uses merge sort for stability reasons.

- Quicksort: not stable.
- Heuristic:
  - `according to number of equal objects, always take entry from
  - higher sort: easy to make it stable if we are careful with merge
  - Stability is important when you must make multiple sorting passes
  - array is same as the order in original array
- Sorting algorithm is stable if the order of equal elements in sorted

**Stable sorting methods**

- **Merge sort**: not stable.
- Heuristic: