Searching and Sorting

- It’s a fundamental area of CS since the beginning of time.
  → Well understood!
  → All professional programmers should have a good understanding of it.
  → Also provides good example for discussion of computational complexity and correctness.
- Our model:
  → Will assume array of integers:
  
  ```java
  int A[];
  final int N = A.length;
  ```
- Once again, it’s easy to translate the ideas for use with Sortable arrays using lessThan, greaterThan, and equals rather than <, >, ==.
- When sorting, want to arrange into non-decreasing order.

Searching

- Linear search
- Binary search
- Searching through various ADTs and data structures
  e.g., BSTs, hash tables, heaps
  → Will discuss appropriate invariants and correctness of these approaches.

Linear Search

```java
// Goal: Return int k such that A[k] is the first occurrence of value x in A,
// or N if x is not in A.
// i.e. 0 <= k <= N && x not in A[0..k-1]
// && (k=N || A[k] == x)
public static int linSearch (int x, int []A) {
  final int N = A.length;
  int k=0;
  // invariant: 0 <= k <= N && x not in A[0..k-1]
  // Note: the order of these conditions
  // is VERY important.
  while (k<N && A[k] != x) {
    k++;
  }
  return k;
  // Likely want a wrapper to return -1
}
```

Binary Search

```java
// Goal: return int k such that -1 <= k <= N
// and A[0..k] <= x < A[k+1..N-1]
// Pre-condition: A is sorted.
public static int binSearch (int x, int []A) {
  final int N = A.length;
  int lo=-1, hi=N-1;
  // invariant: -1 <= lo < hi <= N-1;
  // && A[0..lo] <= x < A[hi..N-1]
  while (lo+1 != hi) {
    // Note integer division.
    final int mid = (lo+hi) / 2;
    // -1 <= lo < mid < hi <= N-1;
    if (A[mid] <= x) {
      lo = mid;
    } else {
      hi = mid;
    }
  }
  return lo;
}
```
Notes on this Version of Binary Search

- Works for empty list (N==0)
- Altered slightly to make invariant easier to discuss.
  → If x is in A, get rightmost occurrence.
  → If x is NOT in A, get the position after which it belongs.
Which means ...
  → Caller has to check that result is in the range 0..N-1 and that A[k] == x.

- Also while it has the same computational complexity as the other versions, it runs a bit faster than the one we showed you last time that checks for equality inside the loop. [i.e., there is one check instead of two.]

Some Useful Aux. Functions

We’ll use a few aux. functions in our discussions, Assert, Sorted, AllLE, AllGE.

→ Useful for discussing correctness.

→ Might want to leave them in while developing and testing the programs ... BUT must leave them out of final version or the running time will be awful.

→ In our running time discussions, we will assume these functions are not used.

```java
class Sorting {
    // Causes the program to die if the condition is false.
    public static void Assert (boolean condition) {
        if (!condition) {
            System.out.println("Assertion failure!");
            System.exit (1);
        }
    }
    // Returns true iff A[lo..hi] is sorted in non-decreasing order.
    public static boolean Sorted (int A[], int lo, int hi) {
        for (int i=lo; i<lo; i++) {
            if (A[i] > A[i+1]) {
                return false;
            }
        }
        return true;
    }
    // Returns true iff A[lo..hi] is all less than or equal to val.
    public static boolean AllLE (int A[], int lo, int hi, int val) {
        for (int i=lo; i<=hi; i++) {
            if (A[i] > val) {
                return false;
            }
        }
        return true;
    }
    // Returns true iff A[lo..hi] are all greater than or equal to val.
    public static boolean AllGE (int A[], int lo, int hi, int val) {
        for (int i=lo; i<=hi; i++) {
            if (A[i] < val) {
                return false;
            }
        }
        return true;
    }
}
```

Sorting: Selection Sort

Basic idea:

→ Each time through, find $k^{th}$ largest element $(k = 1, ..., N)$ and put it into $A[N-k]$.

\[
\begin{array}{ccc}
0 & k & N-1 \\
A & \text{< A[k], unsorted} & \text{>= A[k], sorted} \\
\end{array}
\]

Insertion Sort

Basic idea:

- Start with empty list.
- Progressively add $k^{th}$ element into its correct place in the sublist.

\[i.e., A[0..0] \text{ is sorted} \]
\[A[0..1] \text{ is sorted} \]
\[A[0..2] \text{ is sorted} \]
\[\vdots \]
\[A[0..k-1] \text{ is sorted} \]
\[\vdots \]
\[A[0..N-1] \text{ is sorted} \]

\[
\begin{array}{ccc}
0 & k & N-1 \\
A & \text{values originally in A[0..k-1] but are now sorted} & ??? \\
\end{array}
\]
Computational Complexity of Sorts

- Selection sort:
- Insertion sort:
- Other approaches?

QuickSort

Clever and subtle method due to Hoare.

Idealized version:

- Find “middle” element in the array and fix it in its correct place (at $A[N/2]$).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(N/2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;= pivot</td>
<td>pivot</td>
</tr>
<tr>
<td>(N-1)</td>
<td>&gt;= pivot</td>
<td></td>
</tr>
</tbody>
</table>

- *Partition* the list by placing all elements less than the middle in the first half of the list and those greater in the second half of the list.

- Perform the above two steps recursively on each half of the list. We’re done.

QuickSort

BUT ... there is no fast way to find the middle element of the list.

→ Instead, we just pick an arbitrary element (called the pivot) and hope it’s a reasonable choice.

- Any choice of pivot OK for correctness.
- Process will be most efficient if pivot is middle element (size-wise) in list.

How long does it take to choose the pivot now?

→ We loop through the list, *partitioning* as we go until we have found the correct place for the pivot element. Insert it there.

→ Now we recursively perform this whole process to each “half” of the list.

```java
public static void QuickSort (int A[], int first, int last) {
    if (first < last) {
        int lo = first, hi = last+1;
        final int pivot = A[lo];
        // A[lo] is now "vacant."
        while (true) {
            while (true) {
                hi = hi - 1;
                if (lo==hi || A[hi]<pivot) break;
            }
            if (lo==hi) break;
            A[lo] = A[hi];
            while (true) {
                lo = lo + 1;
                if (lo==hi || A[lo]>pivot) break;
            }
            if (lo==hi) break;
            A[hi] = A[lo];
        }
        A[hi] = pivot;
        Assert (AllLE (A, first, hi-1, pivot) && AllGE (A, hi+1, last, pivot));
        QuickSort (A, first, hi-1);
        Assert (Sorted (A, first, hi-1));
        QuickSort (A, hi+1, last);
        Assert (Sorted (A, hi+1, last));
    }
}
```
QuickSort

Here is the detailed picture:

\[
\begin{array}{cccccc}
\text{pivot} & A & \text{first} & \text{lo}-1 & \text{lo} & \text{hi}-1 & \text{hi} & \text{last} \\
\text{\textless{}= pivot} & \text{???} & \text{\textgtr{}= pivot} \\
\end{array}
\]

Some notes:

- We use \texttt{break} to exit from loops. Some people think this is a bad idea.