Searching and Sorting in Arrays

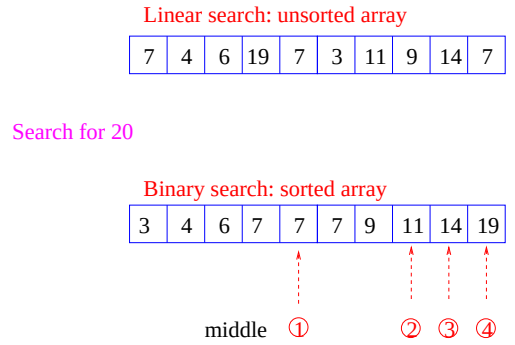
Organization

1. Searching in arrays: linear search and binary search
2. Sorting arrays:
   - Selection sort
   - Merge sort: divide and conquer algorithm
   - Quicksort: divide and conquer algorithm, in-place sort
3. Asymptotic complexity: big-O notation

Linear search:

- Input: unsorted array A, value v
- Output: return true if value is in array, false otherwise
- Algorithm: examine elements of array A in some order till you either
  - find value v in array: return true, or
  - you have unsuccessfully examined all elements of array: return false

Java code: look at class SearchIntArray



If you run binary search and linear search on large arrays, you will find that binary search runs much faster than linear search.

How do we capture this difference formally?

Asymptotic complexity:

Linear search: $O(n)$ where n is number of elements in array

Intuitively, time taken by linear search grows proportionately with array size.

Binary search: $O(log_2(n))$

Intuitively, time taken by binary search grows proportional to the log of the array size (much slower growth).

We will discuss this formally later.

Binary search:

- Input: sorted array A[0..n-1], value v
- Output: return true if value is in array; false otherwise
- Algorithm: think about looking up telephone directory
  - let m be the middle element of the array (element A[n/2])
  - if (m = v) return true
  - if (m < v), search right half of array (A[n/2 + 1]..A[n-1])
  - if (m > v), search left half of array (A[0]..A[n/2 - 1])

Java code: look at class SearchIntArray

Problem with code in class SearchIntArray: works only for integer arrays

Linear and binary search algorithm works for arrays of any type of object that can be compared.

Question: how can we write a generic binary search method that works for all such types of objects, not just integers.

Important Java interface: Comparable

```
public interface Comparable{
 int compareTo(Object o);
}
```

Import from java.util.*

compareTo is instance method that should return

- a negative number if *this* is less than o,
- 0 if *this* is equal to o
- a positive number if *this* is greater than o

Predefined classes like String implement this interface (eg, compareTo orders strings by dictionary order).

If you are implementing a class whose instances you want sorted/searched etc., implement compareTo for your class, and then use existing searching/sorting code for these procedures!

We can now write linear and binary search so that they are polymorphic, and will work for arrays of Comparable elements.

Change from integer array code: types of arrays are Comparable[] and instead of integer comparisons, we use compareTo method.

Java code: see class SearchComparableArray

From now on, we will use Comparable in our discussions of searching and sorting.

Let us look at an example of defining the compareTo method.

Class StudentRecord shows how to define a compareTo method for student records to sort student records:

- order student records by last name
- if last names are same, order by first name
- if first names are also same, order by ID
- Pingali is higher than everyone else though

Irony: integers, reals etc are not objects, so they cannot be stored in arrays of type Comparable[ ]!

What do we do if we want to search/sort integers? Two options:

1. Write separate code for searching/sorting int, real, double, Comparable etc. (bad)
2. Define a new class which can store an integer for example and which implements Comparable. Stick those objects into arrays of type Comparable[ ]!

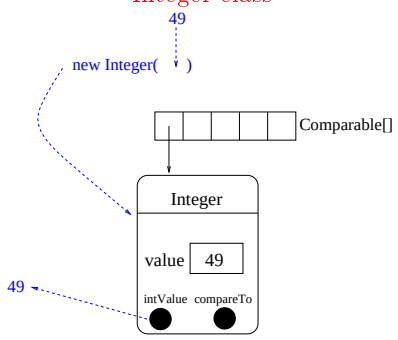
Let us study the second option.

Define a class called `boxedInteger`

```
class boxedInteger implements Comparable{
  protected int value;
  public boxedInteger(int v) {
     value = v;}
  public int intValue() { //extract value of integer
    return value;}
  public int compareTo(Object o) {
    if (!(o instanceof boxedInteger))
       return +1;
    return (value - ((boxedInteger)o).value);}
}....
int n = 5;
boxedInteger b = new boxedInteger(n); //box the integer into an object
int j = b.intValue(); //should return 5
if (b.compareTo(new boxedInteger(7)) > 0) ...
```

Java class library: provides this class to you already!
Name of class: `Integer`
Similar boxed classes: Long, Float, Character, Boolean, Double

Integer class



Sorting

Binary search works great but how do we create a sorted array in the first place?

Sorting algorithms:

- Selection sort: $O(n^2)$ time
- Merge sort: $O(nlog(n))$ time
- Quicksort: $O(n^2)$ time

SelectionSort

- Input: array of Comparables
- Output: same array sorted in ascending order
- Algorithm: assume N is size of array
  1. Examine all elements from 0 to (N-1), find the smallest one and swap it with the 0^{th} element of the input array.
  2. Examine all elements from 1 to (N-1), find the smallest one in that portion of the array and swap it with the 1^{st} element of the array.
  3. In general, at the i^{th} step, examine array elements from i to $(N-1)$, find the smallest one in that range, and exchange it with the i^{th} element of the array.

It is easy to show that selection sort requires *N*(N-1)/2* comparisons, if N is the length of the array to be sorted.

This is called an $O(N^2)$ algorithm since the leading term is N^2 (we also ignore constant coefficients in big-O notation).

We will see a formal definition of big-O notation later.

Question: can we find a way to speed up selection sort?

MergeSort

Quintessential divide-and-conquer algorithm

Divide array into two equal parts, sort each part and merge

Three questions:

- How do we divide array into two equal parts? Use indices into array
- How do we sort the parts? Call merge-sort recursively
- How do we merge sorted arrays? Let us write some code.

Any time you have a $O(N^2)$ algorithm, it pays to break the problem into smaller subproblems, solve subproblems separately and assemble final solution.

Rough argument: suppose you break problem into k pieces. Each pieces takes $O((N/k)^2)$ time, so time for doing all k pieces is roughly $\frac{1}{k}O(N^2)$ time.

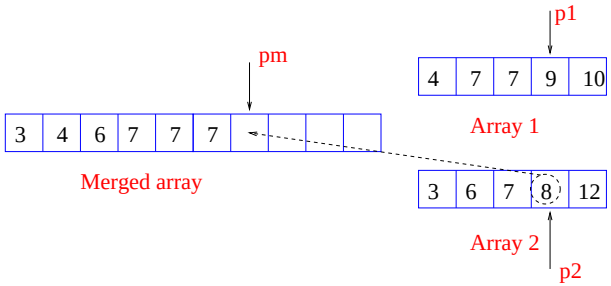
If we divide problem into two subproblems that can be solved independently, we can halve the running time!

Caveat: the partitioning and assembly process should not be expensive.

Can we apply this divide and conquer approach to sorting?

Merging sorted arrays a1 and a2

- Create an array m whose size = size of a1 + size of a2
- Keep three indices: p1 into a1, p2 into a2, pm into m, all initialized to 0.
- Compare first elements of a1 and a2, and move the smaller one (say it is from a2) into first element of m. Increment p2 and pr.
- In general, compare element a1[p1] with a2[p2] and move the smaller one into m[pm], incrementing the appropriate indices.
- If either a1 or a2 is emptied out, copy over all remaining elements of the other array into m.



QuickSort

Intuitive idea:

- Given an array A and a *pivot value* v
- Partition array elements into two subarrays SX and SY
- SX contains only elements less than or equal to v
- SY contains only elements greater than v
- Sort SX and SY separately
- Concatenate sorted SX and sorted SY to produce result

Code for merge sort: see class testSort

Asymptotic complexity of merge sort: $O(n * log_2(n))$

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 algorithms for sorting.*

Good sorting algorithm that does not use extra array storage: Quicksort

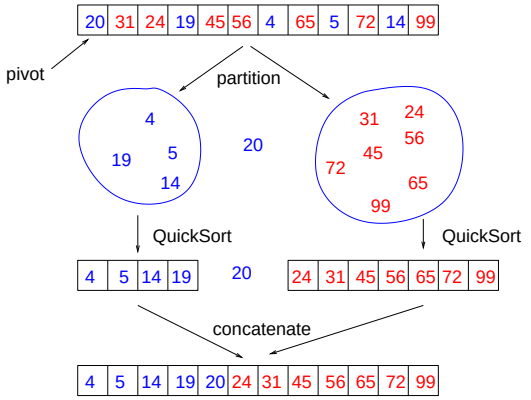
Main advantage of QuickSort: sorting can be done *in-place* (no extra array needs to be created).

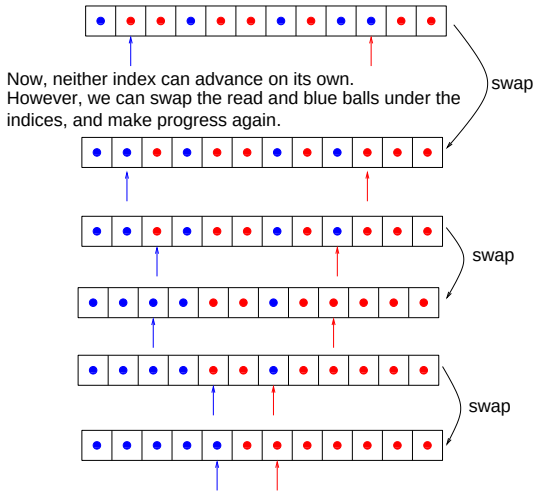
Key problem to be solved: how do we partition array in place?

If we can do this, we can have a Quicksort method of form

`void Quicksort(int[] A, int l, int h)` //sort values between l and h

where `l` is lower bound of subarray to be sorted, and `h` is upperbound.



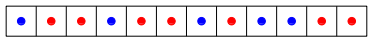


Notes on QuickSort code

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

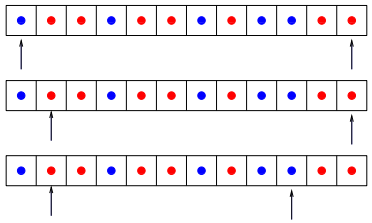
Checks for *array out of bounds* can be done more efficiently with *sentinels*.

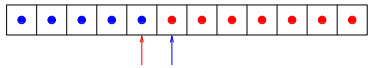
Inplace Partitioning



Can we get all blue balls to the left of all red balls?

Solution: keep two indices LEFT and RIGHT.
Invariant: all balls to left to LEFT index are certainly blue.
all balls to right of RIGHT index are certainly red.
Keep advancing the two indices till they cross.





Once indices cross over, partitioning is done.

What is analog of red/blue for quicksort?

All numbers less than pivot are blue.

All numbers greater than or equal to pivot are red.

Using this algorithm, we can partition in place!

Remaining question: what number do we choose for pivot?

Ideally, we would choose median value since that partitions array into two pieces of equal size.
However, median is expensive to compute exactly.

Heuristic: use first array value as pivot.

some people use middle element as pivot.

others use median of first, middle and last element

Asymptotic Complexity: leading term analysis

We can see that time required to search/sort grows with size of input. How do space/time needs of program grow with input size?

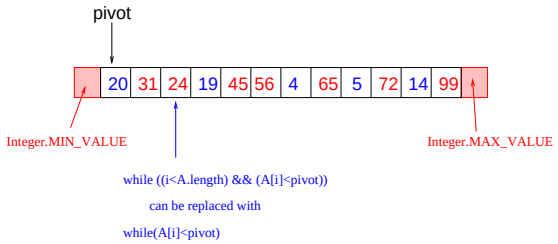
Let us focus on execution time. Space analysis is similar.

Example: selection sort

Execution time: count number of operations

- Basic operation: arithmetic/logical operation counts as 1 operation
- Assignment: counts as 1 operation (operation count of righthand side expression is determined separately)
- Loop: number of operations/iteration * number of loop iterations
- Method invocation: number of operations executed in invoked method

Sentinels



- Pad initial array with Integer.MIN_VALUE (-2147483648) and Integer.MAX_VALUE (2147483647)
- When advancing LEFT and RIGHT indices, Partition does not have to check that indices "fall off" the ends of the array.
- Recursive calls to Quicksort: sentinels are automatically there! (See picture below)

Asymptotic complexity:

- express required number of operations as a function of input size
- drop all terms except leading term, and ignore constant multiplier

Example: f(x) = 13*i + 8

f(x) = O(i)

Subtlety: operation count might depend not only on size of input but also on the value of the input (look at linear search or binary search). For big-O determination, use worst-case scenario.

Formal definition of O() notation:

Let $f(n)$ and $g(n)$ be functions. We say that $f(n)$ is *of order* $g(n)$, written $O(g(n))$ if there is a constant $c > 0$ such that for all but a finite number of positive values of n,

$f(n) \leq c * g(n)$

In other words, $g(n)$ sooner or later overtakes $f(n)$ as n gets large.

Example: $f(n) = n + 5, g(n) = n$. We show that $f(n) = O(g(n))$.

Choose $c = 6$:

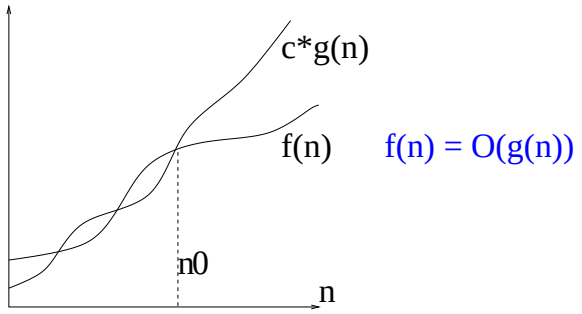
$f(n) = n + 5 \leq 6 * n$ for all $n > 0$.

Example: $f(n) = 17n, g(n) = 3n^2$. We show that $f(n) = O(g(n))$.

Choose $c = 6$:

$f(n) = 17n \leq 6 * 3n^2$ for all $n > 0$.

A graphical view of big-O notation



f(n) = O(g(n))

To prove that $f(n) = O(g(n))$, find an $n0$ and c such that $f(n) \leq c * g(n)$ for all $n > n0$.

Asymptotic complexity gives an idea of how rapidly space/time requirements grow as problem size grows.

Suppose we have a computing device that can execute 1000 operations per second. Here is the size of the problem that can be solved in a second, a minute and an hour by algorithms of different asymptotic complexity.

Complexity	1 second	1 minute	1hour
n	1000	60,000	3,600,000
n log n	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

For searching and sorting algorithms, you can usually determine big-O complexity by counting comparisons.

Reason: you usually end up doing some fixed number of arithmetic/logical operations per comparison.

Example: selection sort

```
public static void selectionSort(Comparable[] a) { <-- array of size n
  for (int i = 0; i < a.length; i++) {     <-- n iterations
    int MinPos = i;
    for (int j = i+1; j < a.length; j++) { <-- n-i-1 iterations
     if (a[j].compareTo(a[MinPos]) < 0)   <-- comparison
MinPos = j;}
    Comparable temp = a[i];
    a[i] = a[MinPos];
    a[MinPos] = temp;}}
```

Total number of comparisons = (n-1) + (n-2) + ... + 1 = n(n-1)/2

Complexity: $O(n^2)$

Analysis of binary search is a little more difficult.

```
public static boolean binarySearch(Comparable[] a, Object v) {
  ......
  while (left <= right){ <-- how many times does this loop execute?
      middle = (left + right)/2;
      int test = a[middle].compareTo(v); <-- comparison
      if (test < 0) left = middle+1;
      else
      if (test == 0) {
         return true;
        }
      else right = middle-1;
  }
  //if we reach here, we didn't find the object
  return false;
}
```


Example: linear search

```
public static boolean linearSearch(Comparable[] a, Object v) {
   int i = 0;
   while (i < a.length) <-- How many times does this loop execute???
     if (a[i].compareTo(v) == 0) return true; <-- comparison
     else i++;
   return false;
```

Number of times while loop executes depends not only on size of input array a but on the values in it and the value of v.

Big-O complexity: worst-case scenario

For linear search, worst-case scenario is that all array elements are examined.

So complexity of linear search = O(n) where n is size of input array.

Number of iterations of while loop depends on values in array and value of v.

OK, let's make worst case estimates: if array is of size n, what is the worst case number of iterations you make?

Easy to see that if size of array is n, first iteration cuts the size of the interval you need to look at to at most ceiling((n-1)/2). So if c(n) is worst-case number of comparisons,

```
c(1) = 1  c(2) = 2
c(n) = 1 + c(ceiling((n-1)/2))
```

It can be shown that c(n) = $O(log_2(n))$

Note: any time you have a procedure whose complexity is less than $O(n)$, it means intuitively that procedure does not examine all of the input.

Analysis of merge-sort:

```
public static Comparable[] mergeSort(Comparable[] A, int low, int high) {
  if (low < high - 1) //at least three elements
    {int mid = (low + high)/2;
     Comparable[] A1 = mergeSort(A, low, mid); <-- comparisons in method
     Comparable[] A2 = mergeSort(A, mid +1, high); <-- comparisons in meth
     return merge(A1,A2);} <-- comparisons in method
  ....
```

```
c(1) = 0 c(2) = 1
c(n) = 2c(n/2) + n
```

It can be shown that c(n) = $O(nlog_2(n))$.

Why is this wrong?

Analysis of quicksort: tricky

```
public static void quickSort(Comparable[] A, int l, int h) {
   if (l < h)
       {int p = partition(A,l+1,h,A[l]); <--- comparisons
            //move pivot into its final resting place
            //swap A[p-1] and A[l]
            Comparable temp = A[p-1];
    A[p-1] = A[l];
    A[l] = temp;
              quickSort(A,l,p-1); <-- comparisons
              quickSort(A,p,h);}} <-- comparisons
```

Incorrect attempt:

```
c(1) = 1 c(2) = 1
c(n) = (n-1)   + 2c(n/2)
       -----     ------
    partition   sorting the two partitioned arrays
```


Remember: big-O is worst-case complexity.

Worst-case for quicksort: one of the partitioned array is empty, and the other has (n-1) elements!

So actual recurrence relation is

```
c(1) = 1 c(2) = 1
c(n) = (n-1)   + c(n-1)
       -----     ------
    partition   sorting the two partitioned arrays
```

It can be shown that c(n) = $O(n^2)$

On the average (not worst-case), quick-sort runs in $n * log_2(n)$ time, which is why it is usually preferred in practice.

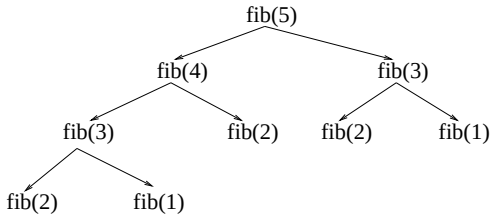
One approach to avoiding worst-case behavior: pick pivot carefully so it partitions array in half. Many heuristics for doing this, but none of them can guarantee that worst-case behavior will not show up.

Programs for the same problem can vary enormously in asymptotic efficiency.

```
fib(n) = fib(n-1) + fib(n-2)
fib(1) = 1
fib(2) = 1
```

Here is a recursive program:

```
static int fib(int n) {
  if (n <= 2) return 1;
  else return fib(n-1) + fib(n-2);
}
```



c(n) = c(n-1) + c(n-2) + 2

c(2) = 1 c(1) = 1

It can be shown that $c(n) = O(2^n)$. Cost of computing value is exponential in the size of the input!

Iterative Fibonnacci Code

fib (n) = fib(n-1) + fib(n-2) | n > 2

```
dad = 1
granddad = 1
current = 1;
for (i = 3; i <= n; i++) {
   granddad = dad;
   dad   = current;
   current = dad +granddad;
  }
printf("answer is " + current);
```

Number of times loop is executed is bounded by n.
Each iteration does some constant amount of work.

=> Time complexity of algorithm = O(n).

Generic Algorithms

Our search and sort algorithms were restricted to 1-D arrays.

However, some of these algorithms can actually work on more general data structures.

Example: linear search

Linear search can work on more general data structures such as a 2-D array: scan all the items in the 2-D array and report success if any of them is equal to the key.

Question: how can we write linear search so that it works for other data structures such as a 2-D array?

Answer: use cursors and iterators

Let us define a new interface:

```
public interface myIterator {
 void resetCursor();
 boolean hasNext(); //more elements to be enumerated
 Object next(); //get next element
 void remove(); //remove
}
```

Intuition:

- keep an internal "finger" (cursor) pointing into data collection
- convenient to think of cursor as being between successive elements
- every time a button is pressed, element just after cursor is returned and cursor is advanced

Let us examine where in linear search we use the fact that the input is an array.

```
  //linear search on possibly unsorted array
 public static boolean linearSearch(Comparable[] a, Object v) {
   int i = 0;
   while (i < a.length) //
              --------  <---- used fact that a is an array
     if (a[i].compareTo(v) == 0) return true;
         ----  <--- used fact that a is an array
     else i++;
   return false;}
```

Observation: we want

- array a to produce its elements on demand
- tell us when there are no more elements left

Look at class *manyBoos* to see an example of a class that implements a cursor.

New linear search code: see how it handles arbitrary class that implement myIterator.

Class that implements myIterator interface presents a stream abstraction of data collection to client class.

Implementing this abstraction may require multiple internal cursors. See class manyBooBoos for an example.

Problem with cursor as we have defined it: only one enumeration of elements can be going on at a time, since each enumeration requires its own cursor

Why would we ever need simultaneous enumerations? look at SelectionSort.

Solution: permit creation of cursors dynamically as needed

Intuition: we create buttons dynamically and attach them to data collection. Each button has its own cursor.

This means that class implementing Iterator must be distinct from data collection class.

Class implementing Iterator needs access to data collection, so use inner class.

See class manyBoosIter.

Summary

1. Asymptotic complexity: measure of space/time required by algorithm
2. Searching array: linear search $O(n)$, binary search $O(log(n))$
3. Sorting array: selection sort $O(n^2)$, merge sort $O(nlog(n))$, quick sort (in-place) $O(n^2)$
4. Writing generic code for linear search:
   - started with arrays of int
   - Comparable interface to permit handling of more general arrays
   - Iterator interface to permit handling of more general data collections

Generic code for binary search and sorting routines

Need more general kinds of enumerations that begin and end at arbitrary points in the data structure, and that can modify the data structure during the enumeration.

See later.

Many applications however require more dynamic data structures in which elements are added and removed in between searches.

Let us study such applications and data structures.