

Lecture 21 Searching

Searching

Given: an array of double numbers

Problem: is the number 42 in the array? *where?*

42 is called the KEY

Solution: we must search the array

check is `array[0] == 42?` `array[1] == 42?` ...

return position of found item, return -1 if not found

Linear Search 1 (naive version)

```
int position = -1;
for( int i=0; i < array.length; i++ ) {
    if( array[i] == 42 )
        position = i;
}
return i;
```

- how many steps?
- always checks entire array
- always takes n steps

Linear Search 2 (better search)

```
for( int i=0; i < array.length; i++ ) {
    if( array[i] == 42 )
        return i;
}
return -1;
```

- how many steps?
- I: yes, it is the 1st element... took 1 step
- II: yes, it is the 2nd element... took 2 steps .. etc.
- III: yes, it is the last element... took n steps
- IV: no, it is not in the array... took n steps

- number of steps: worst case
- if found, takes n steps — no better than naive algorithm!
- if not found, always takes n steps
- number of steps: average case
- loosely, takes $n/2$ steps (need to check half the values)
- more precisely,
 - assume it will be found, equally likely to search for any value in the array
 - probability it is in the first position, $\text{Pr}(\text{position } 0) = 1/n$
 - $\text{Pr}(\text{position } 1)$ is $1/n$, $\text{Pr}(\text{position } 2) = 1/n$, etc.
 - 1st position, takes 1 step at $\text{Pr}(0)$, takes $1 * 1/n = 1/n$ steps
 - 2nd position, takes 2 steps at $\text{Pr}(1)$, takes $2 * 1/n = 2/n$ steps
 - total expected: $1/n + 2/n + 3/n \dots + n/n$
 - $= n*(n+1)/(2n)$
 - $= (n+1) / 2$
 - $= n/2$ (roughly, on average)

How can we improve our searching algorithm?

Linear Search 3 (even better)

- suppose the array has already been sorted.....

```
for( int i=0; i < array.length; i++ ) {
    if( array[i] == 42 )
        return i;
    if( array[i] > 42 )
        return -1;
}
return -1;
```

- since array is sorted, we can stop when we go too far.

What does this improve?

- “if not found” case, don’t have to do all n steps
- on average, improves from n to $n/2$ steps
- not a big improvement

Binary Search (best search)

- suppose the array has already been sorted:
 - $a[0] \leq a[1] \leq a[2] \dots \leq a[a.length-1]$

```
binarySearch( array, key, lower, upper )
• use 2 variables to track where we are searching in array:
• lower — lower bound, an array index
• upper — upper bound, an array index
• look for key in array, from  $a[lower]$  to  $a[upper]$ 
```

As search runs, **the following statements should be true:**

- $a[i] \leq a[i+1]$ (presorted assumption)
- lower \leq upper (look in a valid range)
- $a[lower] \leq a[upper]$
- if *key* is in the array, then
- it will be found in the range $a[lower]..a[upper]$

Initially lower = 0, upper = a.length-1

- we will split the array in half,
 - mid = $(lower+upper)/2$
 - is the key in the first half? second half? at dividing line?
 - if $a[mid] < k$, it is between $a[mid+1]..a[upper]$
 - if $a[mid] > k$, it is between $a[lower]..a[mid-1]$
- ```
int binarySearch(double[] a, double key,
 int lower, int upper) {
 if(lower > upper) return -1; // not found!
 int mid = (lower+upper) / 2; // middle
 if(a[mid] == key)
 return mid;
 else if(a[mid] < key)
 return binarySearch(array, mid+1, upper);
 else // a[mid] > key
 return binarySearch(array, lower, mid-1);
}
```

Note — we don't have to use recursion

```
int binarySearch(double[] a, double key)
{
 int lower=0;
 int u = a.length-1;
 while(lower <= upper) {
 int mid = (lower+upper)/2;
 if(a[mid] == key)
 return mid; // found it!
 else if(a[mid] < key)
 lower = mid+1;
 else
 upper = mid-1;
 }
 return -1; // not found
}
```

How many steps are involved in a binary search?

- assume  $a.length = N = 2^k$
- eg,  $N = 2^10 = 1024$ 
  - step 1, cut array in half to  $N/2 = 512$
  - step 2, cut remainder in half, to  $N/2/2 = N/4 = 256$
  - step 3, cut in half, to  $N/8 = 128$

- ...
- step  $k$ , cut in half, to  $N/(2^k) = 1$

The remaining array search window is 1 ... we are done!

- A total of  $k$  steps:
  - $\log_2(N) = \log_2(2^k) = k$
  - hence,  $k = \log_2(N)$

## Lecture 22 Selection Sort, Complexity

### Sorting

- suppose you have an array of integers, want them sorted

**Given:** an array of integers, e.g.: { 6, 7, 2, 8, 3 }

**Problem:** rearrange the elements so they are in ascending order, i.e.: { 2, 3, 6, 7, 8 }

There are many different ways to sort,

- textbook covers
  - Selection Sort
  - Merge Sort
  - we will do
  - Selection Sort, not very efficient
  - Quicksort (NOT IN BOOK!) next day, very efficient
- you will probably remember the easiest ways to sort
  - eg: Bubble Sort or Selection Sort
- some algorithms are much more efficient than others
  - you should understand *why*
- almost all sorting algorithms involve swapping two values in an array at positions  $i$  and  $j$ , so we'll use the following method:

```
swap(double[] a, int i, int j)
{
 double old_ai = a[i];
 a[i] = a[j];
 a[j] = old_ai;
}
```

### Selection Sort

- scan the whole array  $0..length-1$
- place the smallest in position 0
- scan the rest of the array  $1..length-1$
- place the (second) smallest in position 1
- ... scan the rest of the array  $i..length-1$ , place  $i^{\text{th}}$  smallest in position  $i$
- loop invariant

*all elements at positions  $< i$  are sorted and placed in their correct order*

Example: { 6, 7, 2, 8, 3 }

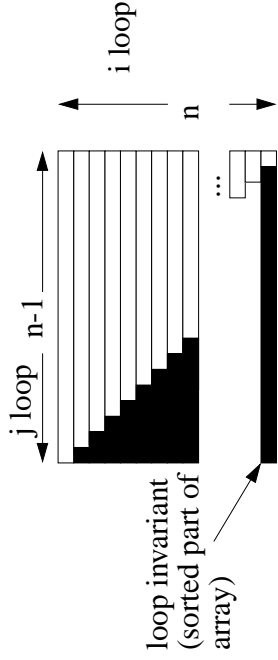
- { 6, 7, 2, 8, 3 } ( $i=0$ )
- { 2, 7, 6, 8, 3 } ( $i=1$ ) (check loop invariant)
- { 2, 3, 6, 8, 7 } ( $i=2$ )
- { 2, 3, 6, 8, 7 } ( $i=3$ )
- { 2, 3, 6, 7, 8 } ( $i=4$ )

```
selectionSort(int[] a)
{
 for(int i=0; i < a.length; i++) {
 // check loop invariant (always true)
 // find the position of the smallest
 // item between i and the end of the array
 int smallPos = i;
 for(int j=i+1; j < a.length; j++) {
 if(a[j] < a[smallPos])
 smallPos = j;
 }
 // move the small item to position i
 swap(a, i, smallPos);
 }
}
```

## How many Steps in Selection Sort?

- inspect the loop structure
 

```
for(int i=0; i < a.length; i++) {
 for(int j=i+1; j < a.length; j++) {
 if(...)
 ...
 }
}
```
- let  $n = a.length$
- outer “i” loop
  - $i=0$ , do inner “j” loop body  $n-1$  times
  - $i=1$ , do inner “j” loop body  $n-2$  times
  - ...
  - $i=n-1$ , do inner “j” loop body 0 times



- total steps
  - $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 + 0$
  - $(n-1)n / 2$  steps
  - or roughly
  - $(n^2)/2$  steps

## Bubble Sort (optional)

- pass through the whole array, in order (first to last)
- if adjacent values are out of order, swap them
- repeat as often as necessary
  - until the array is sorted

### Example:

```
double[] array = { 6, 7, 2, 8, 3 };
```

### Bubblesort 1

```
boolean isSorted;
do {
 isSorted = true; // assume true
 // make a pass through the whole array
 for(int i=0; i<array.length-1; i++) {
 // if out of order, swap the pair
 if(array[i] > array[i+1]) {
 isSorted = false; // proven false
 swap(a, i, i+1);
 }
 }
} while(!isSorted);
```

- after the first pass
  - the last value is the largest
  - after the second pass
    - the second last value is the 2nd largest
  - ...etc...
  - we can reduce the work we do in the inner loop
  - how many times do we execute the outer loop?

## Bubblesort 2

- reduces amount of work done in inner loop
- fixed amount of work to do in outer loop
- always assumes worst case (smallest at end of array)

```
for(int pass=0; pass<array.length; pass++) {
 // make a pass through the part of the
 // array that isn't sorted yet
 for(int i=0; i<array.length-pass-1; i++) {
 if(array[i] > array[i+1]) {
 swap(array, i, i+1);
 }
 }
}
```

## Comparing Bubblesorts

- bubblesort 1 is very efficient when data is already sorted
- only requires a single pass
- bubblesort 1 worst case is roughly  $(n-1)*n$  steps
- bubblesort 2 does less work 'in the middle loop'
- always does full work in outer loop
  - it is ineffective if data is already sorted
  - always requires roughly  $(n-1)*n/2$  steps
- exercise:
  - combine these to make bubblesort 3 to be more efficient

## Introduction to Time Complexity

Often want to know, 'How fast will an algorithm run?'

- depends on:
  - language, computer type, other (timesharing computer?)
  - machine language, C, Java
  - a PC, Sun, SGI, supercomputer
- sometimes hard to compare "efficiency" of algorithms
- don't need exact answers!
- one common question
  - is algorithm A faster than algorithm B?
  - but even this can be hard to quantify!
  - depends on the nature of the input
  - eg: searching:
    - is it the first item in the array? (linear search)
    - is it the last item in the array? (binary search)
    - depends on the size of the input
    - how big is the array?  $10^7$  ? 1,000,000 ?

Computer science studies the questions like this...

- *In the limit, as the problem size  $n$  grows,* how much does algorithm A slow down? algorithm B?
- As  $n$  grows, does A slow down *less than B*?

**Time complexity** describes

- how much work an algorithm does as problem size  $n$  changes
- what do we mean by: how much work?
  - roughly "**how many steps**" it executed...
  - in the best case? average case? worst case?
- usually concerned with the **worst-case** time complexity
- often compare two algorithms based on time complexity

## Big-Oh Notation

Describes how an algorithm *scales* with the problem size  $n$ :

- algorithm is  $O(1)$  ... “of order 1”
- fast, takes a constant number of steps
- does not depend on the problem size (eg: array size)
- example.... what is value at 4th position in array?
- algorithm is  $O(\log n)$  ... “of order  $\log n$ ”
- takes roughly  $\log(n)$  steps
- example: **binary search, fast power**
- if problem size doubles, takes roughly “one more step”
- algorithm is  $O(n)$  ... “of order  $n$ ”
- takes roughly  $n$  steps
- example: **linear search (all versions)**
- if problem size doubles, time taken roughly doubles
- ...  $O(n^2)$  ... “of order  $n$  squared”
- algorithm takes  $n^2$  steps, quadratic runtime
- example: **selection sort, bubble sort**
- if problem size doubles, takes quadruple time!
- ...  $O(n^k)$  ... “polynomial-time”
- algorithm takes  $n^k$  steps, eg: **matrix multiply** is  $O(n^3)$
- $k$  is some constant, *does not depend on problem size*
- many difficult problems are  $O(n^k)$
- ...  $O(x^n)$  ... “exponential time”
- algorithm takes  $x^n$  steps (some constant  $x$ )
- exponential runtime! slow! **Hanoi, recursive Fibonacci**
- if problem size doubles,
  - $x^{2n} = x^n * x^n$  ....
  - ..... takes  $x^n$  times longer than before!!!
- want to avoid exponential algorithms
- usually very difficult problems, to be avoided!

## Lecture 23 Quicksort

### QuickSort

last day we covered selection sort

- basic idea:
  - scan the array in order
  - find smallest value and place at beginning
  - repeat scan on array, skipping the 1st element

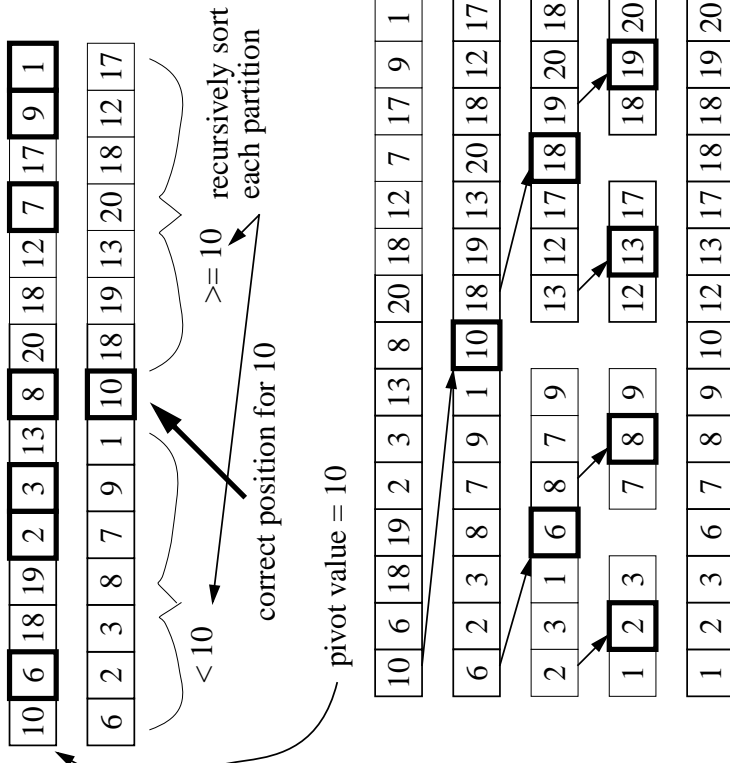
quicksort (leave below on board!!!)

- basic idea:
  - pick one element of the array, the “pivot”
  - partition the rest of the array into *two groups*
    - “little ones” that are  $<$  the pivot value
    - “big ones” that are  $\geq$  pivot value
  - place pivot in correct position (between two groups)
  - recursively sort each group
  - when this is finished, the array is in order
- **quicksort is fast because**
  - once an element is known  $<$  pivot value,
    - we never compare it to any of the big ones
  - similarly, big ones are never compared to little ones
- in comparison, selection sort compares each element directly to all others
- how fast is it?
  - typical case, it is  $O(n \log n)$ , meaning  $n * \log_2(n)$  steps
  - worst case, it is still  $O(n^2)$
  - there are many variations (quicksort, etc...)
    - some are a bit more efficient than others
    - all variations take  $n \log n$  steps

### Example

pick a *pivot*, say first element of array

- partition into 2 groups:  $<$  *pivot* and  $\geq$  *pivot*
- recurse on each group (base case — zero or one elements!)



- we can leave the pivot out of these 2 groups
- it is already placed in the correct position
- this example shows one way to do partitioning
- this way is easy to understand, but program code is ugly
- many ways exist, we'll show code for more elegant way

### Quicksort Pseudocode

```

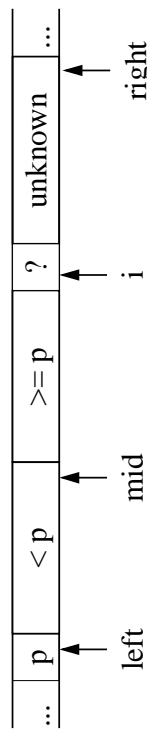
quicksort(double[] a, int left, int right) {
// +-----+-----+
// | | | <- array
// +-----+-----+
// 0 left right ^length-1
// sort a[left]..a[right] using quicksort
// left - lower bound
// right - upper bound
if(left >= right)
return; // base case
else
partition array around a pivot value p;
place p in its correct position @ 'mid';
// ASSERTION 1:
// elements from left..mid-1 are < p
// elements from mid+1..right are >= p
quicksort(a, left, mid-1);
quicksort(a, mid+1, right);
// ASSERTION 2: between left..right all
// elements sorted and in final position
end if
}
main(double[] array) {
quicksort(array, 0, array.length-1);
}

```

- how to “partition the array” ???

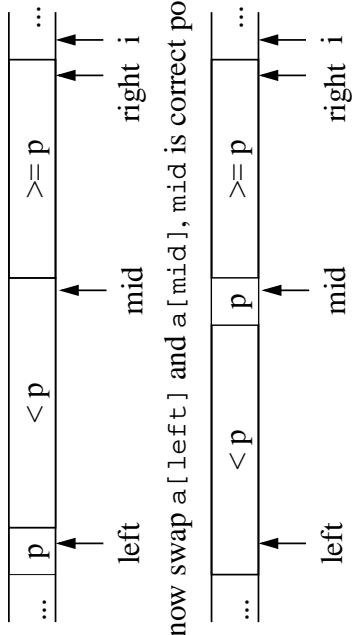
### Partitioning the Array (Partitioning Method 1)

- scan the array from a[left] to a[right]
- using variables mid and i
- maintaining the following relationships



- there are two cases:
  - if( a[i] >= p )
    - i++
  - else if( a[i] < p )
    - mid++
    - note that a[i] < a[mid]
    - swap a[i] and a[mid]
    - i++

- continue until i > right
- at end of scan:



- now swap a[left] and a[mid], mid is correct position for p
- notice how ASSERTION 1 is now correct
- recursively sort < p group, array[left .. mid-1]
- recursively sort >=p group, array[mid+1 .. right]

## QuickSort Code, Partitioning Method 1

```

class QuickSort {
public static void
qsort(double[] array) {
 qsort(array, 0, array.length-1);
}

private static void
qsort(double[] array, int left, int right)
{ // a helper method
 if(left < right) {
 int mid=partition(array, left, right);
 qsort(array, left, mid-1);
 qsort(array, mid+1, right);
 }
}

private static int
partition(double[] array, int left, int right) {
 final double pivot = array[left];
 int mid = left;
 for(int i=left+1; i <= right; i++) {
 if(array[i] < pivot) {
 mid++;
 swap(array, mid, i);
 }
 }
 swap(array, left, mid);
 return mid;
}

private static void
swap(double[] array, int i, int j) {
 double prev_ai = array[i];
 array[i] = array[j];
 array[j] = prev_ai;
}
}

```

## QuickSort Code, Partitioning Method 2

### A different way to partition....

- uses two markers, left and right
- elements to left of left marker are < p
- elements to right of right marker are >= p
- more elegant and efficient than previous method
- uses single assignment rather than swap()

```

private static int
partition(double[] array, int left, int right)
{
 final double pivot = a[left];
 while(left < right) {
 // move the right marker, find small #
 while(pivot<=a[right] && left<right) {
 right--;
 }
 // place small # on left
 a[left] = a[right];
 // move the left marker, find big #
 while(a[left]<pivot && left<right) {
 left++;
 }
 // place big # on right
 a[right] = a[left];
 }
 // place pivot in correct position
 a[left] = pivot; // note: left==right
 return left;
}

```

