Searching & Sorting
The Plan

- Searching
- Sorting
- Java Context
Search (Retrieval)

- “Looking it up”
- One of most fundamental operations
- Without computer
  - Indexes
  - Tables of content
  - Card Catalogue
  - Reference books
- Fundamental part of many computer algorithms
Linear (Sequential) Search

- Plod through material, one item at a time
- Always works
- Can be slow
- Sometimes the *only* way
- Phone Book Example
  - 660-6567
  - Whose number is it?
- (How could this be done faster?)
Binary Search

Often can do better than linear search:

- **Phone Book again (Predates Computer!)**
  - Find midpoint
  - Decide before or after (or direct hit)
  - Discard half of uncertainty
  - Repeat until there

- **Fast! (Don’t even need computer!)**
- **What does it require (why not use all the time)?**
- **How many extra steps if double sized book?**
Hashing

A way of storing info so we can go directly there to retrieve

- Mail boxes in a mail room (know exactly where number 33 is.)
- Hashing is a way of transforming some part of info to allow such straight-forward storage
- What to use for students in classroom
  - Age? Last name? SSN?
Hashing

- Use extra space to allow for faster operation
- Collision Handling
  - What to do if two different items map to the same bin?
  - Many different solutions...
Search Performance

- **Linear Search** (*brute force, plodding*)
  - Proportional to amount \( \sim N \)

- **Binary Search** (*telephone book*)
  - Proportional to log of amount \( \sim \log(N) \)

- **Hashing** (*go directly to …*)
  - Independent of amount! \( \sim \text{constant} \)
Sorting (Motivation)

Fundamental part of many algorithms and procedures

- **Required before other operations possible**
  - E.g., binary search

- **Often a user requirement for manual use**
  - E.g., phone book, dictionary, directory, index...

- **Get lower Postal Rates if sorted by Zip Code**

- **Implicit requirement for “orderly” operation**
Selection Sort

- N items in an array named Data
  [ 2 | 4 | 7 | 3 | 1 | 8 | 5 ]
- Find smallest of elements 0 thru N-1 of Data
- Interchange this with 1st element of array Data
  [ _ | _ | _ | _ | _ | _ | _ ]
- Find smallest of elements 1 thru N-1 of Data
- Interchange this with 2nd element of array Data
  [ _ | _ | _ | _ | _ | _ | _ ]
- ...
- Find smallest of elements k-1 thru N-1 of Data
- Interchange this with kth element of array Data
  [ _ | _ | _ | _ | _ | _ | _ ]
  [ _ | _ | _ | _ | _ | _ | _ ]
  [ _ | _ | _ | _ | _ | _ | _ ]
- Done when k-1 = N-1
  [ _ | _ | _ | _ | _ | _ | _ ]
Selection Sort

- N items in an array named Data
  
  [ 2 | 4 | 7 | 3 | 1 | 8 | 5 ]

- Find smallest of elements 0 thru N-1 of Data
  - Interchange this with 1st element of array Data
    
    [ 1 | 4 | 7 | 3 | 2 | 8 | 5 ]

- Find smallest of elements 1 thru N-1 of Data
  - Interchange this with 2nd element of array Data
    
    [ 1 | 2 | 7 | 3 | 4 | 8 | 5 ]

- ...

- Find smallest of elements k-1 thru N-1 of Data
  - Interchange this with kth element of array Data
    
    [ 1 | 2 | 3 | 7 | 4 | 8 | 5 ]
    [ 1 | 2 | 3 | 4 | 7 | 8 | 5 ]
    [ 1 | 2 | 3 | 4 | 5 | 8 | 7 ]

- Done when k-1 = N-1
  
  [ 1 | 2 | 3 | 4 | 5 | 7 | 8 ]
Selection Sort Performance ($N^2$)

- Assume there are $N$ items to be sorted
- Notice that with each pass we have to make $N$ comparisons
  - (actually $N/2$ on average)
- Notice that we have to make $N$ passes
  - (actually $N-1$)
- Therefore requires $N \times N$ comparisons
  - (actually $N \times (N-1)/2$)
- Performance proportional to $N^2$ or $\sim N^2$
Other Simple Sorts ($N^2$)

- 2 More simple sorts like Selection Sort
  - Insertion Sort
  - Bubble Sort

- All 3 have common properties
  - *Easy* to write
  - Fairly *slow* for large amounts of data
Industrial Quality Sorts

- Can do much better than simple sorts
- *Selection Sort* is often used
  - Divide and conquer strategy
  - Partitions data into two parts
  - Partitions each of these parts into subparts
  - Etc.
- Performance greatly improved over previous
  - Can handle any *real* job
Other Fast Sorts

- **Merge Sort**
  - Stable
  - Requires extra memory

- **Binary Tree Sort**

- **Heap Sort**

- **Shell Sort**

- **Bucket Sort**
  - Can be extremely fast under special circumstances
  - (Analogy to Hashing)
Sort Performance

- **Slowest: \( \sim N^2 \)**
  - Selection Sort
  - Insertion Sort, Bubble Sort

- **Very Fast: \( \sim N \log N \)**
  - QuickSort, Binary Tree Sort
  - Merge Sort, Heap Sort

- **Quite Fast**
  - Shell Sort

- **Fastest (limited situations): \( \sim N \)**
  - Bucket Sort
Java Context (writing your own?)

Don’t need to write your own -- Java includes:

- **For Collections**
  
  static void sort(List list)
  
  stable
  
  static int binarySearch(List list, Object key)

- **For Arrays** (?? = int, double, ..., and Object)
  
  static void sort(?? [] a)
  
  Uses quicksort (not stable)
  
  static int binarySearch( ?? [] a, ?? key)
Practice

1. In a class you design, create an array of ints, initialise with some numeric data and print it out.

2. Utilize the sort method found in the Arrays class. Sort your array and print it out again.

3. Write your own version of selection sort and add it to your class. Compare to the sort of the Arrays class.