Sorting: From Theory to Practice

- Why do we study sorting?
  - Because we have to
  - Because sorting is beautiful
  - Example of algorithm analysis in a simple, useful setting

- There are \( n \) sorting algorithms, how many should we study?
  - \( O(n) \), \( O(\log n) \), ...
  - Why do we study more than one algorithm?
    - Some are good, some are bad, some are very, very sad
    - Paradigms of trade-offs and algorithmic design
  - Which sorting algorithm is best?
  - Which sort should you call from code you write?

Sorting out sorts

- Simple, \( O(n^2) \) sorts — for sorting \( n \) elements
  - Selection sort --- \( n^2 \) comparisons, \( n \) swaps, easy to code
  - Insertion sort --- \( n^2 \) comparisons, \( n^2 \) moves, stable, fast
  - Bubble sort --- \( n^2 \) everything, slow, slower, and ugly

- Divide and conquer faster sorts: \( O(n \log n) \) for \( n \) elements
  - Quick sort: fast in practice, \( O(n^2) \) worst case
  - Merge sort: good worst case, great for linked lists, uses extra storage for vectors/arrays

- Other sorts:
  - Heap sort, basically priority queue sorting
  - Radix sort: doesn’t compare keys, uses digits/characters
  - Shell sort: quasi-insertion, fast in practice, non-recursive

Selection sort: summary

- Simple to code \( n^2 \) sort: \( n^2 \) comparisons, \( n \) swaps

```java
void selectSort(String[] a) {
    int len = a.length;
    for(int k=0; k < len; k++){
        int mindex = getMinIndex(a,k,len);
        swap(a,k,mindex);
    }
}
```

- # comparisons: \[ \sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = O(n^2) \]
- Swaps?
- Invariant: Sorted, won’t move final position

Insertion Sort: summary

- Stable sort, \( O(n^2) \), good on nearly sorted vectors
  - Stable sorts maintain order of equal keys
  - Good for sorting on two criteria: name, then age

```java
void insertSort(String[] a){
    int k, loc; String elt;
    for(k=1; k < a.length; ++k) {
        elt = a[k];
        loc = k;
        // shift until spot for elt is found
        while (0 < loc && elt.compareTo(a[loc-1]) < 0) {
            a[loc] = a[loc-1];   // shift right
            loc=loc-1;
        }
        a[loc] = elt;
    }
}
```

- Sorted relative to each other
Bubble sort: summary of a dog

- For completeness you should know about this sort
  - Really, really slow (to run), really really fast (to code)
  - Can code to recognize already sorted vector (see insertion)
    - Not worth it for bubble sort, much slower than insertion

```java
void bubbleSort(String[] a)
{
    for(int j=a.length-1; j >= 0; j--) {
        for(int k=0; k < j; k++) {
            if (a[k].compareTo(a[k+1]) > 0)
                swap(a,k,k+1);
        }
    }
    Sorted, in final position
}
```

- “bubble” elements down the vector/array

Summary of simple sorts

- Selection sort has n swaps, good for “heavy” data
  - moving objects with lots of state, e.g., ...
  - In C or C++ this is an issue
  - In Java everything is a pointer/reference, so swapping is fast since it's pointer assignment

- Insertion sort is good on nearly sorted data, it's stable, it's fast
  - Also foundation for Shell sort, very fast non-recursive
  - More complicated to code, but relatively simple, and fast

- Bubble sort is a travesty! But it's fast to code if you know it!
  - Can be parallelized, but on one machine don’t go near it (see quotes at end of slides)

Quicksort: fast in practice

- Invented in 1962 by C.A.R. Hoare, didn’t understand recursion
  - Worst case is O(n²), but avoidable in nearly all cases
  - In 1997 IntroSort published (Musser, introspective sort)
    - Like quicksort in practice, but recognizes when it will be bad and changes to heapsort

```java
void quick(String[], int left, int right)
{
    if (left < right) {
        int pivot = partition(a,left,right);
        quick(a,left,pivot-1);
        quick(a,pivot+1, right);
    }
}
```

- Recurrence?

Partition code for quicksort

```java
int partition(String[] a, int left, int right) {
    String pivot = a[left];
    int k, pIndex = left;
    for(k=left+1; k <= right; k++) {
        if (a[k].compareTo(pivot) <= 0) {
            pIndex++;
            swap(a,k,pIndex);
        }
    }
    swap(a,left,pIndex);
}
```

- Easy to develop partition
- loop invariant:
  - statement true each time loop test is evaluated, used to verify correctness of loop
- Can swap into a[left] before loop
  - Nearly sorted data still ok
Tail recursion elimination

- If the last statement is a recursive call, recursion can be replaced with iteration
  - Call cannot be part of an expression
  - Some compilers do this automatically

```java
void foo(int n) {
    if (0 < n) {
        System.out.println(n);
        foo(n-1);
    }
}
```

- What if print and recursive call switched?
- What about recursive factorial? return n*factorial(n-1);

Merge sort: worst case O(n log n)

- Divide and conquer --- recursive sort
  - Divide list/vector into two halves
  - Sort each half
  - Merge sorted halves together
  - What is complexity of merging two sorted lists?
  - What is recurrence relation for merge sort as described?

```java
T(n) = T(n) = 2T(n/2) + O(n)
```

- What is advantage of array over linked-list for merge sort?
  - What about merging, advantage of linked list?
  - Array requires auxiliary storage (or very fancy coding)

Merge sort: lists or arrays or ...

- Mergesort for arrays

```java
void mergesort(String[] a, int left, int right) {
    if (left < right) {
        int mid = (right+left)/2;
        mergesort(a, left, mid);
        mergesort(a, mid+1, right);
        merge(a,left,mid,right);
    }
}
```

- What’s different when linked lists used?
  - Do differences affect complexity? Why?

- How does merge work?

Merge for LinkedList

```java
public static LinkedList<String> merge(LinkedList<String> a, LinkedList<String> b) {
    LinkedList<String> result = new LinkedList<String>();
    while (a.size() != 0 && b.size() != 0) {
        String as = a.getFirst();
        String bs = b.getFirst();
        if (as.compareTo(bs) <= 0) {
            result.add(a.remove());
        } else {
            result.add(b.remove());
        }
    }
    // what's missing here??
```
Merge for arrays

- Array code for merge isn’t pretty, but it’s not hard
  - Mergesort itself is elegant

```java
void merge(String[] a, int left, int middle, int right) {
    // pre: left <= middle <= right,
    //      a[left] <= ... <= a[middle],
    //      a[middle+1] <= ... <= a[right]
    // post: a[left] <= ... <= a[right]
}
```

- Need extra storage, can’t easily merge in place
  - Can alternate between arrays: one merged into, then swap

Summary of O(n log n) sorts

- Quicksort is relatively straight-forward to code, very fast
  - Worst case is very unlikely, but possible, therefore ...
  - But, if lots of elements are equal, performance will be bad
    - One million integers from range 0 to 10,000
    - How can we change partition to handle this?

- Merge sort is stable, it’s fast, good for linked lists, harder to code?
  - Worst case performance is O(n log n), compare quicksort
  - Extra storage for array/vector

- Heapsort, more complex to code, good worst case, not stable
  - Basically heap-based priority queue in a vector

Sorting in practice

- Rarely will you need to roll your own sort, but when you do ...
  - What are key issues?

- If you use a library sort, you need to understand the interface
  - In C++ we have STL
    - STL has sort, and stable_sort
  - In C generic sort is complex to use because arrays are ugly
  - In Java guarantees and worst-case are important
    - Why won’t quicksort be used?

- Comparators permit sorting criteria to change simply

Non-comparison-based sorts

- lower bound: \( \Omega(n \log n) \) for comparison based sorts (like searching lower bound)
- bucket sort/radix sort are not-comparison based, faster asymptotically and in practice

<table>
<thead>
<tr>
<th>10</th>
<th>12</th>
<th>23</th>
<th>34</th>
<th>44</th>
<th>55</th>
<th>65</th>
<th>76</th>
<th>87</th>
<th>98</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

- sort a vector of ints, all ints in the range 1..100, how?
  - (use extra storage)
- radix: examine each digit of numbers being sorted
  - One-pass per digit
  - Sort based on digit

<table>
<thead>
<tr>
<th>10</th>
<th>12</th>
<th>23</th>
<th>34</th>
<th>44</th>
<th>56</th>
<th>73</th>
<th>76</th>
<th>87</th>
<th>98</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
**Scoreboard**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>stable/in-place</th>
<th>worst</th>
<th>avg</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quick</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>radix</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A Rose by any other name…C or Java?**

- Why do we use Java in our courses (royal we?)
  - Object oriented
  - Large collection of libraries
  - Safe for advanced programming and beginners
  - Harder to shoot ourselves in the foot

- Why don't we use C++ (or C)?
  - Standard libraries weak or non-existant (comparatively)
  - Easy to make mistakes when beginning
  - No GUIs, complicated compilation model

**Why do we learn other languages?**

- Perl, Python, PHP, mySQL, C, C++, Java, Scheme, ML, …
  - Can we do something different in one language?
    - Depends on what different means.
    - In theory: no; in practice: yes
  - What languages do you know? All of them.
  - In what languages are you fluent? None of them

- In later courses why do we use C or C++?
  - Closer to the machine, we want to understand the machine at many levels, from the abstract to the ridiculous
    - Or at all levels of hardware and software
  - Some problems are better suited to one language
    - What about writing an operating system? Linux?

**C++ on two slides**

- Classes are similar to Java, compilation model is different
  - Classes have public and private sections/areas
  - Typically declaration in .h file and implementation in .cpp
    - Separate interface from actual implementation
    - Good in theory, hard to get right in practice
  - One .cpp file compiles to one .o file
    - To create an executable, we link .o files with libraries
      - Hopefully someone else takes care of the details

- We #include rather than import, this is a preprocessing step
  - Literally sucks in an entire header file, can take a while for standard libraries like iostream, string, etc.
  - No abbreviation similar to java.util.*;
C++ on a second slide

- We don't have to call new to create objects, they can be created "on the stack"
  - Using new creates memory "on the heap"
  - In C++ we need to do our own garbage collection, or avoid and run out of memory (is this an issue?)

- Vectors are similar to ArrayLists, pointers are similar to arrays
  - Unfortunately, C/C++ equate array with memory allocation
  - To access via a pointer, we don't use . we use ->

- Streams are used for IO, iterators are used to access begin/end of collection
  - ifstream, cout correspond to Readers and System.out

How do we read a file in C++ and Java?

Scanner s = new Scanner(new File("data.txt"));
TreeSet<String> set = new TreeSet<String>();
while (s.hasNext()){
    String str = s.next();
    set.add(str);
}
myWordsAsList = new ArrayList<String>(set);

string word;
set<string> unique;
ifstream input("data.txt");
while (input >> word){
    unique.insert(word);
}
myWordsAsVector = vector<string>(unique.begin(), unique.end());

- What are similarities? Differences?

How do we read a file in C?

FILE * file = fopen("/u/ola/data/poe.txt","r");
char buf[1024];
char ** words = (char **) malloc(5000*sizeof(char **));
int count = 0;
int k; 
while (fscanf(file,"%s",buf) != EOF){
    int found = 0;  // look for word just read 
    for(k=0; k < count; k++){
        if (strcmp(buf,words[k]) == 0){
            found = 1;
            break;
        }
    }
    if (!found){   // not found, add to list
        words[count] = (char *) malloc(strlen(buf)+1); 
        strcpy(words[count].buf); 
        count++;
    }
}

- What if more than 5000 words? What if string length > 1024? What if?
  - What is complexity of this code?

The exam

- Wednesday, April 29, 7pm-10pm in White Lecture Hall
- Open book/open note
- ~40% multiple choice/short answer
- Cumulative
- By Friday, April 24:
  - All grades up (except huff)
  - Test solutions out
  - Grade problems: Submit assignment issues
- Final grades up Saturday, May 2
- Available by appointment throughout reading period and exam week
- Help session Wednesday in class