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?

Selection sort: summary

- Simple to code \(n^2\) sort: \(n^2\) comparisons, \(n\) swaps
  
  ```java
  void selectSort(String[] a)
  {
    for(int k=0; k < a.length; k++)
    {
      int minIndex = findMin(a,k);
      swap(a,k,minIndex);
    }
  }
  ```

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

Insertion Sort: summary

- Stable sort, \(O(n^2)\), good on nearly sorted vectors
  
  ```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

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
Bubble sort: summary of a dog

- For completeness you should know about this sort
  - Few, if any, redeeming features. Really slow, really, really
  - 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] > a[k+1])
                swap(a,k,k+1);
        }
    }
}
```

“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);    }
}
```

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
Analysis of Quicksort

- Average case and worst case analysis
  - Recurrence for worst case: \( T(n) = T(n-1) + T(1) + O(n) \)
  - What about average? \( T(n) = 2T(n/2) + O(n) \)

- Reason informally:
  - Two calls vector size n/2
  - Four calls vector size n/4
  - ... How many calls? Work done on each call?

- Partition: typically find middle of left, middle, right, swap, go
  - Avoid bad performance on nearly sorted data
- In practice: remove some (all?) recursion, avoid lots of "clones"

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?

```java
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?
    \( T(n) = T(n-1) + T(1) + O(n) \)

- 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 vectors

- Mergesort for vectors

```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?
Mergesort continued

- 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]
```

- Why is this prototype potentially simpler for linked lists?
  - What will prototype be? What is complexity?

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?

- Mergesort 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 the 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
- 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