Sorting: From Theory to Practice

- **Why do we study sorting?**
  - Because we have to
  - Because sorting is beautiful
  - Because ... and ...

- **There are** \( n \) **sorting algorithms, how many should we study?**
  - \( O(n) \), \( O(\log n) \), ...
  - Why do we study more than one algorithm?
    - Because we can! And they're beautiful!
    - Paradigms of trade-offs and algorithm design
  - Which sorting algorithm is best?
  - Which sort should you call from code you write?

Sorting out sorts (see also sortall.cpp)

- **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, though \( O(n^2) \) 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 (see also sortall.cpp)

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

```c++
void selectSort(tvector<string>& a)
{
    int k;
    for(k=0; k < a.size(); k++)
    {
        int minIndex = findMin(a,k,a.size());
        swap(a[k],a[minIndex]);
    }
}
```

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

Insertion Sort

- **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

```c++
void insertSort(tvector<string>& a)
{
    for(int k=1; k < a.size(); k++)
    {
        string elt = a[k]; int loc = k;
        while (0 < loc && elt < a[loc-1])
        {
            a[loc] = a[loc-1];   // shift right
            loc=loc-1;
        }
        a[loc] = elt;
    }
}
```

- **Sorted relative to each other**
Bubble sort

- 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

```cpp
void bubbleSort(tvector<string>& a)
{
    int j, k;
    for(int j=a.size()-1; j >= 0; j--)
    {
        for(int k=0; k < j; k++)
            if (a[k] > a[k+1])
                swap(a[k],a[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., ...
    - A string isn't heavy, why? (pointer and pointee)
    - What happens in Java?
    - C++: wrap heavy items in "smart pointer proxy"

- 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
  - Can be parallelized, but on one machine don't go near it

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

```cpp
void quick(tvector<string>& a, 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

```cpp
int partition(tvector<string>& a, int left, int right)
{
    string pivot = a[left]; int k, pIndex = left;
    for(k=left+1, k <= right; k++)
        if (a[k] <= pivot)
            pIndex++; swap(a[k], a[pIndex]);
    swap(a[left], a[pIndex]);
}
```

Complexity?

- <= X
- X
- > X

Partition code for quicksort

```cpp
void quick(tvector<string>& a, int left, int right)
{
    if (left < right)
    {
        int pivot = partition(a, left, right);
        quick(a, left, pivot-1); quick(a, pivot+1, right);
    }
}
```

Loop invariant:

- statement true each time loop test is evaluated, used to verify correctness of loop
  - Can swap into [left] before loop
    - Nearly sorted data is a problem
Analysis of Quicksort

- Average case and worst case analysis, recurrences:
  - Recurrence for worst case: \( T(n) = T(n-1) + T(1) + \Omega(n) \)
  - What about average? \( T(n) = 2T(n/2) + \Omega(n) \)
- Reason informally, why is complexity \( \Omega(n \log n) \)?
  - Two calls vector size \( n/2 \)
  - Four calls vector size \( n/4 \)
  - ... How many calls? Work done on each call?
- Partition: typically find average of left, middle, right, swap, go
  - Try to 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

```
void foo(int n)                 void foo2(int n)
{                               {
  if (0 < n) {                    while (0 < n) {
    cout << n << endl;            cout << n << endl;
    foo(n-1);                      n = n-1;
  }                               }
}                               }
```

Merge sort: worst case \( \Omega(n \log n) \)

- Divide and conquer — recursive sort (see code on next page)
  - 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?
    \[ T(n) = T(n/2) + T(n/2) + \Omega(n) = 2T(n/2) + \Omega(n) \]
- What is advantage of vector over linked-list for merge sort?
  - What about merging, advantage of linked list?
  - Vector requires auxiliary storage (or very fancy coding)

Merge sort: lists or vectors

- Mergesort for vectors

```c
void mergesort(vector<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

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

```cpp
void merge(tvector<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?

- 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, see libsort.cpp

- 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 and sortall.cpp in Tapestry
    - STL has sort, and stable_sort
    - Tapestry has lots of sorts, Mergesort is fast in practice, stable, safe
  - In C the generic sort is complex to use because arrays are ugly
    - See libsort.cpp
  - In Java guarantees and worst-case are important
    - Why won't quicksort be used?

- Function objects permit sorting criteria to change simply

Standard sorts: know your library

- Know how to use the STL sorts even if you don’t use STL
  - The sort function takes iterators as parameters
  - vectors, strings and other containers: “give me iterators”
    - What about linked-list iterators? Why aren’t these “sortable”?

```
string s = "...";
sort(s.begin(), s.end());
vector<string> vs; // fill vs with values
sort(vs.begin(), vs.end());
```

- Beware C qsort, vary widely and wildly on different platforms
  - See qsort on Linux/cygwin compared to g++ on Solaris?
In practice: templated sort functions

- Function templates permit us to write once, use several times for several different types of vector
  - Template function "stamps out" real function
  - Maintenance is saved, code still large (why?)

- What properties must hold for vector elements?
  - Comparable using < operator
  - Elements can be assigned to each other

- Template functions capture property requirements in code
  - Part of generic programming
  - Some languages support this better than others (not Java)

Function object concept

- To encapsulate comparison (like operator <) in a parameter
  - Need convention for parameter: name and behavior
  - Enforceable by templates or by inheritance (or both)
    - Sorts don’t use inheritance, tpqueue<...> does

- Name convention: class/object has a method named compare
  - Two parameters, the (vector) elements being compared
  - See comparer.h, used in sortall.h and in tpq.h

- Behavior convention: compare returns an int
  - zero if elements equal
  - +1 (positive) if first > second
  - -1 (negative) if first < second

Function object example

```cpp
class StrLenComp // : public Comparer<string> {
    public:
        int compare(const string& a, const string& b) const
        // post: return -1/+1/0 as a.length() < b.length()
        {
            if (a.length() < b.length()) return -1;
            if (a.length() > b.length()) return 1;
            return 0;
        }
    // to use this:
    StrLenComp scomp;
    if (scomp.compare("hello", "goodbye") < 0) {
        // We can use this to sort, see sortall.h, libsort.cpp
        // Call of sort: InsertSort(vec, vec.size(), scomp);
    }
}

// to use this:
StrLenComp scomp;
if (scomp.compare("hello", "goodbye") < 0) {
    // We can use this to sort, see sortall.h, libsort.cpp
    // Call of sort: InsertSort(vec, vec.size(), scomp);
}
```

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

- 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
Shell sort: not fast, not slow, just right?

- Comparison-based, similar to insertion sort
  - Using Hibbard's increments (see sortall.h) yields $O(n^{3/2})$
  - Sequence of insertion sorts, note last value of h!!

```c
int k, loc, h; string elt;
int k, loc, h; string elt;
h = ...; // set h to 2^p-1, just less than a.size()
h = ...; // set h to 2^p-1, just less than a.size()
while (h > 0) {
  while (h > 0) {
    for (k = h; k < n; k++) {
      for (k = h; k < n; k++) {
        elt = a[k];
        elt = a[k];
        loc = k;
        loc = k;
        while (h <= loc && elt < a[loc-h]) {
          while (h <= loc && elt < a[loc-h]) {
            a[loc] = a[loc-h];
            a[loc] = a[loc-h];
            loc -= h;
            loc -= h;
          }
          }
        a[loc] = elt;
        a[loc] = elt;
      }
      h /= 2;
      h /= 2;
    }
    }
  }
}
```

Barbara Simons

- President of ACM: '98-'00
  - Founder and chair of branch that deals with public policy
- Awards from CPSR and EFF
  - Computer Professionals for Social Responsibility
  - Electronic Frontier Foundation

"... I urged the relaxation of export controls and the availability of better tools to protect online privacy and security. I have written and spoken of the need to increase the security of the Internet, both by encouraging the use of robust software and technology and by discouraging widespread surveillance and monitoring."