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

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

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

Insertion Sort: summary

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

  ```cpp
  void insertSort(tvector<string>& a)
  {
    int k, loc; string elt;
    for(k=1; k < a.size(); k++) {
      elt = a[k];
      loc = k;
      // shift until spot for elt is found
      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

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, $O(n^2)$ 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
Bubblesort: 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

```cpp
void bubbleSort(tvector<string>& a)
{
    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

QuickSort: fast in practice

- Invented in 1962 by C.A.R. Hoare, didn't understand recursion
  - Worst case is O(n^2), 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);
}
```

- Recurrence?

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?
    - 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 (see quotes at end of slides)

Partition code for quicksort

```
what we want
 left

<= pivot > pivot
 pIndex

right

what we have
 left

????????????

right

invariant
 left

<= > ???
 pIndex

right

k
```

- 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"

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?
  \( T(n) = T(n) = 2T(n/2) + O(n) \)

- 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)

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

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

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

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

Merge sort: lists or vectors

- Mergesort for vectors

```c
void mergesort(tvector<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

```c
void merge(vector<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”?

```c
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?
Using STL sorts with tvector
- Include the right header files
  - “tvector.h” for tapestry vectors
  - <algorithm> for STL sorts
- Use STL iterators to get beginning and end of vector
  ```cpp
tvector<string> vec;
  vec.push_back("hello");
  vec.push_back("alligator");
  sort(vec.begin(), vec.end());
```
- Can sort strings this way too, but not sets even though they
  have begin() and end() in STL, why?
  - Don’t support random access, not sortable

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, tqueue<...> 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

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 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
  Call of sort: InsertSort(vec, vec.size(), scomp);
```
Non-comparison-based sorts

- lower bound: \( O(n \log n) \) for comparison based sorts (like searching lower bound)
- bucket sort/radix sort are non-comparison based, faster asymptotically and in practice

<table>
<thead>
<tr>
<th></th>
<th>23</th>
<th>34</th>
<th>56</th>
<th>25</th>
<th>44</th>
<th>73</th>
<th>42</th>
<th>26</th>
<th>10</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>42</th>
<th>23</th>
<th>73</th>
<th>44</th>
<th>26</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

Shell sort

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