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?
➤ Which sorting algorithm is best?

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

Selection sort

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

```cpp
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:
\[
\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = O(n^2)
\]

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

```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

Sorted, won’t move final position
**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(j=a.size()-1; j >= 0; j--)
  { for(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?
  - 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 $O(n^2)$, 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);
 }
}
```

**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]);
}
```

**Recurrence?**

```
<table>
<thead>
<tr>
<th>&lt;= pivot</th>
<th>&gt; pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>right</td>
</tr>
</tbody>
</table>
```

- Easy to develop partition
- 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 still ok
Analysis of Quicksort

- Average case and worst case analysis
  - Recurrence for worst case: $T(n) =$
  - What about average?

- 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
  ```
  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?

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?
  - $T(n) =$

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

- Function objects permit sorting criteria to change simply

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

```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;
    }
};
```

```cpp
if (scomp.compare("hello", "goodbye") < 0) …
```

**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;
    }
};
```

```cpp
// to use this:
StrLenComp scomp;
if (scomp.compare("hello", "goodbye") < 0) …
```

**Non-comparison-based sorts**

- lower bound: Ω(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?
- radix: examine each digit of numbers being sorted

**Shell sort**

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

```cpp
int k, loc, h; string elt;
h = 2; // set h to 2^k-1, just less than a.size()
while (h > 0)
{
    for (k=h; 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;
}
```