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

- **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

```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: $\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = O(n^2)$
  - 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

```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 | ?????
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

```c
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
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

```c
void quick(vector<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?

<table>
<thead>
<tr>
<th>&lt;= X</th>
<th>X</th>
<th>&gt; X</th>
</tr>
</thead>
</table>

pivot index
Partition code for quicksort

what we want

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

what we have

<table>
<thead>
<tr>
<th>??????????????</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
</tr>
</tbody>
</table>

invariant

<table>
<thead>
<tr>
<th>&lt;=</th>
<th>&gt;</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>pIndex</td>
<td>k</td>
</tr>
</tbody>
</table>

● Easy to develop partition

```c
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]);
}
```

● 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: $T(n) =$
  ➤ 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

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

```cpp
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

```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?
Creating a function template

```cpp
template <class Type>
void selectSort(tvector<Type> & a)// pre: a contains a.size() elements
// post: elements of a are sorted in non-decreasing order
{
    int j,k,minIndex,numElts = a.size();
    Type temp;
    // invariant: a[0]..a[k-1] in final position
    for(k=0; k < numElts - 1; k++)
    {
        minIndex = k; // minimal element index
        for(j=k+1; j < numElts; j++)
        {
            if (a[j] < a[minIndex])
            {
                minIndex = j; // new min, store index
            }
        }
        temp = a[k]; // swap min and k-th elements
        a[k] = a[minIndex];
        a[minIndex] = temp;
    }
}
● When the user calls this code, different versions are compiled
```
Some template details

- 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
Templates and function objects

- In a templated sort function vector elements must have certain properties (as noted previously)
  - Comparable using `operator <`
  - Assignable using `operator =`
  - Ok for int, string, what about `Date`, `GrammarElement`?

- What if we want to sort by a different criteria
  - Sort strings by length instead of lexicographically
  - Sort students by age, grade, name, ...
  - Sort stocks by price, shares traded, profit, ...

- We can’t change how `operator <` works
  - Alternative: write sort function that does NOT use `<`
  - Alternative: encapsulate comparison in parameter, pass it
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, \texttt{tpqueue<...>} does

- Name convention: class/object has a method named \texttt{compare}
  - Two parameters, the (vector) elements being compared
  - See \texttt{comparer.h}, used in \texttt{sortall.h} and in \texttt{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
    ➤ Call of sort: InsertSort(vec, vec.size(), scomp);
```
Another function object example

- Consider “directory.h” and the class \texttt{DirEntry}
  - \texttt{DirEntry} encapsulates file/directory
  - Methods: \texttt{Name()}, \texttt{Size()}, \texttt{Path()}, \texttt{GetTime()}, ...

- To sort using \texttt{Name()} use class below, what about \texttt{Size()}?

```cpp
class DirNameComp // : public Comparer<DirEntry>
{
public:
    int compare(const DirEntry& a, const DirEntry& b) const
    // post: return -1/+1/0 as a.Name() < b.Name()
    {
        if (a.Name() < b.Name()) return -1;
        if (a.Name() > b.Name()) return 1;
        return 0;
    }
}
```
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?

- radix: examine each digit of numbers being sorted

\[
\begin{array}{cccccccccccc}
23 & 34 & 56 & 25 & 44 & 73 & 42 & 26 & 10 & 16 \\
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]
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=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;
}
```