Lecture 10
Elementary Sorting Methods

EECS 281: Data Structures & Algorithms

Sorting Overview

Data Structures & Algorithms

Computational Task & Solutions

Sort *records* in a sequence* by *keys*, with respect to an operator/functor

```
bool operator<(const T&, const T&) const
```

Elementary sorts

- Bubble Sort
- Selection Sort
- Insertion Sort
- Counting sort (`int`, few different keys)

High-Performance Sorts

- Quicksort
- Merge Sort
- Heapsort
- Radix Sort (int, char*, …)

***Sort keys in ascending order, unless directed otherwise**

Elementary ≠ Useless

Elementary algorithms for sorting…

- Illustrate how to approach a problem
- Illustrate how to compare multiple solutions
- Illustrate program optimization
- Are sometimes “good enough”
- Often combined with sophisticated sorts

Which Containers Can Be Sorted?

Array vs. Linked List Data Representation

- Both will be considered, begin with arrays
- Some basic tasks are better suited for arrays
- Some basic tasks can be adapted to linked lists

Accessing Containers

- STL sorting is done using *iterators*

```
#include <algorithm>
vector<int> a{4, 2, 5, 1, 3};
std::sort(begin(a), end(a));  // sorts [left, right)
```

- In this lecture, we use *indices* and arrays

```
int a[] = {4, 2, 5, 1, 3};
bubble(a, 0, 5);  // sorts [left, right)
```

- Exercise at home: rewrite using iterators

How Size Affects Sorting

<u>Internal Sort</u>: “I can see all elements”

- Sequence to be sorted fits into memory
- $O(1)$-time random access is available

<u>Indirect Sort</u>: “I can see all indices”

- Reorder indices of items rather than items
  (important when copying is expensive)

<u>External Sort</u>: “Too many elements”

- Items to be sorted are on disk
- Items are accessed sequentially or in blocks

Exercise

- Write `swap()`
  - Swap items A and B
- Return type?
- Write a templated version

Solution

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}  // swap()

template <typename TYPE>
void swap(TYPE &a, TYPE &b) {
  TYPE tmp = a;
  a = b;
  b = tmp;
}  // swap()
```

Don’t do this!
`#include <utility>`
Use the STL `swap()` template
Uses C++1z move semantics

Desirable Qualities of Algorithms

- Asymptotic complexity, number of comparisons (and/or swaps)
  - Worst-case
  - Average-case
- Memory efficiency considerations
  - Sort in place with $O(1)$ extra memory
  - Sort in place with recursion (consider stack)
  - Sort in place, but have pointers or indices (n items need an additional n ptrs or indices)
  - Some sorts need $\Omega(n)$ extra memory

Desirable Qualities of Algorithms

- Stability
  - Definition: preservation of relative order of items with duplicate keys
  - Elementary sorts tend to be stable (not all)
  - Complex sorts are often *not* stable
  - *Important for sorting on two or more keys (sort* alphabetically*, then by* SSN*)*

Sorting Algorithms: Stability

A sorting algorithm is <u>stable</u> if data with the same key remains in the <u>same relative locations</u>

Example Input**:** (already sorted by first name)
{“**Sheen**, Charlie”, “**Berry**, Halle”, “**Liu**, Lucy”, “**Sheen**, Martin”}

↓

$\boxed{\text{Sort by } \textbf{Last Name}}$

Example <u>Stable</u> Output**:** (two “Sheen” stay in relative locations)
{“Berry, Halle”, “Liu, Lucy”, “Sheen, Charlie”, “Sheen, Martin”}

Example <u>Unstable</u> Output**:**
{“Berry, Halle”, “Liu, Lucy”, “Sheen, Martin”, “Sheen, Charlie”}

Types of Algorithms

- Non-Adaptive Sort
  - Sequence of operations performed is independent of order of data
  - Usually simpler to implement
- Adaptive Sort
  - Performs different sequences of operations depending on outcomes of comparisons

Worst-case versus best-case complexity

Sorting Overview

Data Structures & Algorithms

Bubble Sort

Data Structures & Algorithms

Bubble Sort

```
void bubble(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i)
    for (size_t j = right - 1; j > i; --j)
      if (a[j] < a[j - 1])
        swap(a[j - 1], a[j]);
}  // bubble()
```

- Find minimum item on first pass by comparing adjacent items, and swap item all the way to left
- Find second smallest item on second pass
- Repeat until sorted
- Non-adaptive

Example

```
void bubble(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i)
    for (size_t j = right - 1; j > i; --j)
      if (a[j] < a[j - 1])
        swap(a[j - 1], a[j]);
}  // bubble()
```

```
input: {4, 2, 5, 1, 3}
pass1: {1, 4, 2, 5, 3}  // Smallest item first
pass2: {1, 2, 4, 3, 5}  // …second smallest
pass3: {1, 2, 3, 4, 5}  // …
pass4: {1, 2, 3, 4, 5}  // No change, why pass?
```

Bubble Sort: Pop Quiz

- Currently non-adaptive
- How can bubble sort be fine-tuned to minimize work? i.e., make it *adaptive.*
- Hint: think about what happens if we bubble sort {1, 2, 3, 4, 5}

Adaptive Bubble Sort

```
void bubble(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i) {
    bool swapped = false;
    for (size_t j = right - 1; j > i; --j) {
      if (a[j] < a[j - 1]) {
        swapped = true;
        swap(a[j - 1], a[j]);
      }  // if
    }  // for j
    if (!swapped)
      break;
  }  // for i
}  // bubble()
```

Bubble Sort Analysis

- Non-adaptive version
  - ~$n^2/2$ comparisons and swaps in worst and average cases
  - ~$n^2/2$ comparisons, 0 swaps in best case
- Adaptive version
  - ~$n^2/2$ comparisons and swaps in worst and average cases
  - ~n comparisons in best case
  - ~n swaps (as few as 0) in best case

Bubble Sort

- Advantages
  - Simple to implement and understand
  - Completes some ‘pre-sorting’ while searching for smallest key (adjacent pairs get swapped)
  - Stable
  - Adaptive version may finish quickly if the input array is *almost sorted*
- Disadvantages
  - $O(n^2)$ time
    - $n^2/2$ comparisons and $n^2/2$ swaps
  - Slower than high-performance sorts

Bubble Sort

Data Structures & Algorithms

Selection Sort

Data Structures & Algorithms

Selection Sort

```
void selection(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i) {
    size_t minIndex = i;
    for (size_t j = i + 1; j < right; ++j)
      if (a[j] < a[minIndex])
        minIndex = j;
    swap(a[i], a[minIndex]);
  }  // for
}  // selection()
```

- Find smallest element in array, swap with first position
- Find second smallest element in array, swap with second position
- Repeat until sorted
- Non-adaptive

Example

```
void selection(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i) {
    size_t minIndex = i;
    for (size_t j = i + 1; j < right; ++j)
      if (a[j] < a[minIndex])
        minIndex = j;
    swap(a[i], a[minIndex]);
  }  // for
}  // selection()
```

```
input: {4, 2, 5, 1, 3}
pass1: {1, 2, 5, 4, 3}
pass2: {1, 2, 5, 4, 3}
pass3: {1, 2, 3, 4, 5}
pass4: {1, 2, 3, 4, 5}
```

Adaptive Selection Sort

```
void selection(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i) {
    size_t minIndex = i;
    for (size_t j = i + 1; j < right; ++j)
      if (a[j] < a[minIndex])
        minIndex = j;
    if (minIndex != i)
      swap(a[i], a[minIndex]);
  }  // for
}  // selection()
```

- Extra comparison checks if item is already in position
- Eliminates unnecessary (costly) swaps

Selection Sort Analysis

- Non-adaptive version
  - ~$n^2/2$ comparisons
  - n - 1 swaps in best, average, worst case
  - Run time is *insensitive* to input
- Adaptive version
  - $(n^2 - n)/2 + (n - 1)$ comparisons
  - n - 1 swaps worst case
  - 0 swaps best case
  - Run time is *sensitive* to input

Selection Sort Advantages

- Minimal copying of items
  - Good choice when objects are large and copying is expensive
- Fairly efficient for small *n*

Selection Sort Disadvantages

- $\Theta(n^2)$ time
- Run time only slightly dependent upon pre-order
  - The smallest key on one pass tells nothing about the smallest key on subsequent passes
  - Runs about the same on
    - Already sorted array
    - Array with all keys equal
    - Randomly arranged array
- Sort is not stable

Selection Sort

Data Structures & Algorithms

Summary/Preview

- Sorting algorithms useful to
  - Satisfy direct requests to sort something
  - Do so quickly
- Two sorts (Bubble and Selection)
  - Asymptotically “slow” ($O(n^2)$)
  - Minimal opportunity for tuning
- Next: Insertion Sort
  - Also asymptotically “slow” ($O(n^2)$)
  - More opportunities for tuning

Insertion Sort

Data Structures & Algorithms

Insertion Sort

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = left + 1; i < right; ++i)
    for (size_t j = i; j > left; --j)
      if (a[j] < a[j - 1])
        swap(a[j - 1], a[j]);
}  // insertion()
```

- Divide items into two groups: sorted and unsorted
- Move items one at a time from unsorted to sorted, inserting each into its proper place
- Sorted group grows, unsorted group shrinks
- Repeat until sorted
- Non-adaptive

Insertion Sort Example

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = left + 1; i < right; ++i)
    for (size_t j = i; j > left; --j)
      if (a[j] < a[j - 1])
        swap(a[j - 1], a[j]);
}  // insertion()
```

```
input: {4, 2, 5, 1, 3}
pass1: {2, 4, 5, 1, 3}
pass2: {2, 4, 5, 1, 3}
pass3: {1, 2, 4, 5, 3}
pass4: {1, 2, 3, 4, 5}
```

Insertion Sort Analysis

- Non-adaptive version
  - ~n^2/2 comparisons in worst case
  - ~n^2/2 swaps worst case
  - ~n^2/4 swaps average case
  - Run time only *slightly sensitive* to input (Comparisons always the same, but swaps can change)

Improving Insertion Sort

- General purpose elementary sorts involve comparisons and swaps
- Best method of improving sort efficiency is to use a more efficient algorithm
- Next best method is tighten the inner loop (comparison and swap)

Insertion Sort: First Improvement

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = left + 1; i < right; ++i) {
    Item v = a[i]; size_t j = i;
    for (~~size_t j = i~~; j > left; --j) {
      ~~if (a[j] < [j - 1])~~
        ~~swap(a[j - 1], a[j]);~~
      if (v < a[j - 1])
        a[j] = a[j - 1];
      else
        break;
    }  // for j
    a[j] = v;
  }  // for i
}  // insertion()
```

Move vs. Swap

Insertion Sort: Second Improvement

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = left + 1; i < right; ++i) {
    Item v = a[i]; size_t j = i;
    while ((j > left) && (v < a[j - 1])) {
      ~~if (v < a[j - 1])~~ a[j] = a[j - 1];
      --j;
    }  // while
    a[j] = v;
  }  // for i
}  // insertion()
```

Replace **`for`** with **`while`**
Remove the **`break`**

Insertion Sort So Far

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = left + 1; i < right; ++i) {
    Item v = a[i]; size_t j = i;
    while ((j > left) && (v < a[j - 1])) {
      a[j] = a[j - 1];
      --j;
    }  // while
    a[j] = v;
  }  // for i
}  // insertion()
```

- The `j > left` test is often performed, but seldom `false`
- Eliminate the need for frequent testing with a *sentinel*

Insertion Sort: Third Improvement

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = right - 1; i > left; --i)  // find min item
    if (a[i] < a[i - 1])                     // put in a[left]
      swap(a[i - 1], a[i]);                  // as sentinel

  for (size_t i = left + 2; i < right; ++i) {
    Item v = a[i]; size_t j = i;           // v is new item
    while (~~(j > left) &&~~ v < a[j - 1]) {   // v in wrong spot
      a[j] = a[j - 1];                     // copy/overwrite
      --j;
    }  // while
    a[j] = v;
  }  // for i
}  // insertion()
```

`j > left` is often performed, but seldom **`false`**

Eliminate frequent tests with a *sentinel*

Insertion Sort: Adaptive/Example

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = right - 1; i > left; --i)  // find min item
     if (a[i] < a[i - 1])                    // put in a[left]
      swap(a[i - 1], a[i]);                  // as sentinel

  for (size_t i = left + 2; i < right; ++i) {
    Item v = a[i]; size_t j = i;          // v is new item
    while (v < a[j - 1]) {                // v in wrong spot
      a[j] = a[j - 1];                    // copy/overwrite
      --j;
    }  // while
    a[j] = v;
  }  // for i
}  // insertion()
```

a = { 4 2 5 1 3 }	input
a = { 1 4 2 5 3 }	sentinel
a = { 1 2 4 5 3 }	pass 1
a = { 1 2 4 5 3 }	pass 2
a = { 1 2 3 4 5 }	pass 3

Adaptive Insertion Sort

- Comparisons
  - ~$n^2/2$ in worst case
  - ~$n^2/4$ in average case
  - ~n in best case
- Copies
  - ~$n^2/2$ in worst case
  - ~$n^2/4$ in average case
  - ~n in best case
- Run time *sensitive* to input

Insertion Sort Comparison

Advantages of Adaptive Insertion Sort

- More efficient data manipulation
  - Use move instead, 1/3 of the work of swap
- Fewer comparisons/copies
  - Used only when key is smaller than key of item being inserted
- Find sentinel key
  - Sentinel key is a smallest key in range
  - Adds overhead of explicit first pass to find sentinel

Insertion Sort Analysis

- Advantages
  - Run time depends upon initial order of keys in input
  - Stable
  - Algorithm is tunable
  - Best sort for small values of *n*
- Disadvantages
  - Still $\Theta(n^2)$

Insertion Sort

Data Structures & Algorithms

Sorting Performance

Data Structures & Algorithms

Performance Characteristics

- Run time proportional to
  - Number of comparisons
  - Number of swaps
  - Sometimes both comparisons and swaps
- Elementary sorts are all quadratic time ($O(n^2)$) worst and average
  - Assuming inputs are uniformly distributed
  - Better average on pre-sorted inputs

Performance Analysis

Inversion: A pair of keys that are out of order

For each item, can determine the number of inversions (number of items to left that are greater)

For each input, can determine total number of inversions

- Gives sense of *“presorted-ness”*
- Some sorts work better on presorted data

Performance Analysis

Property: Insertion and Bubble sort use **linear number of comparisons and swaps** for data with a **constant upper limit to the number of inversions for each element** (i.e. almost sorted: elements are not very far out of place)

Interpretation: Insertion sort and bubble sort are **efficient on partially sorted data when each item is close to its "final" position**

Performance Analysis

Property: Insertion sort uses a **linear number of comparisons and swaps** for data with a **constant number of elements having more than a constant number of inversions** (i.e. a small number of elements are very far from their final location)

Interpretation: Insertion sort is **efficient on sorted data when a new item is added to the sorted data**

Performance Analysis

Property: Selection sort runs in **linear time** for data with **large items and small keys** (i.e. sort 1MB/person medical records in order by a single integer: their social security number)

Interpretation: Selection sort is **expensive in terms of comparisons, but cheap in terms of swaps; thus it is *O*(*n*) in the number of swaps**

Summary: Elementary Sorts

- $O(n^2)$-time, in-place sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
    - Non-adaptive
    - Adaptive
- Counting sort is $O(n)$-time, but not in-place
- Some sorts execute more quickly in special cases

Highly recommended:
http://www.sorting-algorithms.com/
Click on one to watch it go!

Sorting Performance

Data Structures & Algorithms

Counting Sort

Data Structures & Algorithms

Counting Sort

For sorting a limited number of different keys

Example: **D F B G A F C B F D D F G**

- **First pass:** count the number of items that match each key (don't copy data, just count it)

  Example: **1 2 1 3 0 4 2**

- **Second pass:** compute offset where each key would start in sorted order

  Example: **0 1 3 4** 7 **7 11** (last used slot: 12)

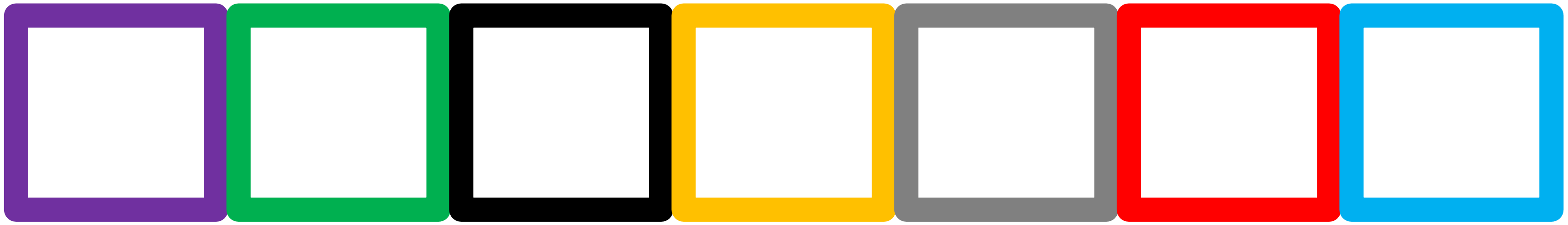
- **Third pass:** copy records into output container, grouped by key

Counting Sort

1) Count items and sum into buckets

D	F	B	G	A	F	C	B	F	D	D	F	G

2) Calculate offsets from counts



3) Copy from input to output

A	B	B	C	D	D	D	F	F	F	F	G	G

When Counting Sort Can Be Used

- ONLY usable when the number of keys is limited (and small)
  - For example:
    - Number of possible birthdays annually = 366
    - Number of possible class standings = 4
- `INT_MAX` is not “limited”
  - Changes with architecture
  - For 64 bit computers, 2^{64} 8-byte integers = 144 quintillion bytes = 160 million 10TB hard drives

Example Code

```cpp
void counting_sort(vector<Student> &students) {
  vector<Student> result(students.size());
  vector<size_t> counts(MAX);
  size_t temp, sum = 0;

  for (auto &s : students)                       // Pass 1
    ++counts[s.classStanding()];
  for (size_t i = 0; i < MAX; ++i) {             // Pass 2
    temp = counts[i];
    counts[i] = sum;
    sum += temp;
  } // for i
  for (auto &s : students) {                     // Pass 3
    result[counts[s.classStanding()]] = s;
    ++counts[s.classStanding()];
  } // for
  swap(students, result);
} // counting_sort()
```

Counting Sort Analysis

- A “distribution sort”, where items are not compared to each other
- Time complexity is linear in the number of items and buckets $O(n + k)$
- Space complexity is $O(n + k)$
- Stable
- Similar to and often used in Radix Sort

Counting Sort

Data Structures & Algorithms

Using `std::sort()`

Data Structures & Algorithms

`std::sort()` Parameters

- Expects:
  - Two iterators [begin, end)
  - A natural sort order (contained data type supports operator <)
- If no natural sort order, use the overloaded version with three parameters:
  - Two iterators [begin, end)
  - A functor

`std::sort()` Functor

- The function object determines sort order
- Assume parameters are `a, b`
  - For ascending sort, return true when `a < b`
  - For descending sort, use the same functor but reverse the arguments, `b < a`

Using `std::sort()`