Lecture 9
Elementary Sorting Methods

EECS 281: 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
- Bucket & counting sort (int, few different keys)

High-Performance Sorts

- Quick Sort
- Merge Sort
- Heap Sort
- Radix Sort (int, char*, …)

Simple ≠ Useless

- Simple 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 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(a.begin(), a.end()); // 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

Internal Sort: “I can see all elements”

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

Indirect Sort: “I can see all indices”

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

External Sort: “Too many elements”

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

Basic Building Blocks

- operator<(): compare item A and B
- operator[] : access *k*-th element
- swap(): swap item A and B
- compswap(): compare two items (A & B), and if B is smaller calls swap(), e.g., puts in ascending order
- mySort(): algorithm / implementation *du jour*
- main(): fill an array with numbers, calls sort, checks and prints result

Allow students to examine many array sorting algorithms

Exercise

- Write swap() and compswap()
  - swap(): swap item A and B
  - compswap(): compare two items (A & B), and if B is smaller calls swap(), e.g., puts in ascending order

Solution

```
1   template <typename    T>
2   void  swap(T  &a,  T  &b) {
3      T  tmp  =  a;
4      a  =  b;
5      b  =  tmp;
6   }
7
8   template <typename    T>
9   void  compswap(T &a,  T  &b) {
10     if   (b  <  a)
11        swap(a,  b);
12  }
```

Don’t do this!

```
#include <utility>
```

Use the STL `swap()` template

Uses C++11 move semantics

Desirable Qualities of Algorithms

Asymptotic complexity, number of swaps

- Worst-case
- Average-case

Memory efficiency (three cases):

- Sort in place with $O(1)$ extra memory
- Sort in place, but have pointers or indices (n items need an additional n ptrs or indices)
- Need $\Omega(n)$ extra memory

Desirable Qualities of Algorithms

Stability

- Definition: preservation of relative order of items with duplicate keys in a file
- Simple sorts tend to be stable
- Complex sorts are often *not* stable

Important for doubly-sorted lists
(sort alphabetically*, then by* SSN*)*

Types of Algorithms

Non-Adaptive Sort

- Sequence of operations 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

Bubble Sort (non-adaptive)

```
1  void bubble(item a[], int left, int right) {
2     for (int i = left; i < right - 1; i++)
3        for (int j = right - 1; j > i; j--)
4           compswap(a[j - 1], a[j]);
5  }
```

- Find minimum item on first pass by comparing adjacent items, and move item all the way to left
- Find second smallest item on second pass, …

Example

```
1  void bubble(item a[], int left, int right) {
2     for (int i = left; i < right - 1; i++)
3        for (int j = right - 1; j > i; j--)
4           compswap(a[j - 1], a[j]);
5  }
```

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

```
1         void  bubble_adaptive(item  a[],  int left,  int       right)    {
2             for(int i  =  left; i  <  right  -  1;  i++)  {
3         bool  swapped   =  false;
4         for  (int   j  =       right  -  1; j a[j   >  i;    j--)    {
5            if  (a[j] <       -  1])  {
6               swapped  =  true;
7               swap(a[j  -    1],    a[j]);
8            }
9         }
10        if  (!swapped)
11           break;
12     }
13  }
```

Bubble Sort

Analysis: non-adaptive version

- ~$n^2/2$ comparisons
- ~$n^2/2$ swaps worst/best/average case

Adaptive version

- ~n swaps (as few as 0) in best case
- ~$n^2/2$ swaps average case
  - Difficult to prove

Bubble Sort: Advantages

- Simple to implement
- Simple to understand
- Completes some ‘pre-sorting’ while searching for smallest key
- Adaptive version may finish quickly if the input array is *almost sorted*

Bubble Sort: Disadvantages

- O(n^2) time
  - $n^2/2$ comparisons and $n^2/2$ swaps
- Slower than high-performance sorts

Selection Sort (non-adaptive)

```
1  void selection(Item a[], int left, int right)
2      for(int i = left; i < right - 1; i++)
3          int min = i;
4          for (int j = i + 1; j < right; j++)
5              if (a[j] < a[min])
6                  min = j;
7          swap(a[i], a[min]);
8      }
9  }
```

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

Example

```
1  void selection(Item a[], int left, int right)                {
2     for(int i = left; i < right - 1; i++)                     {
3        int min = i;
4        for (int j = i + 1; j          < right;     j++)
5              if (a[j] < a[min])
6              min = j;
7        swap(a[i], a[min]);
8     }
9  }
```

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}

Selection Sort: Complexity Analysis

- ~$n^2/2$ comparisons
- n - 1 swaps in best, average, worst case
- Non-adaptive
  - Run time is *insensitive* to input

Selection Sort (adaptive)

```
1  void  selection(item  a[],  int left,  int  right)                {
2     for(int i  =  left; i  <  right  -  1;  i++)                 {
3        int  min  =  i;
4        for (int  j  =  i  +  1; j  <  right;  j++)
5           if  (a[j] <  a[min])
6              min  =  j;
7        if (min  !=  i) swap(a[i],  a[min]);
8     }
9  }
```

- Find smallest element in array, swap with first position
- Find second smallest element in array, swap with second position
- Repeat until sorted
  - Don’t swap if item is already in correct position

Selection Sort: Analysis 2

Analysis

- $(n^2 - n)/2 + (n - 1)$ comparisons
- $n - 1$ swaps worst case
- 0 swaps best case
- Adaptive
  - Run time is now *sensitive* to input

Selection Sort: Advantages

- Minimal copying of items
- In practice, hard to beat for small arrays

Selection Sort: Disadvantages

- $\theta(n^2)$ time
- Run time only slightly dependent upon pre-order
  - The min key on one pass tells nothing about the min key on subsequent passes
  - Runs about the same on
    - Already sorted array
    - Array with all keys equal
    - Randomly arranged array

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

Sorting Background

All sorts so far 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 (non-adaptive)

```
1  void insertion(Item a[], int left, int right) {
2    for (int i = left for + 1; i < right; i++)
3      (int j = i; j > left; j--)
4        compswap(a[j - 1], a[j]);
5  }
```

- Consider elements one at a time
- Insert each into its proper place among those already considered

Insertion Sort (non-adaptive)

Analysis

- $n^2/2$ comparisons
- $n^2/2$ swaps worst case
- $n^2/4$ swaps average case
- Run time *insensitive* to input

Insertion Sort: First Improvement

```
1   void  insertion(Item  a[],  int left,        int  right)      {
2     for (int  i =  left  +  1; i  <  right;     i++)   {
3         Item  v  =  a[i];  int j  =  i;
4         for (~~int  j  =  i~~; j  >  left;  j--)      {
5            ~~compswap(a[j  -  1], a[j]);~~
6            if (v   <  a[j   -  1])
7               a[j]  =    a[j    -   1];
8            else
9               break;
10        }
11        a[j]  =  v;
12     }
13  }
```

Move vs. Swap

Insertion Sort: Second Improvement

```
1   void insertion(Item a[], int left, int right) {
2     for (int i = left + 1; Item i < right; i++) {
3       v = a[i]; int j while = i;
4         ((j > left) && (v < a[j - 1])) {
5         if (v < a[j - 1]) a[j] = a[j - 1];
6         j--;
7       }
8       a[j] = v;
9     }
10  }
```

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

Insertion Sort So Far

```
1   void insertion(Item a[], int left, int right) {
2     for (int i = left + 1; Item i < right; i++) {
3       v = a[i]; int j while ((j = i;
4       > left) && (v < a[j - 1])) {
5         a[j] = a[j - 1];
6         j--;
7       }
8       a[j] = v;
9     }
10  }
```

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

Insertion Sort: Third Improvement

```
1   void insertion(Item a[], int left, int right)                    {
2     for(int i = right-          1; i >   left;    i--)      //  find min item
3       compswap(a[i - 1],    a[i]);                          //  put in a[left]
4                                                             //  this   is sentinel
5     for(int i = left + 2; i         < right;    i++)  {
6       Item v = a[i]; int j =        i;                      //  v is   new item
7       while ((j > left) &&  v       < a[j  - 1])  {         //  v in   wrong spot
8         a[j]   = a[j - 1];                                  //  half   swap (move)
9         j--;
10      }
11      a[j] = v;
12    }
13  }
```

`j > left` is often performed, but seldom `false`
Eliminate frequent tests with a *sentinel*

Insertion Sort: Adaptive/Example

```
1   void insertion(Item  a[],  int left,  int right)                {
2      for(int i  =  right -  1;  i >  left;                 i--)    //   find  min  item
3         compswap(a[i  -  1],  a[i]);                               //   put  in  a[left]
4
5          for (int  i  =  left  +  2;     i  <  right;    i++)   {
6         Item  v  =  a[i];     int  j     =  i;                     //   v  is    new item
7          while  (v  <  a[j    -  1])    {                         //   v  in    wrong  loc
8              a[j]  =  a[j  -    1];                               //   half    swap
9            j--;
10        }
11        a[j]  =  v;
12     }
13  }
```

insertion({ 4 2 5 1 3 }, 0 , 5):

a = { 1 4 2 5 3 }
a = { 1 4 2 5 3 }
a = { 1 2 4 5 3 }
a = { 1 2 4 5 3 }
a = { 1 2 3 4 5 }

Insertion Sort (Adaptive)

Analysis

- $n^2/4$ comparisons average case
- $n^2/2$ comparisons worst case
- $n^2/4$ swaps (moves) average case
- $n^2/2$ swaps (moves) worst case
- Run time *sensitive* to input

Insertion Sort Comparison

Advantages of Adaptive Insertion Sort

- More efficient swaps
  - Use ‘half-swaps’ instead
- Fewer comparisons/swaps
  - Used only when key is smaller than key of item being inserted
- Find sentinel key
  - Sentinel key is a smallest key in range
  - But overhead of explicit first pass to find sentinel

Insertion Sort Analysis

Advantages

- Run time depends upon initial order of keys in input
- Algorithm is ‘tune-able’

Disadvantages

- Still $O(n^2)$

Performance Characteristics

- Run time proportional to
  - Number of comparisons
  - Number of swaps
  - Or both
- Simple 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: pair of keys that are out of order in file

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

For each file, can determine total number of inversions

- Gives sense of *pre-sortedness*
- Some sorts work better on presorted data

Performance Analysis

Property: Insertion and Bubble sort use **linear number of comparisons and swaps** for files 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 files 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 file**

Performance Analysis

Property: Selection sort runs in **linear time for files 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**

Bucket(Bin) Sort for Small Universes

- Sort students by birthday (day of the year)
- Establish an array of 366 items
- **First pass**: over all students
  - Copy each student record into a bucket (bin) according to birthday
- **Second pass**: over all bins
  - Copy (push_back) each student record to the output container
- Complexity for *N* students?
- Can we sort students by hometown?

Example C++11 Code

```
1  void  binsort(vector<size_t> &A)      {
2     vector<vector<size_t>> B(MAX); // bins
3     for  (auto a : A)
4        B[a].push_back(a);
5     size_t current =      0;
6     for (size_t i      = 0;  i < MAX; ++i)  {
7        for (auto item  :      B[i])
            A[current++] =    item;
8     } //  for
9  } // binsort()
```

The code is simplified for benchmarking, it is not reusable

http://www.baptiste-wicht.com/2012/11/integer-linear-time-sorting-algorithms/

Counting Sort

- An optimization of bucket sort

  Example: **D F B G A F C B F D D F G** (13 total)

- **First pass** only counts records that would go into each bucket
  - No records are copied (⇒faster)

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

- **Second pass** computes bucket addresses

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

- **Third pass** copies records into buckets

Example C++11 Code

```
1  void  counting_sort(vector<size_t> &A)      {
2     vector<size_t>     B(A.size()), C(MAX);
3        for (auto a : A)                          //  Pass 1
4        ++C[a];
5        for (size_t i = 1; i  <= MA   ++i)        //  Pass 2
6        C[i] += C[i - 1];                  X;
7        for (auto a : A) {                        //  Pass 3
8        B[C[a]  - 1] = a;
9        --C[a];
10    } // for
11    A = B;
12 } // counting_sort()
```

The code is simplified for benchmarking, it is not reusable

http://www.baptiste-wicht.com/2012/11/integer-linear-time-sorting-algorithms/

Example C++11 Code

```
1  void  in_place_counting_sort(vector<size_t>          &A  {
2     vector<size_t>     C(MA  +  1);                       )
3       for  (auto a    :   A)                    //   Pass   1
4         ++C[a];
5       int   current    =  0;
6       for  (size_t    i   =  0;  i j  <  MA    ++i)       //   Pass   3
7        for  (size_t       =  0;      j   <  C[i];  ++j)
8           A[current++]  =     i;
9  }  // in_place_counting_sort()
```

The code is simplified for benchmarking, it is not reusable

http://www.baptiste-wicht.com/2012/11/integer-linear-time-sorting-algorithms/

Summary: Elementary Sorts

- $O(n^2)$-time, in-place sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
    - Non-adaptive
    - Adaptive
- Bucket 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!

Using STL sort()

- Expects:
  - Two iterators [start, end)
  - A natural sort order (contained data type supports operator <)
- If the second is false, there's an overloaded version with 3 parameters:
  - Two iterators [start, end)
  - A functor

STL `sort()` Functor

- The function object determines sort order
- Assume parameters are `a, b`
  - If it produces `a < b`, the sort is increasing
  - A comparison of `a > b` (or better yet, `b < a`) produces a decreasing sort order