COMP 161 - Lecture Notes - 18 - Sorting in Quadratic Time

April 30, 2015

In these notes we look at two quadratic sorting algorithms: insertion sort and selection sort.

Sorting in the C++ STL

The C++ standard template library provides an optimized version of what we'll come to know as *quicksort*[1]. It's a mutator procedure that modifies the vector in place. By default it sorts from least to greatest order. For vector v, the following will sort v:

[1] http://www.cplusplus.com/reference/algorithm/sort/

```
std::sort(std::begin(v),std::end(v))
```

You can do greatest to least order using the same mechanism that we specified the sort order for *std::binary_search*.

Sorts

The sorts we're studying utilize key strategies in algorithm design and are representative of complexity classes above linear. By having more than one example from each class, we hope to better understand the nuance and variation within each class.

Sort	Complexity
Insertion	$O(n^2)$
Selection	$O(n^2)$
Mergesort	$O(n \log n)$
Quicksort	$O(n \log n)$

Quicksort technically has a worst case of $O(n^2)$ but we can avoid it the vast, vast majority of the time by utilizing randomness. So, for all intents and purposes, it's $O(n \log n)$.

Insertion sort is what you'd expect to get by applying the basic iterative/recursive strategy. Selection sort is what you'd get if you apply a new variation on that strategy in which we effectively rearrange the data prior to proceeding to the next iteration. The authors of HtDP refer to this as Generative Recursion[2]. We'll see it in its iterative form.

[2] http://www.ccs.neu.edu/home/matthias/HtDP2e/part_five.html

Mergesort is what you'd expect to get by applying the basic divide and conquer strategy to the problem of sorting. Quicksort combines a generative version of divide and conquer recursion along with randomization.

Insertion Sort

As is now our practice, we'll begin with the complete set of code needed for insertion sort and then proceed with the analysis and design. This implementation of insertion sort uses a helper *insert_last*. Often you'll see this written as an inner-loop within *insertionsort* itself.

```
/**
 * Sort the contents of data in least to greatest order
 * @param data vector of integers
 * @return none
 * @pre none
 * @post contents of data have been sorted in least to greatest order
 */
void insertsort(std::vector<int>& data);

/**
 * Move data[lst] into sorted region data[fst..lst-1] such that
 * the whole region is sorted.
 * @param data vector of integers
 * @param fst lowest index of region for insertion
 * @param lst the location of the item to be inserted. Also
 *  one more than the last of the insertion region
 * @pre fst < lst. data[fst .. lst-1] is sorted in
 *  least to greatest order
 * @post data[fst..lst] is sorted in least to greatest order
 */
void insert_last(std::vector<int>& data,
 unsigned int fst, unsigned int lst);

TEST(insert_last,all){

  std::vector<int> testme({3,2});
  insert_last(testme,0,1);
  EXPECT_EQ(std::vector<int>({2,3}),
    testme);

  testme = std::vector<int>({2,4,5,7,2,4});
  insert_last(testme,0,4);
  EXPECT_EQ(std::vector<int>({2,2,4,5,7,4}),testme);

}

TEST(insertsort,all){
  std::vector<int> sortme;

  insertsort(sortme);
  EXPECT_EQ(std::vector<int>({}),
    sortme);

  sortme = std::vector<int>({1});
  insertsort(sortme);
  EXPECT_EQ(std::vector<int>({1}),
    sortme);

  sortme = std::vector<int>({7,4});
  insertsort(sortme);
  EXPECT_EQ(std::vector<int>({4,7}),
    sortme);

  std::vector<int> data{8,7,6,5,4,3,2,1};
  insertsort(data);
  EXPECT_EQ(std::vector<int>({1,2,3,4,5,6,7,8}),
    data);

}

  void insertsort(std::vector<int>& data){

  for(unsigned int i{1}; i < data.size(); i++){
    insert_last(data,0,i);
  }
  return;
}

void insert_last(std::vector<int>& data,
 unsigned int fst, unsigned int lst){

  for(unsigned int i{lst-1}; i >= fst && data.size(); i--){
    if( data[i+1] < data[i] ){
    std::swap(data[i],data[i+1]);
    }
    else{
   return;
    }
  }
  return;
```

```
}
```

Insertion Sort Complexity

We begin by analyzing *insertsort* in the same fashion as we did our searches. The the analysis strategy we'll use is to set aside the analysis of the helper procedure *insert_last* until we've completely analyzed insertsort. Much in the same way worked around the loop repetition in the searches, we'll work around the helper procedures used by the sorts.

The *insertsort* procedure is made up entirely of an obviously linear loop[3]. Each iteration of the loop invokes the *insert_last* helper so for a vector of size n, we'll invoke *insert_last* n times. Thus we expect the total complexity of *insertsort* to be $O(n)$ for the loop plus $n * O(f)$ for n invocations of *insert_last* where $O(f)$ is the complexity of *insert_last*.

[3] that was easy

The helper procedure *insert_last* loops over what is clearly $O(1)$ work[4]. The worst case occurs when *data[i+1]* $<$ *data[i]* is always false for all i in *[fst,lst)*. Put another way, if the vector data is sorted from greatest to least order for the index range *[fst,lst]*. The loop itself counts down from *lst-1* to *fst*. Check against *data.size()* protects against the overflow of unsigned integers. If i is 0, then $- - i$ sets i to $2^3 2$, which is presumably larger than the vector's size[5]. So, the loop repeats as many times as there are numbers in the interval *[fst,lst-1]*. This is just $lst - fst$[6]. We could say that *insert_last* is *O(lst-fst)*, which is just another way of saying *insert_last* is *linear in the size of region of the vector upon which its operating*[7].

[4] it's all elementary operations plus an $O(1)$ procedure, std::swap

[5] perhaps this is a precondition?

[6] $|[a, b]| = b - a + 1$

[7] *[fst,lst]*

We no return to *insertsort* to finish the analysis. Informally, we can probably guess that repeating a linear complexity procedure, *insert_last* a linear number of times is probably quadratic. It would be obviously quadratic if we invoked *insert_last(data,0,data.size()-1)* every time. That invocation of *insert_last* does $O(data.size())$ work. However, every time we invoke *insert_last* the size of the region of data upon which it's working is different. More specifically, we invoke *insert_last(data,0,i)* for all values of i from 1 to $data.size() - 1$.

For a vector size of n, the sum total of the work carried out by all of our calls to *insert_last* is the sum of $O(n-1) + O(n-2) + \ldots + O(2) + O(1) = O(1 + 2 + \ldots + (n-2) + (n-1))$. This sum, the sum of all the integers from 1 to some integer k has a name, the arithmetic series.It also has a more compact notation.

$$1 + 2 + \ldots + (k-1) + k = \sum_{i=1}^{k} i$$

Let's do a little formal math, just for fun.

Theorem 1.

$$\sum_{i=1}^{n} i = O(n^2)$$

Proof. We'll prove this theorem by directly finding the closed form solution to the series. This form is a quadratic form and is thereby $O(n^2)$. Let $S = \sum_{i=1}^{n} i$. Then, we can show that $2 * S = n(n+1)$ as follows:

$$\begin{array}{cccccccccc} & 1 & + & 2 & + & \ldots & + & (n-1) & + & n \\ + & n & + & (n-1) & + & \ldots & + & 2 & + & 1 \\ \hline & (n+1) & + & (n+1) & + & \ldots & + & (n+1) & + & (n+1) \end{array}$$

It's then follows that $S = \frac{n(n+1)}{2} = \frac{1}{2}(n^2+n) = O)(n^2)$. □

Returning back to *insertsort*, we know know that for our vector of size n the sum total of work carried out by all of our calls to *insert_last* is $\sum_{i=1}^{n-1} i = O((n-1)^2) = O(n^2)$. To complete the analysis we notice that this quadratic complexity of all the *insert_last* calls dominates the linear complexity of the loop that generated them and that *insertsort* has a complexity of $O(n^2)$.

The Design of Insertion Sort

Insertion sort is what you're likely to come up with if you follow the basic iterative design pattern. Let's imagine a vector with the following contents:

```
{3,5,1,7,4,9,2}
```

Now if we're thinking in terms of iterative mutation in order to sort the data, then after four iterations we should have modified the first four locations such that they're sorted. Let's show that by breaking the vector into the three pieces, the stuff we sorted[8] so far, the current element to be sorted, and everything that's left.

[8] accumulated

```
{1,3,5,7} 4 {9,2}
```

Now the question is what do we do with 4 in order to have sorted the first four elements after this, the fourth iteration? From the high-level perspective, we need to be able to *insert* 4 *into the already sorted data such that the result is still sorted*. This is exactly what *insert_last* accomplishes. By following the basic iterative structure we arrived at insertion sort. Before we move on to *insert_last*, let's settle on the base case for our iterative insertion. The classic is the empty case. We start having changed nothing. The picture looks like this:

```
{} 3 {5,1,7,4,9}
```

This will work fine as long as *insert_last* can handle the "insert into an empty space" case. However, this seems like some largely wasted work. The result we expect to get is really just this:

```
{3} 5 {1,7,4,9}
```

It doesn't matter what the first number was, it always ends up in spot zero. So why not start by inserting the second number into the the space of the first. This is what you see in our implementation. Let's start with a trivially sorted space, the space of just one item, and build up from there rather than start with going from trivial case, empty spaces are sorted, to trivial case, one number is sorted.

Now that we've identified the helper procedure and it's specifications, we can turn our attention to its implementation. Once again, basic iterative design should do the trick. The catch this time is that we're not working a full vector, just *[fst,lst]*, and that we need to traverse the vector not from *fst* to *lst* but lst to first. We begin with a picture like this:

```
{1,3,5,7} 4
```

We then proceed using iterative swaps and early termination once we no longer need to swap.

```
{1,3,5} 4 {7}
{1,3} 4 {5,7}
{1,3,4,5,7}
```

Iteration and Quadratic Complexity

Theorem (1) is exceedingly useful in complexity analysis. A great many procedures make use of a structure similar to that of insertsort. If we tease out the pattern from insertsort, then we see a nested loop structure. The outer loop is classically linear, the inner loop is linear but *dependent on the current state of the outer loop*. Like this:

```
for( int i{0}; i < n ; ++i){
  for(int j{0}; j < i ; ++j){
   ...
  }
}
```

The *total* cost of the inner loop here is exactly what we saw with insertsort: $\sum_{j=0}^{n-1} j = O(n^2)$. This quadratic cost dominates the linear cost of the loop that drives it. With *insertsort* that inner loop was hidden within a helper procedure.

A more overt form of quadratic looping can sometimes occur:

```
for( int i{0}; i < n ; ++i){
  for(int j{0}; j < n; ++j){
   ...
  }
}
```

Here the inner-loop does $O(n)$ work regardless of the value of i. So, when we repeat it n times, the total work is obviously $O(n^2)$.

Finally, we should notice that, once again, varying the step size of either loop in either case will not impact the overall complexity. It simply reduces the actual work by some fixed fraction. This falls inline with the general intuition that both of the nested loops shown here are loops of linear complexity and $O(n * n) = O(n^2)$.

Selection Sort

Like *insertsort*, *selectsort* utilizes the iteration of a helper procedure. Take a moment to look at the the two procedures and the big picture and then we'll proceed with the analysis and the design.

```
  /**
   * Sort the contents of data in least to greatest order
   * @param data a vector of integers
   * @return none
   * @pre none
   * @post data has been sorted in least to greatest order
   */
  void selectsort(std::vector<int>& data);

  /**
   * Compute the index of the smallest item in data from index
   * range [fst,lst).
   * @param data vector of integers
   * @param fst the first index in the search range
   * @param lst the index after the last index in the search range
   * @return the index from [fst,lst) where the min value for that range
   *   in data can be found
   * @pre fst < lst (i.e. data is at least size 1)
   * @post none
   */
  unsigned int min_idx(const std::vector<int>& data,
       unsigned int fst,unsigned int lst);


  TEST(min_idx,all){
```

```
  std::vector<int> data{2,3,394,1,9,8,5,7,9};

  EXPECT_EQ(0,min_idx(data,0,1));
  EXPECT_EQ(0,min_idx(data,0,3));
  EXPECT_EQ(6,min_idx(data,4,data.size()));
  EXPECT_EQ(3,min_idx(data,0,data.size()));

}

TEST(selectsort,all){

  std::vector<int> sortme;

  selectsort(sortme);
  EXPECT_EQ(std::vector<int>({}),
    sortme);

  sortme = std::vector<int>({1});
  selectsort(sortme);
  EXPECT_EQ(std::vector<int>({1}),
    sortme);

  sortme = std::vector<int>({7,4});
  selectsort(sortme);
  EXPECT_EQ(std::vector<int>({4,7}),
    sortme);

  std::vector<int> data{8,7,6,5,4,3,2,1};
  selectsort(data);
  EXPECT_EQ(std::vector<int>({1,2,3,4,5,6,7,8}),
    data);

}

void selectsort(std::vector<int>& data){

  for(unsigned int i{0}; i < data.size() ; i++){
    unsigned int next_min_idx = min_idx(data,i,data.size());
    std::swap(data[i],data[next_min_idx]);
  }
}

unsigned int min_idx(const std::vector<int>& data,
     unsigned int fst,unsigned int lst){

  unsigned int curr_idx = fst;
  for(unsigned int i{fst+1} ; i < lst ; i++ ){
    if( data[curr_idx]  > data[i] ){
    curr_idx = i;
    }
  }
  return curr_idx;
}
```

Selection Sort Complexity

The main procedure, *selectsort* utilizes the now very familiar linear complexity loop to manage the iteration. The operations within the loop are an $O(1)$ swap and the call to the helper *max_idx*. Right away we notice that *max_idx* is called with each *i* in *[0,data.size())*. Without even getting into the *min_idx* analysis, we might guess that the exact same quadtratic pattern we saw in *insertsort* was going on[9]. Let's look to be sure.

[9] we'd be right

The helper procedure *min_idx* has a loop that's doing $O(1)$ work per iteration. The loop counts from *fst+1* to *lst-1* in steps of one. This should be familiar from *insert_last*. The loop is linear in the size of the interval *[fst+1,lst-1]*[10]. This means that by repeatedly calling it with a vector of size n and with i taking on the values in $[0, n)$ generates the following series:

[10] *lst-fst-2*

$$\sum_{i=0}^{(n-2)} i = O((n-2)^2) = O(n^2 - 4n + 4) = O(n^2)$$

The Design of Selection Sort

The underlying principle behind selection sort is GENERATIVE RECURSION. We're just implementing it iteratively[11]. With this strategy we don't restrict ourselves to only the data we've encountered as we traverse the vector, instead we rearrange the vector at each step so that the problem is simplified at the next iteration.

[11] Generative Iteration

The underlying picture of selection sort is similar to insertion sort. On the kth step of iteration the first $k-1$ items should be in sorted order and the rest have yet to be sorted. The kth step will then sort one more item. The difference is that we don't need to sort item k nor are the first $k-1$ items the original first $k-1$ items in the original vector.

The strategy selection sort uses to *select* the smallest item in the $k-1$ unsorted items and put that item at location k. We do this by simply swapping that item with whatever is at location k. The procedure *min_idx* finds the minimum value of a region of the vector and returns the index at which it can be found. Let's look at what happens. With this original vector

```
{3,5,1,7,4,9,2}
```

First we find the global min and swap it with whatever is at location zero, then the next min goes to location one, and so forth. We begin with no accumulated change. The element at the location being sorted is pulled out, but notice that we're not sorting it, we're just swapping it with the next min.

```
{} 3 {5,1,7,4,9,2}
{1} 5 {3,7,4,9,2}
{1,2} 3 {7,4,9,5}
{1,2,3} 7 {4,9,5}
{1,2,3,4} 7 {9,5}
{1,2,3,4,5} 9 {7}
{1,2,3,4,5,7}  9
{1,2,3,4,5,7,9}
```

The generative strategy is less formulaic than standard iteration because we must discover a way to rearrange around the current item in our vector. Selection sort replaces the item with what should be in that location in the final solution– the next smallest item in the unsorted data.

Insertion vs. Selection

Despite using two very different design strategies, both insertion sort and selection sort belong to the quadratic complexity class. The culprit is stepwise iteration. Both the main sort procedures and the iterated helpers utilize linear loops. If we wish to break out of this pattern, we need one or both of the loops to have sub-linear complexity. Binary search gives us a clue as to how this can be achieved– divide and conquer.

Before we dive into new sorts, let's consider how selection and insertion differ and which might exhibit better performance in practice. The outer loops in both procedures are effectively identical. They traverse the entire length of the vector and do not terminate early. The inner loop of selection sort also traverses the complete part of the vector it's searching for the min in and never terminates early. Insertion sort's inner loop, on the other hand, will stop early when the item being inserted gets to the right spot. This implies that insertion sort is capable of doing less work if we're only concerned about the loops. In fact, it's not too hard to see that if the vector is already sorted, then insertion sort requires on $O(n)$ work where selection sort is always, without fail, $O(n^2)$.

The last thing to consider is that for each item sorted, selection sort will perform one and only one swap. Insertion sort must swap the item being sorted with each of the currently sorted items greater than it. In the worst case this means a linear number of swaps and that the total work done by insertion sort is more than selection sort. So, if we're expecting to sort data that's nearly sorted but in the reverse order from what we want, insertion sort is not a good choice relative to selection sort. On the other hand, if the data is nearly sorted, insertion sort stands a chance of exhibiting near linear time behavior. In the end, choosing the right algorithm is a function of knowing the actual data you'll be working with as well as knowing the performance characteristics of your algorithm choices on that data.