COMP 161 - Lecture Notes - 19 - Sorting in $O(n \log n)$ Time

May 5, 2015

In these notes we look at the divide and conquer driven sorts mergesort and quicksort. Mergesort is in $O(n \log n)$ time and the randomized version of quicksort discussed here is .

Mergesort

Mergesort uses recursive divide and conquer to break sorting of out of the $O(n^2)$ complexity class. We cannot select sub-sections of a vector in constant time so we use the strategy of recursing over the index interval rather than the vector itself. The procedure *mergesort* is overloaded so that a single argument version that sorts the entire vector is available[1] and can make use of the index-based recursive procedure. For our purposes, we're assuming we only ever call the single argument *mergesort* directly and that it then calls the three argument variant as a helper. This is illustrated in the tests.

[1] this is the expected interface to sorting

```
/**
 * Sort the contents of data in least to greatest order
 * @param data vector of integers to be sorted
 * @return none
 * @pre none
 * @post data now contains the same numbers but sorted
 */
void mergesort(std::vector<int>& data);

/**
 * Sort the contents of data in the index range [first,last] in
 * least to greatest order
 * @param data vector of integers to be sorted
 * @param fst least index of the region to be sorted
 * @param lst greatest index of the region to be sorted
 * @return none
 * @pre fst,lst < data.size()
 * @post data from fst to lst now contains the same numbers but sorted
 */
void mergesort(std::vector<int>& data,unsigned int fst, unsigned int lst);

/**
 *
 *@param data vector containing portions to be merged
```

```
 *@param lfst least index of left region
 *@param llst greatest index of the left region
 *@param rfst least index of right region
 *@param rlst greatest index of the right region
 *@return none
 *@pre lfst<=llst rfst<=llst llst == rfst-1. data from [lfst,llst] and
 *  [rfst,rlst] are sorted
 *@post data from lfst to rlst is sorted
 */
void merge(std::vector<int>& data, int lfst, int llst, int rfst, int rlst);


 TEST(mergesort,all){

  std::vector<int> sortme;

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

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

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

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

}

  void mergesort(std::vector<int>& data){
  if( data.size() > 1 ){
    mergesort(data,0,data.size()-1);
  }
  return;
}
```

```
  void mergesort(std::vector<int>& data,unsigned int fst, unsigned int lst){
    if( fst >= lst ){
      return;
    }

    int mid = (lst+fst)/2;
    mergesort(data,fst,mid);
    mergesort(data,mid+1,lst);
    merge(data,fst,mid,mid+1,lst);

    return;
  }

  void merge(std::vector<int>& data, int lfst, int llst, int rfst, int rlst){

    if( data[llst] <= data[rfst] ){
      //already sorted. no merge needed
      return;
    }

    int lcurr{lfst};
    int rcurr{rfst};
    std::vector<int> merged(rlst-lfst+1);
    unsigned int mcurr{0};

    // get values from both
    while(lcurr <= llst && rcurr <= rlst  ){
      if( data[lcurr] <= data[rcurr] ){
merged[mcurr] = data[lcurr];
++lcurr;
      }
      else{
merged[mcurr] = data[rcurr];
++rcurr;
      }
      ++mcurr;
    }

    //finish out left
    while( lcurr <= llst ){
merged[mcurr] = data[lcurr];
++lcurr;
++mcurr;
    }
```

```
    //finish out right
    while( rcurr <= rlst ){
merged[mcurr] = data[rcurr];
++rcurr;
++mcurr;
    }

    //copy merged to data
    for(unsigned int i{0}; i < merged.size(); ++i){
      data[lfst+i] = merged[i];
    }

    return;
  }
```

Complexity Analysis of Mergesort

We're concerned with the complexity of the mergesort procedure that takes a single argument as it's the intended call to make in order to sort an entire vector of integers. However, it's quite clear that the complexity of this procedure is equivalent to the complexity of calling *mergesort(data,0,data.size()-1)* as the remainder of the procedure is constant time work.

Everything we've analyzed thus far has used loops for repetition. Mergesort uses recursion. The essential task is the same. Analyze the complexity of the work being repeated, determine the amount of repetition, and analyze the product. Before moving forward it's helpful to establish our vector size n in terms of the procedure's arguments. Working with a single size variable is easier than the implied size we're given with *fst* and *lst*. We've seen this before. Given *fst* and *lst* we can compute the size n of the interval *[fst,lst]*as *lst-fst+1*. Now when we say n what we mean is *lst-fst+1*.

The recursive *mergesort* does a little bit of $O(1)$ work, makes two recursive calls to mergesort, and one call to the helper *merge*. The recursive calls sort the data from *[fst,mid]* and *[mid+1,lst]*, respectively, where *mid* is the middle index of *[fst,list]*. In plain english, the two recursive calls sort the lower and upper halves of the vector. The helper *merge* acts on the same n elements of the vector that the *mergesort* calling it is sorting. It might seem like we know very little, bu t that doesn't stop us from saying something very concrete.Let's say the time complexity function[2] for *mergesort* is f and for *merge* is g. Then

[2] number of operations as a function of vector size

we know the form of f exactly.

$$f(n) = 2 * f(\frac{n}{2}) + g(n)$$

We also can tell that $f(0) = f(1) = 0$. It shouldn't be surprising that the time complexity function for a recursive procedure can be specified with yet another recursive function. The recursive specification of a mathematical function is called a RECURRENCE RELATION. In future classes you'll learn some techniques for solving them, which in this context means finding an equivalent function that doesn't use recursion. For now, we'll just do some careful analysis to figure out what's going on with this recurrence.

Let's pick a nice, manageable n and trace this recurrence to completion to see what happens.

$$\begin{aligned}
f(8) &= 2 * f(4) + g(8) \\
&= 2 * (2 * f(2) + g(4)) + g(8) \\
&= 4f(2) + 2g(4) + g(8) \\
&= 4(2 * f(1) + g(2)) + 2g(4) + g(8) \\
&= 8f(1) + 4g(2) + 2g(4) + g(8) \\
&= 4g(2) + 2g(4) + g(8)
\end{aligned}$$

First that's note that the recursive calls to f eventually hit the base case and become zero. Along the way, we leave a trail of calls to g. What does this tell us about *mergesort*? Well, it's really all about *merge*, which requires $O(g(n))$ work, and that *mergesort* itself really just lays out a sequence of *merge* calls that eventually sorts the vector. It also tells us that to understand the complexity of *mergesort* we just need to understand how many times *merge* gets called and what the complexity, g, of merge sort happens to be. Let's do something different. Let's figure out how often mergesort calls merge and with what sizes. Let's analyze the repetition before we analyze the work being repeated.

Going back to the mergesort recurrence we can see that n decreases by half at each "layer"[3] until it eventually hits 1 or 0. We know this pattern. When n is a power of 2, then here will be $\log_2(n)$ layers until we hit the base cases $f(1)$. At layer k we seem to be adding in a new term with the following pattern $2^k * g(\frac{n}{2^k})$. We can actually express this whole thing with summation notation.

$$\sum_{k=0}^{\log_2 n - 1} 2^k g(\frac{n}{2^k})$$

[3] with each substitution of a new f

This looks terrible because of those exponential, but we have to remember that this sum has only $\log n$ terms to it. We know mergesort is $O(n \log n)$, so we can now *guess* what the complexity of merge is. If $g(n) = O(n)$, then every term of this sum becomes n because the exponential terms cancel out.

$$\sum_{k=0}^{\log_2 n - 1} 2^k \frac{n}{2^k}$$

This leaves us with a sum of $O(\log n)$ terms each with the value n for a grand total of $n \log n$.

If we think of merge as creating a vector of size n form two vectors of size $\frac{n}{2}$ then the analysis is a little easier to tease out. Merge first creates a vector into which the data is merged. Once the merging is done the data is copied back to the original vector. The copy clearly has $O(n)$ complexity. What's less clear is that the merging takes $O(n)$ as well. A good way to see that it must is to look at *mcurr*. This variable tracks the current location to put data in the vector *merged* and rightfully starts at 0. Each iteration of all three while looks increments this variable by 1 exactly once. If this merges the $\frac{n}{2}$ data form the left half with the $\frac{n}{2}$ data from the right into the n locations in *mereged*, then the total number of iterations performed by all three while loops *must* be n. Everything else we see is $O(1)$ work with the exception of the construction of merge, which requires another $O(n)$ work[4]. Thus the total work done by *merge* is three linear operations, construction, merging, copying giving us $O(3n) + O(1) = O(n)$.

[4] `http://www.cplusplus.com/reference/vector/vector/vector/`

The Design of Mergesort

The basis of the mergesort design is to view the vector recursively as either empty, size one, or a a left and right half vector. The base cases are trivially sorted. When the vector has at least two elements in it, then we recursively sort the left and right sub-vectors. This creates a new problem: putting two sorted vectors together inorder to create one single sorted vector. That is the exact problem we design *merge* to solve.

Merge is really just basic iteration. We accumulate a sub-solution with the data we've traversed so far. However, rather than iterating on a single vector, we're iterating over two vectors in parallel.[5] Usually we iterating by counting through the input vector. This gets hairy given that we have two input vectors and we need to conditionally step through each selecting the minimum between the two options. The version you see above manages This by a series of loops. The first works as long as there is data from each vector available.

[5] Those two vectors also happen to be two contiguous regions of the same vector.

It selects one item at a time, so one vector counter advances and one does not. As soon as one vector is exhausted, then the loop terminates. The remaining two loops then move all the data from the remaining vector into our accumulator vector. One of these loops is guaranteed to not do any iterations.

There's an alternative that is probably a bit easier to understand but that will require more actual work[6] than the three loop version. Rather than work in terms of index values for the input vector, we can iterate over each index of the accumulator. Think, "for each spot in the accumulator, find the right data from the two sorted vectors". Here's how we might manage that with a for loop.

[6] as in same complexity but more work per vector element

```
for(unsigned int i{0}; i < merged.size(); ++i){
    if( lcurr <= llst && rcurr <= rlst ){
   if( data[lcurr] <= data[rcurr] ){
      merged[i] = data[lcurr];
     ++lcurr;
   }
      else{
      merged[i] = data[rcurr];
     ++rcurr;
      }
   }
   else if( lcurr <= llst ){
  merged[i] = data[lcurr];
    ++lcurr;
    }
   else{
    merged[i] = data[rcurr];
   ++rcurr;
   }
  }
```

Thinking in terms of the output rather than the input(s) is often really helpful for simplifying logic. In this case, it let's us work out of a very, very basic *for* loop rather than three separate loops. A really good exercise would be to view the first version as an optimized version of the loop shown above and to pick through them until you see what work is minimized and how it's minimized.

Quicksort

Like *mergesort*, *quicksort* is overloaded[7] to allow a "natural" signature as well as the signature we need to efficiently make recursive calls to sort sub-sections of the vector.

[7] multiple signatures

```
/**
   * Sort data in least to greatest order
   * @param data the vector to be sorted
   * @param prng random number generator used in sorting
   * @return none
   * @pre prng has been seeded
   * @post data is sorted and prng has produced O(log(data.size())) numbers
   */
  void quicksort(std::vector<int>& data, std::default_random_engine& prng);

  /**
   * Sort data from fst to least in least to greatest order
   * @param data the vector to be sorted
   * @param fst the least index of the region to sort
   * @param lst the greatest index of the region to sort
   * @param prng random number generator used in sorting
   * @return none
   * @pre prng has been seeded. fst <= lst < data.size()
   * @post data from fst to lst is sorted and prng has produced O(log(|[fst,lst]|) numbers
   */
  void quicksort(std::vector<int>& data,
 unsigned int fst, unsigned int lst,
 std::default_random_engine& prng);

  /**
   * Move the numbers in data from fst to lst such that there exists a pivot element where
   * all the items to its left are less than or equal to it and all items to the right are
   * greater than it. The pivot index is returned.
   * @param data vector of integers to be partitioned
   * @param fst least index of the partition space
   * @param lst greatest index of the partition space
   * @param prng random number generated used in pivot selection
   * @return the pivot index
   * @pre fst < lst < data.size()
   * @post for pivot index p, data[fst..p-1] <= data[p] < data[p+1..lst]. One number produced by prng.
   */
  unsigned int partition(std::vector<int>& data,
 unsigned int fst, unsigned int lst,
 std::default_random_engine& prng);

TEST(quicksort,all){

    std::vector<int> sortme;
    std::default_random_engine prng{1};
```

```
  quicksort(sortme,prng);
  EXPECT_EQ(std::vector<int>({}),
    sortme);

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

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

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


void quicksort(std::vector<int>& data, std::default_random_engine& prng){

  if( data.size() > 1 ){
    quicksort(data,0,data.size()-1,prng);
  }
  return;
}

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

  if( fst >= lst || fst >= data.size() || lst >= data.size() ){
    return;
  }

  unsigned int pidx = partition(data,fst,lst,prng);
  quicksort(data,fst,pidx-1,prng);
  quicksort(data,pidx+1,lst,prng);
  return;
}

unsigned int partition(std::vector<int>& data,
unsigned int fst, unsigned int lst,
```

```
 std::default_random_engine& prng){


    if( fst+1 == lst ){
      // 2 items
      if(data[fst] > data[lst] ){
// out of order. swap
std::swap(data[fst],data[lst]);
      }
      // let the pivot be fst
      return fst;
    }

    // 3+ items
    // choose random item for pivot. move to fst
    std::uniform_int_distribution<unsigned int> gen{fst,lst};
    std::swap(data[fst],data[gen(prng)]);

    // set region pointers
    unsigned int last_s1{fst};

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

if( last_s1 < (i-1) ){
  std::swap(data[i],data[last_s1+1]);
}
last_s1++;
      }
    }

    if( last_s1 != fst ){
      std::swap(data[last_s1],data[fst]);
    }
    return last_s1;
  }
} //end namespace searchsort
```

Complexity Analysis of Quicksort

Understanding the complexity of *quicksort* is complicated by the fact that the workhorse, *partition*, is randomize. However, knowing what

we know from analyzing mergesort we can start to tease out some possibilities. First we notice that, like *mergesort*, quicksort makes two recursive calls on subsections of the vector. In this case we're recursing on *data[fst..pidx-1]* and *data[pidx+1..lst]* for some pivot index *pidx* that comes from the interval *[fst,lst]*. Each call makes a single call to *partition* on the entire region *data[fst..lst]*.

If that pivot is the middle of the region that we're doing the exact same divide and conquer pattern we did with merge sort. We make two recursive calls to regions roughly half the size of the original and a call to *partition*. For a vector of size n it will take $O(\log n)$ steps to chop the vector down to empty or singleton[8] regions. If *partition* has linear complexity then we already know that the complexity of *quicksort* (if the subsections are half the size of the original *always*) is $O(n \log n)$.

[8] size 1

The pivot won't always be the exact middle. Let's see what happens if it's always the first[9]. If the pivot *pidx* is *fst*, then the sub-vector *data[fst..(fst-1)]* is effectively the empty vector. This is a base case for *quicksort* and requires $O(1)$ work to sort out. However, the other region, *data[fst+1..lst]* is exactly one element smaller than the original vector. If we continue end up with the first item as the pivot, then we'll need $O(n)$ recursive calls to chop the "left" region down to the base case and we'll call *partition* on vector If *partition* has complexity $O(g(n))$ then we'll end up call partition on with all the sizes from n down to 1 or 2. We've seen this before -

[9] The same logic will apply if the pivot is always the last of the region

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

When the function g is linear then we end with with a complexity of $O(n^2)$. Linear seems like a pretty fair assumption for *partition*[10], so this implies a worst case complexity for *quicksort* of $O(n)$.

[10] if we can partition with simple iteration then it will be $O(n)$

Notice now that we've covered the two extremes of *quicksort*. The further our split gets from half and half, the closer we get to the none and all but one split that leads to quadratic complexity. If we could always choose the median value[11] then we'd be set, but we can't know that a priori so it'll take some non-constant work to find that[12]. If we always choose a value at a specific location, then we can always find an original vector state that trips the worst case. That is to say, there's always a chance the the location we choose is always occupied with the min or max value. For example, if you always let the first element of the sub-vector serve as the pivot value, then sorted vectors will trip the worst case $O(n^2)$ complexity.

[11] the middle value

[12] for fun you should develop some procedures for finding the median and then go research the optimal algorithm

What we need is to *randomly choose one of the values to be the pivot value*. This works because we can show that when you do this you have a fifty-fifty chance of getting a value that's "in the middle"

versus one that's close to the min/max. So, you might get some pivots that trigger worst case like performance, but you'll also get some that give you best case performance. In the end, we get a good enough mix that the result is more $O(n \log n)$ than it is $O(n^2)$. You're about as likely to get $O(n^2)$ performance as you are to get n heads in a row when flipping a fair coin[13].

[13] remember we're talking *large n*

The Design of Quicksort

We arrived at selection sort by using the generative strategy with basic iteration. The core of this idea is to not work with the data as it comes but work with it in a way that respects the problem at hand. The end result is usually that you modify the data that gets processed next so that what comes next is easier. Quicksort applies this principle to divide and conquer recursion.

In mergesort we cut the data in half, sorted whatever was in each half, then merged the result. In sorting each half we're likely to move a lot of data that then later gets moved again in the merge. For example, if the max value is in the left sub-section of the vector, then we'll move it around a bunch to get it into the last place of the left sub-half, then move it again when we merge it with the right sub-half.

To avoid this we need re-organize the data before we split it into two chunks. The split we want is to put the smallest values[14] in the left half and the largest[15] in the right half. If we can do that, then we never have to move these items from those regions. Sorting those two-halves would mean the entire vector is sorted.

[14] in statistics speak we mean the first and second quartile

[15] the third and fourth quartile

Getting this *partition*[16] perfect would require knowing the median value so that we can move data with respect to that value. It is not possible to do this in a time that doesn't overwhelm the gains we'd get from using divide and conquer. So we'll half to settle for a less than perfect partition. So rather than give up, we can try a less than perfect split. This is what you see with *partition*.

[16] yes, this is what our helper does for us

Partition's goal is to pick a value from the vector at random. That value acts as the pivot. To simplify the partitioning, we swap that value with the first value. Our goal now is to partition everything else into the two halves: those less than or equal to the pivot and those greater. To do this we use basic iteration. Traverse the data, if the current item is in the smaller group, then swap it with the first item in the larger group. If it's in the larger group, then we don't need to move it. When we're done, we can swap the pivot with the last of the smaller half.Watch this play out. We'll leave the pivot to the right of the "vector" and we'll use separate logical vectors for the two halves.

```
5 {} {} {3,4,7,8,2,9}
```

```
5 {3} {} {4,7,8,2,9}
5 {3,4} {} {7,8,2,9}
5 {3,4} {7} {8,2,9}
5 {3,4} {7,8} {2,9}
5 {3,4,2} {8,7} {9}
5 {3,4,2} {8,7,9}
{2,3,4} 5 {8,7,9}
```

Using swaps, as opposed to shifting data, saves a lot of work. It's made possible by the fact the do not care about the order of the two regions, just that they're smaller and larger than the pivot, respectively.