CPSC200 Class - October 25/07

Robert Pringle

October 24, 2007

Sorting Algorithms and Considerations

- There are a number of sorting algorithms available, examples of these include Quick Sort, Insertion Sort and Merge Sort just to name a few.
- When considering which sorting algorithm to use you should consider the situation in which you are using it with such things as:
  1. The time complexity (worst-case, average-case, best-case where appropriate).
  2. The storage complexity (worst-case, average-case, best-case where appropriate).
  3. The stability of the sorting algorithm.
  4. Other general considerations, including special cases, for the sorting algorithm.
- We will refer to these metrics for gauging the appropriateness of a sorting algorithm as its **Report Card**.

Sort Stability

- Stability in a sorting algorithm refers to the ordering of equal elements in the collection being sorted.
- Stable sorting algorithms will leave the order of equal elements the same relative to themselves from the unsorted to the sorted collection while unstable sorting algorithms will not guarentee this.

Sort Stability Example

Consider the following unsorted list on a collection of integers where we differentiate equals elements through bolding below we have examples of possible results from a stable and unstable sorting algorithm.

Unsorted Collection:	5	3	9	**3**	1
Stable Sorted Collection:	1	3	**3**	5	9
Unstable Sorted Collection:	1	**3**	3	5	9

Merge Sort

- Merge sort is a divide and conquer algorithm for sorting collections of elements that works by dividing a collection into smaller collections, sorting these divisions and then merging them back together in sorted order.
- Merge sort generally requires temporary storage at least equal to the size of the original collection in order to perform the merge sort however it is possible to perform merge sort without such overhead when using lists.
- Each merger in general requires a comparison of elements in one of the collections to the other it is being merged with.
- Merge sort is generally a stable sorting algorithm (as it is simple to make the merge step stable).

Merge Sort Algorithm

- The general merge sort algorithm involves the following steps:
  1. Split the provided collection into two if there are enough elements, if there are less than two elements than it can be said the given collection is sorted and there is nothing further to do.
  2. Merge sort the collections formed from the split.
  3. Merge the sorted collections together to form the sorted collection.
- Note the merge sort algorithm generally uses temporary storage to store the intermediate results of sorting before merging the results back into the target.

C++ Merge Sort Algorithm

- In the notes provided for this class you are given a merge sort algorithm implemented with C++ and STL.
- The **mergeSortTo** function provided will sort the collection starting at the iterator **begin** and ending at the iterator **end** into a (possibly) new space specified by the iterator **to** and the function itself returns an iterator one past the range where the sorted data has been put.
- The **mergeRange** function takes two collections whose beginning and ending iterators are provided the performs a sorting merge on them into the memory space whose start is specified by the iterator **destination** and returns a iterator one past the range where the sorted data has been put.
- The **middleOf** function is used to find the middle point of a collection.

mergeRanges Function

```
template <typename Iterator1,
          typename Iterator2,
          typename Iterator3>
Iterator3
mergeRanges(Iterator1 begin1,
            Iterator1 end1,
            Iterator2 begin2,
            Iterator2 end2,
            Iterator3 destination)
{
    while (begin1!=end1 && begin2!=end2)
        {
        if (*begin2 < *begin1)
            *destination++ = *begin2++ ;
        else
            *destination++ = *begin1++ ;
        }
    while (begin1!=end1)
            *destination++ = *begin1++ ;
    while (begin2!=end2)
        *destination++ = *begin2++ ;
    return destination ;
}
```

mergeSortTo Function

```
37 template <typename Iterator>
38 Iterator
39 mergeSortTo(Iterator begin, Iterator end, Iterator to, Iterator temp)
40 {
41     Iterator middle=middleOf(begin,end) ;
42     if (begin==end)
43         {
44         // base case: range is size 0
45         return to ;
46         }
47     else if (begin==middle)
48         {
49         // base case: range is size 1
50         *to++ = *begin ;
51         return to ;
52         }
53     else
54         {
55         // we have range of at least size 2.  recursion kicks in
56         Iterator temp2 = mergeSortTo(begin, middle, temp, to) ;
57         Iterator temp3 = mergeSortTo(middle, end, temp2, to) ;
58         return mergeRanges(temp, temp2, temp2, temp3, to) ;
59         }
60 }
```

middleOf Function

```
29  template <typename Iterator1>
30  Iterator1 middleOf(Iterator1 b, Iterator1 e)
31  {
32      Iterator1 middle(b) ;
33      std::advance(middle, std::distance(b,e)/2) ;
34      return middle ;
35  }
```

middleOf Function

```
18  template <typename Iterator>
19  void mergeSort(Iterator begin, Iterator end)
20  {
21      typedef typename std::iterator_traits<Iterator>::value_type elt_t ;
22      std::vector<elt_t> temp(begin, end) ;
23      mergeSortTo(begin, end, begin, temp.begin()) ;
24      return ;
25  }
```

test Function

```
27  bool test(int i)
28  {
29      assert(i>=0) ;
30      std::vector<int> v ;
31      for(int j=0;j<i;++j)
32          {
33          v.push_back(j) ;
34          }
35      std::random_shuffle(v.begin(), v.end()) ;
36      mergeSort(v.begin(), v.end()) ;
37      return isSorted(v.begin(), v.end()) ;
38  }
```

main Function

```
40  int main()
41  {
42      unsigned long seed(time(0)) ;
43      srand(seed) ;
44      srand48(seed) ;
45      std::cout << "seed is " << seed
46                << "; first rand() is " << rand()
47                << "; first lrand48() is " << lrand48()
48                << "." << std::endl ;
49
50
51      bool everythingsOK = test(0) && test(1) && test(2) && test(10000) ;
52      std::cout << (everythingsOK ? "tests pass." : "tests fail.")
53                << std::endl ;
54      return (everythingsOK ? 0 : 100) ;
55  }
```

Merge Sort Evaluation

- You can see from looking at the merge sort algorithm that its time and space complexity are propotional to the number of the elements in given collection and not on the current state of the elements in the collection (as would be the case with insertion sort or quick sort).

Merge Sort Evaluation

- The time complexity of the given merge sort algorithm is the same for the worst and the average case and is $\Theta(nlogn)$.
  - Given each merge sort step we can see the time required to perform a merge sort on a collection of size n is
$$T(n) = \begin{cases} C_0, & \text{if n=0} \\ C_1, & \text{if n=1} \\ T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + C_2 n, & \text{otherwise} \end{cases}$$
  - The equation for the time required by the merge sort algorithm is representitive of a telescoping sum that we can simply to our complexity of $\Theta(nlogn)$
  - This can also be seen by looking at the algorithm from the perspective of the elements, you can see the steps that are required are the merger of 2,...,n/2 sized collections which is equivalent to $log_2(n)$ different merges over n elements with each merge taking a constant time giving us $\Theta(1 * n * log_2 n) = \Theta(nlogn)$.