Algorithm Efficiency and Sorting

1 Efficiency of Sorting Algorithms

1.1 Preliminaries

Sorting is the act of rearranging data so that it is in some pre-specified order, such as ascending or descending order. The exact ordering does not really matter, as long as it is a linear, or total ordering, which means that any two elements can be ordered. For simplicity, we will assume that all sorting algorithms put their elements in ascending order.

Sometimes we sort scalar objects such as numbers, and other times we might want to sort non-scalar objects such as records with multiple members. In the latter case, one must designate a specific member to be used as the ***sort key***. A ***sort key*** is part of the data item that determines the ordering relation used in the sort.

Sometimes the data to be sorted is so large that it cannot fit in memory. The sorting algorithms in this case have very different requirements, since only a fraction of the data can be examined at a time. Algorithms that sort data that cannot all be stored in memory are called ***external sorting algorithms***. Those that sort data that can fit entirely in memory are called ***internal sorts***.

Sometimes the data to be sorted does not have unique keys and multiple elements can have the same key. It may be important for a sorting algorithm to preserve the relative order of elements with equal keys. Sorts that preserve the order of equal keys are called ***stable sorts***. To give an example, suppose that data is sorted by last name, and if there are any elements with equal last names, then ties are broken by sorting by first name. One way to do this is to first sort all of the data by first name. Then, the data can be sorted by last name, as long as it has the property that elements with equal keys stay in the same relative order. This will work because if there are two people with the same last name but different first names, then the second sort will move elements with different last names relative to each other, but those with equal last names will stay in the same relative order, and since they are already sorted by first name, those with smaller first names and equal last names will precede those with larger first names and equal last names. E.g.,

```
zachary abigail
smith harry
jones mary
smith sam
```

becomes

```
jones mary
smith harry
smith sam
zachary abigail
```

When data is sorted, mostly what takes place is that keys are compared and data items are moved around. When measuring the performance of sorting algorithms, sometimes it is only the number of key comparisons that are of interest. However, if the data items are very big, then moving a data item can be a costly operation. In this case we may also be interested in how many data movement operations take place as a function of input size. The number of key comparisons might be small but if the data movements are frequent, then it may not perform as well as an algorithm with fewer data movements and more key comparisons. One

measure of a sort is how much data movement it performs when the initial data set is already sorted. Sorts that do not move the data much in that case are called ***natural sorts***.

The algorithms that we will examine are all internal sorting algorithms that act on data stored in arrays. We begin with selection sort.

1.2 Selection Sort

Selection sort proceeds in stages. At the start of each stage, the array will be partitioned into a two disjoint regions. The upper region will contain those elements already in sorted order. The lower region will contain those elements that are not necessarily sorted. (They might be by chance.) The upper region will also have the property that every element in it is greater than or equal to any element in the unsorted region.

The action performed during each stage is therefore to find the largest key in the unsorted region and put it into the lowest position of the sorted region. Since it was in the unsorted region, it is less than or equal to every key in the sorted region. By placing it in the lowest position in the sorted region it preserves the ordering of that region. In addition, the size of the unsorted region is reduced by one. This implies that after n steps, where n is the number of elements, the unsorted region will be size zero, or that the array has been sorted.

Algorithm

The following function defines selection sort of an array `A` of size `n`. `T` is the underlying element type. The `index_of_maximum_key()` function performs a linear search of its array argument from index 0 to index `last` for the key with maximum value and returns its index. The `swap()` function swaps its arguments.

```
void SelectionSort ( T A[ ], int n)
{
    // Precondition:  A[0..n-1] contains the data to be sorted
    // Postcondition: A[0..n-1] is sorted in ascending order.
    int largest;      // index of largest item in unsorted part of array
    int last;         // index of last item in unsorted part of array
    for ( int last  = n-1; last >= 1; last--) {
        //  Invariant:   A[last+1...n-1] is already sorted and
        //               if j <= last and k > last, A[j] <= A[k]
        largest = index_of_maximum_key(A, last);
        swap( A[largest], A[last]);
    }
}
```

Analysis

During each iteration of the loop, `index_of_maximum_key()` is called once with an array of size `last`+1. To find the maximum element in an array of size n, n-1 comparisons must be performed. (Why not n?) Therefore, the function performs `last` comparisons when its second argument is `last`. Since `last` runs from n-1 down to 1, the total number of key comparisons is the sum of the numbers 1, 2, 3,..., n-1, which is $n(n-1)/2$. Notice that the total number of key comparisons performed by selection sort is the same in all cases – it does not depend on the data.

The `swap()` function performs a constant number of data exchanges. To swap two elements requires three assignments. It is called once in each iteration of the loop. Since there are $(n-1)$ iterations, there are $3(n-1)$ data exchanges caused by `swap()`. As there is no other data movement in the algorithm, the total number of data exchanges is $3(n-1)$. This too is independent of the original state of the data.

Expressing these results in order notation, we can see that selection sort has a worst case, average case, and best case running time that is $O(n^2)$.

1.3 Insertion Sort

Insertion sort also splits an array into a sorted and an unsorted region, but unlike selection sort, it repeatedly picks the lowest index element of the unsorted region and inserts it into the proper position in the sorted region. Hence its name.

Although it makes no difference whether the sorted region is below the unsorted one or not, in this description the sorted region initially consists of the single element A[0] and grows upward. An array of size one is always sorted. Unlike selection sort, the sorted region starts out at non-zero size. The unsorted region is everything from the second array element to the last one.

The algorithm starts at the second position and stops when the element in the last position has been inserted, thereby forcing the size of the unsorted region to zero. Inserting an element into a specific position requires shifting data. The procedure that insertion sort uses is to copy the element to be inserted next into a temporary location, then scan downwards from its initial position while the keys are larger than it. When a key is found that is not larger, the element is inserted above that key and below the one right above it. To make this possible, as the keys are examined during the scan, each element is shifted up one to make room for the inserted element. The algorithm is as follows:

```
void InsertionSort ( T A[ ], int n)
    {
        // Precondition:  A[0..n-1] contains the data to be sorted
        // Postcondition: A[0..n-1] is sorted in ascending order.
        // for each array element A[i], insert it into the sorted region
        for ( int i = 1; i < n; i++) {
            // A[0..i-1] is in sorted order
            tmp = A[i];   // copy
            j = i;
            while ( j > 0 && tmp < A[j-1] ) {
                A[j] = A[j-1];
                j--;
            }
            A[j] = tmp;
            // A[0..i] is now sorted
        }
    }
```

Analysis

In the worst case, the next item to be inserted into its position in the sorted region is always smaller than everything in that region. In other words, the initial array is in reverse sorted order. In this case, the inner while loop makes i key comparisons, since it starts comparing when $j = i$ and stops when $j = 1$. As i runs from 1 to $n - 1$, the total number of comparisons is

$1 + 2 + 3 + \cdots (n - 2) = n(n - 1)/2.$

Each time that the inner while loop iterates, it makes one data exchange. The number of iterations of that loop is the current value of i. So the number of data exchanges within that while loop in the worst case is the same as the number of comparisons, or $n(n-1)/2$. In addition, it makes one data exchange before and after the loop. Therefore the number of data exchanges in the worst case is $n(n-1)/2 + 2(n-1) = n^2 + 3n - 4$.

In short the worst case behavior of insertion sort is $O(n^2)$. It moves much more data than selection sort, since selection sort moves $O(n)$ data items as compared to insertion sort's $O(n^2)$ data exchanges. But insertion sort is natural; if the array is sorted initially, it will only perform $2(n - 1)$ data exchanges, and it is stable as implemented above, since elements with equal keys are never swapped with each other.

1.4 Quicksort

Quicksort is the fastest known sorting algorithm, on average, when implemented *correctly*, which means that it should be written without using recursion, using a stack instead, and putting the larger partition onto the stack each time. But this will not have much meaning to you now. It was invented, or perhaps we should say "discovered" by C.A.R. Hoare in 1960 when he was just 26 years old, and published in the Communications of the ACM in 1961; it revolutionized computer science. He was knighted in 1980.

Quicksort is a divide-and-conquer sorting algorithm. The general idea is to pick an element from the array and to use that element to partition the array into two disjoint sets: one that consists of all elements no larger than the selected element, and the other consisting of elements no smaller than the selected element. The first set will reside in the lower part of the array and the second in the upper part. The selected element will be placed between the two. This process is recursively repeated on each of the two sets so created until the recursion reaches the point at which the set being partitioned is of size one.

1.4.1 The Basic Idea

Let S be the set of elements in the array. S may have elements with equal keys. Quicksort(S) performs the following steps:

1. If the number of elements in S is 0 or 1, then do nothing and return.
2. Pick any element v in S. This element is called the ***pivot***.
3. Partition S–v into two disjoint sets $S1 = x \mid x \in S \wedge x \leq v$ and $S2 = x \mid x \in S \wedge x \geq v$. If an element is equal to v it will be placed into one of $S1$ or $S2$, but not both.
4. Form the sorted array from Quicksort(S1) followed by v followed by Quicksort(S2).

To illustrate, suppose an array contains the elements

```
8 1 4 9 0 3 5 2 7 6
```

Suppose that we pick 6 as the pivot. Then after partitioning the array it will look like

```
1 4 0 3 5 2  6  8 9 7
```

where S1 is the set {1,4,0,3,5,2} and S2 is the set {8,9,7}. Space is inserted just to identify the sets. If we recursively repeat this to these sets S1 and S2 (and assume by an induction argument that it does indeed sort them), then after the recursive calls to quicksort S1 and S2, the array will be in the order

```
0 1 2 3 4 5  6  7 8 9
```

1.4.2 The Partitioning Algorithm

Quicksort can only be efficient if the array can be partitioned in a single pass, i.e., if the partition step takes time proportional to n, the size of the array. Each element should be compared to the pivot once. There are various methods of achieving this. The algorithm described here uses two variables that act like inspectors. These variables start at opposite ends of the array and march towards each other. As they march, they compare the elements they pass on the way. The inspector that starts on the bottom looks for array elements that are greater than or equal to the pivot. If it finds one, it stops on it and waits. The inspector that starts at the top looks for array elements that are less than or equal to the pivot. If it finds one, it stops on it. When each is stopped on an element, the two elements are swapped, because each is in

the wrong set[1]. The inspectors then resume their marches. If, when either one is marching ahead, it passes the other one, then the partitioning has finished because it means that everything that they passed on the way to each other has been placed into the proper set.

```
8 1 4 9 0 3 5 2 7 6        initial array
i               j pivot

8 1 4 9 0 3 5 2 7 6        i found 8; j found  2
i             j

2 1 4 9 0 3 5 8 7 6        2 and 8 are swapped, then advance
  i         j

2 1 4 9 0 3 5 8 7 6        i found 9; j found 5
      i     j

2 1 4 5 0 3 9 8 7 6        9 and 5 are swapped, then advance
        i j

2 1 4 5 0 3 9 8 7 6        i advanced past j
        j   i

2 1 4 5 0 3 6 8 7 9        pivot swapped with A[i]
```

In the above example the array elements were all unique. Soon you will see what happens when they are not all unique.

The first step in the algorithm is to pick a pivot element. For reasons of performance, the pivot should never be the very largest or very smallest element. Once way to make this very unlikely is to pick three random elements, sort them, and make the pivot the middle value. The choose_pivot() function will do that and more. It will also rearrange the array so that the first element has the smallest of the three values, the last element has the pivot, and the second-to-last has the largest element. It will look like this:

```
smallest  ..............  largest pivot
0                         n-2    n-1
```

assuming the array has n elements. The reason for this is that the smallest and largest will act as sentinels, preventing the partitioning step from going out of the array bounds. Since the array has to have at least three elements to do this, quicksort will only be called for arrays of size three or larger.

```
void quicksort( T A[], int left, int right)
{
    // make sure array has at least three elements:
    if ( left + 2 <=  right ) {
        // pick three values, sort them, and put the pivot into A[right],
        // the smallest of the three into A[left] and the largest into A[right-1].
        // assume that the function choose_pivot() does all of this.
        T pivot  = choose_pivot(A, left, right);

        // A[left] <= pivot <= A[right-1] and pivot is A[right]
```

[1]The inspectors stop on elements equal to the pivot. They are not in the wrong set. It will be seen that this prevents poor performance. That will be explained shortly.

```
            // now partition
            int i = left;
            int j = right-1;
            bool done = false;

            while ( ! done ) {
                while( A[++i] < pivot ) { }   // advance i
                while ( pivot < A[--j]) { }   // advance j

                // A[i] >= pivot and A[j] <= pivot
                if ( i < j )
                    swap( A[i], A[j] );
                else
                    done = true;
            }
            // now j <= i and A[i] >= pivot and A[j] <= pivot
            // so we can swap the pivot with A[i]
            swap( A[i], A[right] );

            // Now the pivot is between the two sets and in A[i]
            // quicksort the left set:
            quicksort(A, left, i-1);

            // quicksort the right set:
            quicksort(A,i+1, right);
        }
        else {
            // A has just two elements
            if ( A[left] > A[right] )
                swap(A[left], A[right] );
        }
    }
```

1.4.3 Analysis

The outer loop of quicksort iterates until i and j pass each other. They start at opposite ends of the array and advance towards each other with each key comparison. Therefore, roughly n key comparisons take place for an array of size n. The function is called twice recursively, once for each set. Suppose that the pivot is chosen very badly each time and that it is always the largest element in the set. Then the upper set will be empty and the lower set will be roughly size n-1. This means that the recursive call to the lower set will make n-1 key comparisons and that to the upper will not even execute. This implies that the recursion will descend down the lower sets, as they get smaller by one each time, from n, to n-1 to n-2 to n-3 and so on until the last call has an array of size 3, when it stops. The total number of key comparisons is therefore roughly

$$3 + 4 + 5 + \cdots (n-2) + (n-1) + n \approx n(n-1)/2 = O(n^2).$$

Thus, the worst case of quicksort is $O(n^2)$. That is why it is important to choose the pivot properly. A poor choice of pivot will cause one set to be very small and the other large. The result will be that the algorithm achieves its $O(n^2)$ worst case behavior. If the pivot is smaller than all other keys or larger than all other keys this will happen. Quicksort is best when the pivots divide the array roughly in half every time. Since the array is divided in half each time, the depth of recursion is $\log_2 n$, when the array is sorted. It can be proved that each level of the recursion uses $O(n)$ key comparisons, so that at best it takes $O(n \log n)$ steps. It can be proved that the average case is $O(n \log n)$ as well. In fact, it can be proved that the average case requires about 1.39 times the number of comparisons as the best case.

1.4.4 Equal Keys

Consider the array

```
1 1 1 1 1 1 1 1 1 1
```

in which all keys are identical. A good implementation of quicksort should take $O(n \log n)$ steps in this case. If you follow the algorithm described above, you will see that it does. Because i and j stop on keys equal to the pivot, they each progress one element at a time and swap, eventually meeting in the center:

```
1 1 1 1 1 1 1 1 1 1        initial array
i               j pivot

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
  i           j

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
    i       j

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
      i   j

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
        i                  they are now on the same element
        j                  and the partition stops
```

Now the two sets are equal size, and the recursion is balanced. This leads to the best case behavior.

To be efficient, quicksort should be implemented non-recursively. The idea is to use a stack to store the parameters of the recursive call. Instead of making the two recursive calls, the parameters of one of the recursive calls are pushed on the stack and the other call is executed within the while loop again. The stack is popped to retrieve the parameters and simulate the call. To avoid the stack growing too large, the call for the larger partition is pushed onto the stack and the one for the smaller is executed immediately. You can find the details elsewhere.

1.5 Merge Sort

Merge sort is another example of a divide-and-conquer, inherently recursive, sorting algorithm. The basic idea is very simple: divide the array in half, sort each half separately, and then merge the sorted halves into a single array again. Of course each half is sorted using the merge sort algorithm, which is why it is recursive. And of course, like any recursive algorithm, it has a base case, which is simply that when the array to be sorted has just two elements, they are simply compared and swapped if necessary. The "merge" part of the algorithm does the real work; its job is to compare the elements in each half of the array, moving the smaller of the two it compares into the "output" array. Merge sorting is easiest when there is a spare array equal in size to the original array.

For simplicity, suppose that the array to be sorted has $2^k = n$ elements.