CSc 3102: Supplementary Notes

Recursive Sorts: Quick and Merge Sort Algorithms

- The Sorting Problem
- Merge Sort
- Quick Sort
- Comparison of Sorting Algorithms

1 The Sorting Problem

In this course, we will look at the sorting problem as one of reordering the elements in a data structure so that they fall in some defined sequence. You should already be familiar with basic sorting algorithms. Here we focus on two important divide-and-conquer algorithms, quick sort and merge sort. These algorithms have very elegant recursive formulations and are highly efficient. The comparison of data items in the data structure is based on a value called the key. A sorting algorithm is *stable* if duplicate keys retain their relative order. An *in-place* sorting algorithm is one in which the sorted items occupy the same storage as the original ones. These algorithms may use $O(n)$ additional memory for bookkeeping, but at most a constant number of items are kept in auxiliary memory at any time.

Formal Definition:
The sort operation may be defined in terms of an initial array, S, of N items and a final array, S', as follows.

1. $S_i \leq S_{i+1}$, $0 < i < N$. Here $\leq$ means precede in some ordering.
2. S' is a permutation of S.

COPYRIGHT

2 Merge Sort

The merge sort algorithm works as follows:

1. Check to see if the array is empty or has only one element. If so, it must already be sorted, and the function can return without doing any work. This condition defines the simple case for the recursion.
2. Divide the array into two new subarrays, each of which is half size of the original.
3. Sort each of the subarrays recursively.
4. Merge the two subarrays back into the original one.

2.1 Merge Sort Code

```
void sort(Item[] array, int F, int L)
{
   int mid;

   if (F >= L) {return; }
   mid = (F + L)/2;
   sort(array, F, mid);
   sort(array, mid+1, L);
   Merge(array, F, mid, L)
}
```

We will show that merge sort is $\Theta(n \lg n)$ in class by solving a recurrence equation. It has a better worst-case performance than selection and insertion sorts. We will also use a bit of hand-waiving to show the same asymptotic bound. Merge sort is asymptotically optimal.

COPYRIGHT

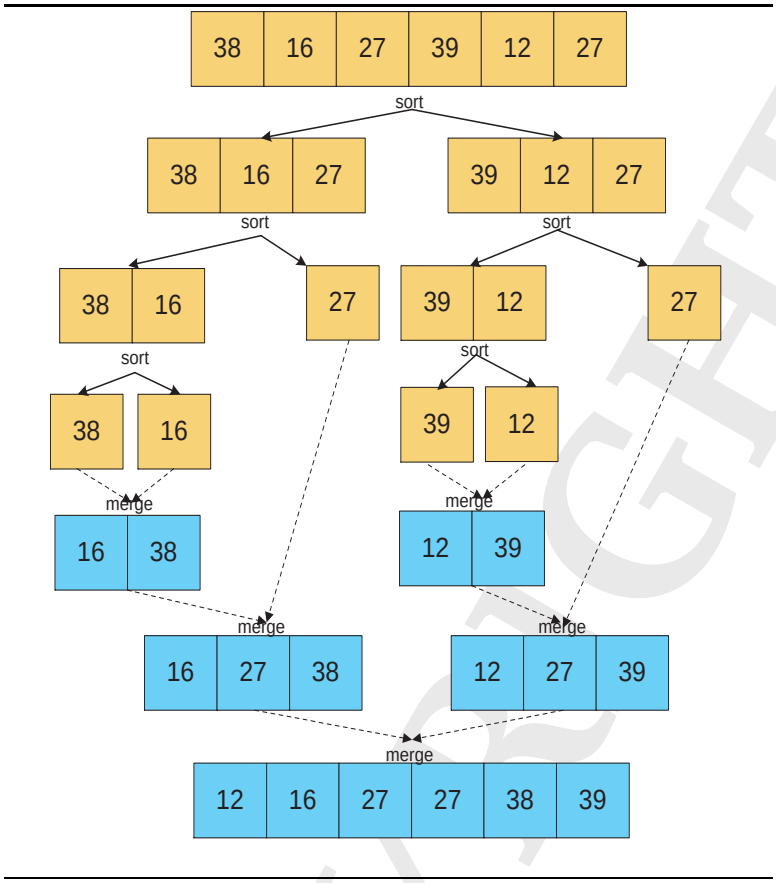

Figure 1: A merge sort of an array of six integers

3 Quick Sort

Although mergesort is an extremely efficient algorithm with respect to time, it does have one drawback: To perform the step

```
merge sorted halves A[First..Mid] and A[Mid+1...Last]
```

the algorithm requires an auxiliary array whose size equals the size of the original array. Quicksort has no such space requirement. Quicksort partitions an array into two regions - one with items that are less than or equal to the pivot and the other with those that are greater than or equal to the pivot.

COPYRIGHT

3.1 Quick Sort

```
void sort(Item[] array, int from, int to)
{
   if (from >= to) {return; }
   p <- partition(array. from, to)
   sort(array, from,p);
   sort(array, p+1,to);
}
```

Quick sort partitions the array into two regions about an array element called the pivot such that all the elements in the first region are less than or equals to the pivot and the elements in the second region are greater than or equal to the pivot. Here is a hand-trace of the algorithm.

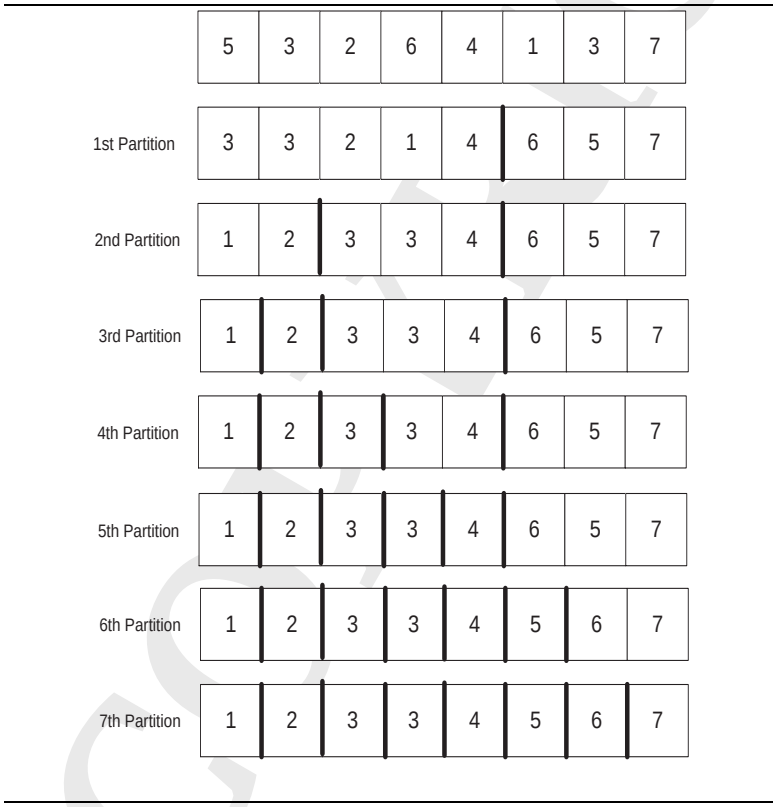

Figure 2: A quick sort of an array of eight integers

COPYRIGHT

There are several algorithms to partition an array, each with its own merit and pitfalls. Here is the pseuoocode for the partition algorithm that we will use in this course:

```
integer partition(item[] array, from, to)
{
   item pivot <- array[from]
   integer i, j
   i <- from - 1
   j <- to + 1
   while (i < j)
      i <- i + 1
      while (array[i] < pivot)
         i<-i + 1
      endWhile
      j <- j - 1
      while (array[j] > pivot)
         j<-j - 1
      endWhile
      if (i < j)
         swap(array,i,j)
      endIf
   endWhile
   return j
}
```

We will show that quick sort is $O(n \lg n)$ in the best-case and $O(n^2)$ in the worst-case. We will derive these by solving recurrence equations. In spite of its poor worst-case performance, when the pivot is randomly chosen during each stage of the algorithm, on average like Merge Sort, it is $O(n \lg n)$.

COPYRIGHT

4 Comparison of Sorting Algorithms

Finally, let's end our discussion on recursive sort algorithms with a table comparing various common sort algorithms, some of which we have not and will not study in this course.

Algorithm	Worst-case	Average-case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Merge sort	$n \lg n$	$n \lg n$
Quick sort	n^2	$n \lg n$
Radix sort	n	n
Treesort	n^2	$n \lg n$
Heapsort	$n \log n$	$n \lg n$

Figure 3: Approximate growth rates of time required for eight sorting algorithms

Both merge sort and quick sort are *divide-and-conquer* algorithms? Why? Why is binary search not a D&C algorithm?

Exercise 1. Suppose the array $[5, 3, 2, 5, 4]$ is used as the initial argument to the merge sort algorithm shown in the pseudo-code description in section 2.1.

1. Draw a diagram similar to the one in Figure 1 showing the action of the merge sort algorithm on the array.
2. Using the notation **sort(**$[\cdots]$**)** and **merge(**$[\cdots]$,$[\cdots]$**)**, where $\cdots$ denotes elements of the arrays listed in the order in which they are stored in the arrays, list the calls to these subroutines in the order that they are made.
3. How many calls are made to sort? How many calls are made to merge?
4. For each call to merge list all (x_i, x_j) pairs of elements of the arrays that are compared, where x_i is an element in the array used as the left argument and x_j is an element in the array used as the right argument.

COPYRIGHT