Neil Patrick Del Gallego 10/15/14
11482850
XALGOCOM

Merge Sort Report Summary

Introduction

This is the summary of the report regarding the Merge Sort algorithm which follows the Divide and Conquer algorithm paradigm.

Overview of the Divide and Conquer Paradigm

Divide and Conquer is an algorithm design technique wherein a given problem is divided into smaller similar sub problems until the sub problems become manageable enough to be solved directly. The results of all the sub problems are combined to formulate a solution to the original problem.

Implementations and Limitations

- Algorithms designed based from Divide and Conquer are implemented as recursive procedures that has a base case simple enough to solve a sub problem. A simpler base case usually results into a more elegant solution since there are fewer cases to consider.

- Divide and Conquer algorithms may have repeated sub problems which may otherwise be wasted recomputing them since Divide and Conquer does not have a lookup table to store previous computations. Dynamic Programming attempts to solve this situation.

The Merge Sort Algorithm

Overview

Given an unsorted list of size N, the merge sort algorithm attempts to return a sorted list of size N after execution. Figure 1 shows how the merge sort algorithm is performed given a sample unsorted list. The algorithm follows the Divide and Conquer paradigm through the following guidelines:

- **Divide –** Create two sublists of size N/2 repeatedly until it becomes manageable. A "manageable" case for the merge sort algorithm is a list of size 1, which means that a list of size 1 is already sorted.
- **Conquer –** Compare two sublists and sort them using the *merge()* sort procedure.
- **Combine** – Merge the two sublists to produce a sorted list of size N.

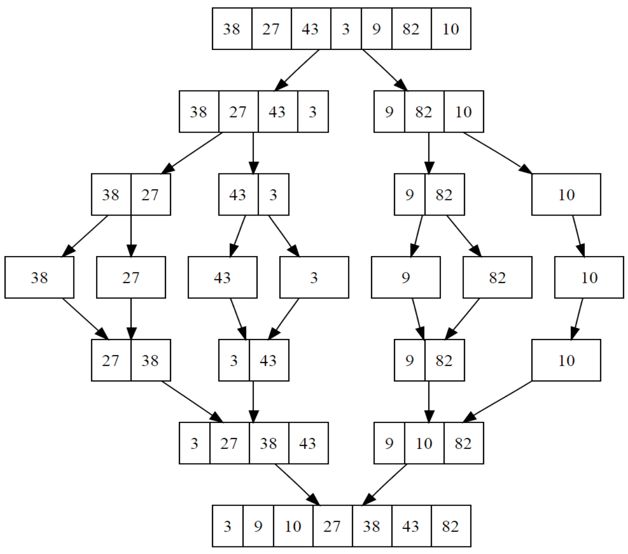

Figure 1: Illustration for Merge Sort Procedure

Pseudocode

Listing 1 shows the general recursive procedure of merge sort wherein an array is used as part of input. The resulting output is the same array reference but modified such that the elements inside it are already sorted. The starting call for the merge sort procedure should be *mergeSort(someArray, 1, someArray.length).* The *merge()* function attempts to merge the two sublists to produce a larger sorted list.

```
function mergeSort(int[] arrayN, int beforeMiddle, int afterMiddle) {
      if(beforeMiddle < afterMiddle) {
            int middle = Math.floor((beforeMiddle + afterMiddle) / 2);
            mergeSort(arrayN, beforeMiddle, middle);
            mergeSort(arrayN, middle + 1, afterMiddle);
            merge(arrayN, beforeMiddle, middle, afterMiddle);
      }
}
```

Listing 1: Merge sort general recursive procedure

The merge procedure

Listing 2 shows the *merge()* procedure which modifies a certain input array by dividing it into two smaller subarrays and comparing each elements inside. The pseudocode code given introduces a 'sentinel' card which is an arbitrary symbol that is added to the end of the sub lists. If the current index of a certain sub array reaches the sentinel, then it means that the rest of the values of the its comparing sub array are automatically put at the end of the merged list (since this means that values being compared to the sentinel are guaranteed to be of lesser value).

Figure 2 illustrates the *merge()* procedure with emphasis on how sentinel cards are used. Given two subarrays, which came from a larger sub array, the *merge()* procedure adds the sentinel cards at the end of the the sub arrays for easier comparison in the combine procedure. The numbers shown at the bottom of the sub arrays indicate the 'steps' on how the pointer index of each sub array is moved after each resulting comparison. So at the start of the combine procedure, both indices start at the first element on both of the sub arrays. Therefore, comparing 3 and 4, this results in 3 being 'combined' into the larger sub array. Afterwards, the index of the left sub array is moved towards step 1. Comparing 5 and 4 results in the index of the right sub array to move towards step 2. Comparing 5 and 7 moves the left sub array index to step 3 and so forth.

Notice that upon reaching the sentinel, the rest of the elements found in the right sub array (in this case, only 7) is put towards the end of the larger list.

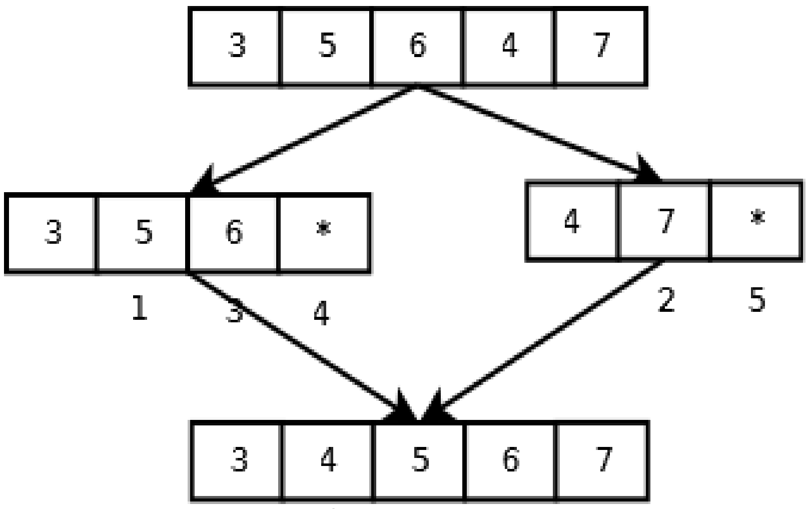

Figure 2: Illustration of the merge procedure with sentinels

```
//attempts to modify a sorted arrayN by dividing the initial arrayN into
two smaller arrays
//and comparing the elements inside.
function merge(int[] arrayN, int beforeMiddle, int middle, int
afterMiddle) {
	int leftNSize = middle - beforeMiddle + 1;
	int rightNSize = afterMiddle - beforeMiddle;

	//initialize left and right sub-array with additional index to
store the 'sentinel'
	int[] leftN = new int[leftNSize + 1];
	int[] rightN = new int[rightNSize + 1];

	for(int i = 0 : leftNSize) {
		leftN[i] = arrayN[beforeMiddle + i];
	}

	for(int j = 0 : rightNSize) {
		rightN = arrayN[middle + j];
	}

	int sentinel = Integer.MAX; //let this be a special sentinel value
	leftN[leftNSize] = sentinel;
	rightN[rightNSize] = sentinel;

	int leftIndex = 0;
	int rightIndex = 0;

	//this is the 'combine' process
	for(int k = beforeMiddle : afterMiddle) {
		if(leftN[leftIndex] <= rightN[rightIndex]) {
			arrayN[k] = leftN[leftIndex];
			leftIndex++;
		}
		else {
			arrayN[k] = rightN[rightIndex];
			rightIndex++;
		}

	}
}
```

Listing 2: The merge procedure

Sample Problems

1. A 4-way mergesort is a modified mergesort algorithm wherein a given list of size N , which instead of dividing the list into two, it is evenly divided into 4 sublists. Give the recurrence relation and it's Big-Oh.
   T(n) = { 1 n = 1
   { 4 T(N/4) + 2N otherwise

2. A k-way merge operation accepts k sorted arrays, each with n elements. The *merge()* procedure is modified such that it produces a sorted array of k*n elements. Determine the Big-Oh of the modified *merge()* procedure.
   =(n + n) + (2n + n) + (3n + n) + . . . + ((k − 1)n + n))
   = 2n + 3n + 4n + . . . + kn (2)
   = 0(k^2 n).

References

Cormen, T. H., & Cormen, T. H. (2009). Introduction to algorithms 3rd edition. Cambridge, Mass: MIT Press.

Martin, D. (2007). Sorting Algorithms - Merge Sort. Electronic website: http://www.sorting-algorithms.com/merge-sort

Zeh, N (2014). CSCI 3110: Design and Analysis of Algorithms. Online Course Resources. Electronic website: https://web.cs.dal.ca/~nzeh/Teaching/3110/