CH08-320143: Assignment #6

Harit Krishan
h.krishan@jacobs-university.de

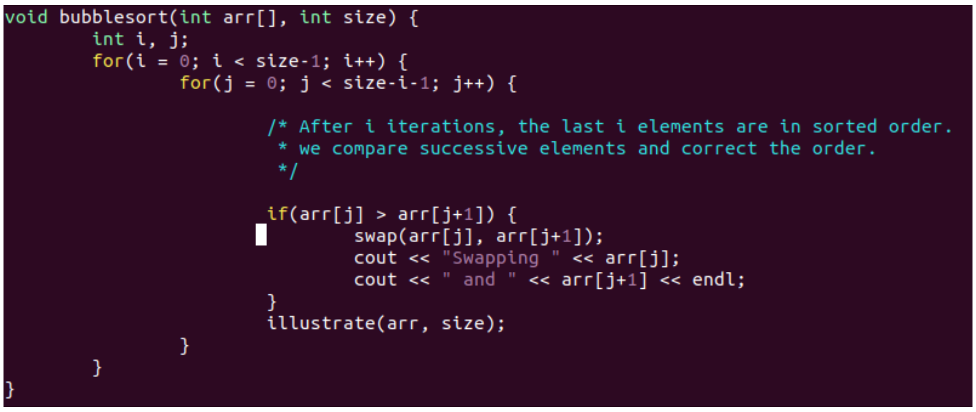
Jacobs University Bremen — March 16, 2020

1 Bubble Sort || Stable and Adaptive Sorting

Bubble Sort is a sorting algorithm that works by repeatedly iterating through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. This is repeated until no swaps are needed, which indicates that the list is sorted.

1.1 Implementation

We can implement this algorithm in C++. This is illustrated in Figure 1.



```
void bubblesort(int arr[], int size) {
        int i, j;
        for(i = 0; i < size-1; i++) {
                for(j = 0; j < size-i-1; j++) {

                        /* After i iterations, the last i elements are in sorted order.
                         * we compare successive elements and correct the order.
                         */

                        if(arr[j] > arr[j+1]) {
                                swap(arr[j], arr[j+1]);
                                cout << "Swapping " << arr[j];
                                cout << " and " << arr[j+1] << endl;
                        }
                        illustrate(arr, size);
                }
        }
}
```

Figure 1: Implementation of Bubblesort in C++

1.2 Asymptotic Analaysis

When we apply asymptotic analysis, we must be clear on the different cases - best, average and worst. The best case scenario for this algorithm would be if the input array was already sorted. In this case, the algorithm in Figure 1 will run $n-1$ times. Therefore, we get the following expression -

$$O(n-1) \quad \therefore \quad O(n)$$

The worst case scenario for this algorithm would be if the input array is in reverse order. This implies every element needs swapping. The run complexity calculation is as follows -

$$n + (n-1) + (n-2) + (n-3) + ... + 1 = \frac{n(n+1)}{2}$$
$$\therefore \frac{n^2+1}{2} = n^2 + \frac{1}{2}$$
$$\therefore O(n^2)$$

The average case holds the same explanation and logic as the worst case.

1.3 Stability

Stable sorting algorithms maintain the relative order of records with equal keys. Thus, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

Insertion Sort : This is a stable algorithm. In insertion sort, we have an exterior loop condition which prevents the algorithm from running over the start of the input array. A secondary condition is used to insert the selected item into the correct spot. These conditions ensure sort stability by checking both keys.

Merge Sort : This algorithm is stable. While merging the different halves of the input array, there is a condition which favors swapping the vales from the left side if they are equal.

Heap Sort : This algorithm is not stable. The final permutation of the input sequence comes from removing the item from the heap in a size order. This implies that any sense of order is lost through the heap creation, hence unstable.

Bubble Sort : This algorithm is stable. In this algorithm, the condition for the swap favors the left element over the right due to the 'less than or equal to' operator. Because of this, the exact order is conserved, hence stable.

1.4 Adaptability

A sorting algorithm is adaptive, if it takes advantage of existing order in its input. Thus, it benefits from the pre-sortedness in the input sequence and sorts faster.

Insertion Sort : This algorithm is adaptive. In the best case, the algorithm runs n times where n is the number of elements in the input array. In the worst and average case, the complexity becomes $O(n^2)$.

Merge Sort : This algorithm is non-adaptive. For merge sort, the time complexity is $\Theta(nlog(n))$ since it breaks down the array to compare at its smallest level, regardless of input order.

Heap Sort : This algorithm is non-adaptive. The second loop in this algorithm has a time complexity of $O(nlog(n))$. When the heap is created, the numeric order is lost.

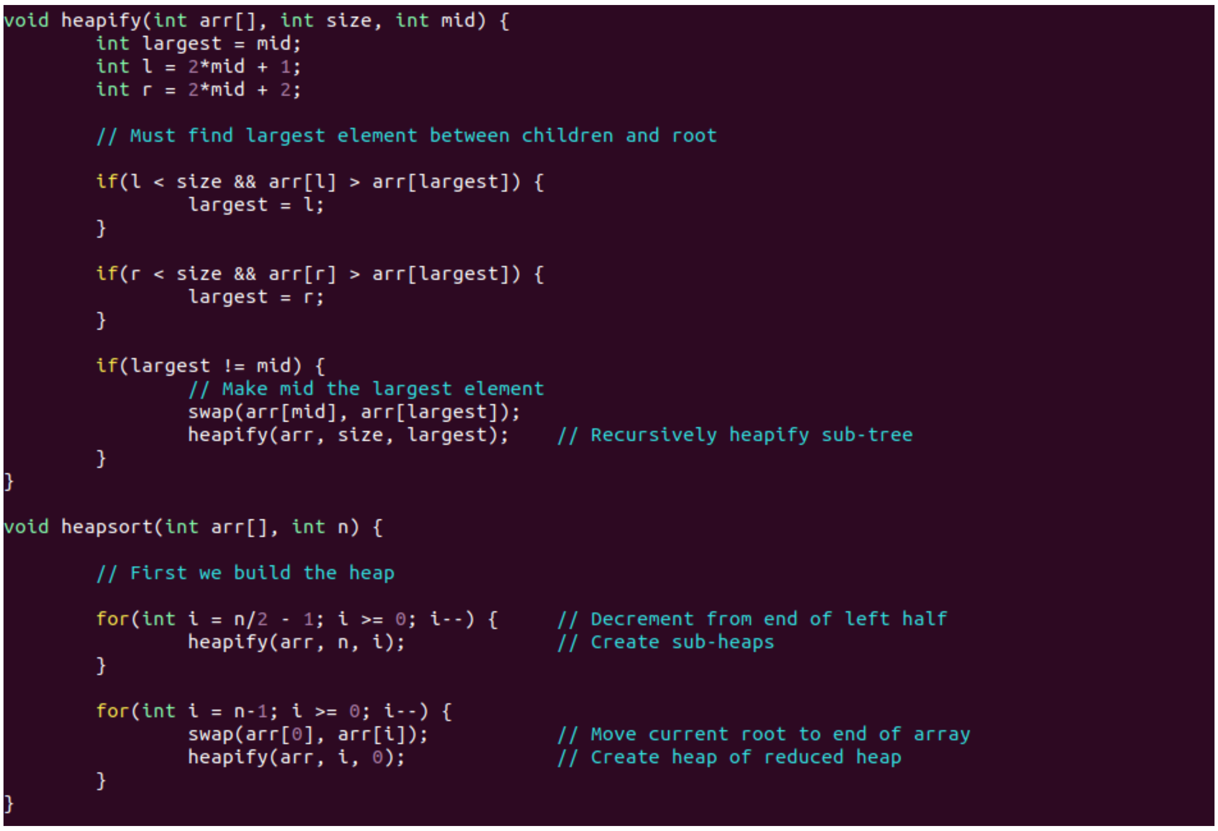
Bubble Sort : This algorithm is adaptive. When the input array is sorted, no swaps will be made, therefore the inner loops will not run. The time complexity for best case is $O(n)$. The average time complexity is $O(n^2)$.

2 Heap Sort

Heap sort is a comparison-based sortin technique based on the binary heap data structure. It is similar to selection sort where the maximum element is found and placed at the end.

2.1 Implementation

This algorithm can be implemented in C++. The algorithm is broken down into two algorithms - heapify and heapsort. Heapify is used to create a heap of the sub-sequences. Heapsort executes the comparison statements and swaps the elements. The implementation is illustrated in Figure 2.

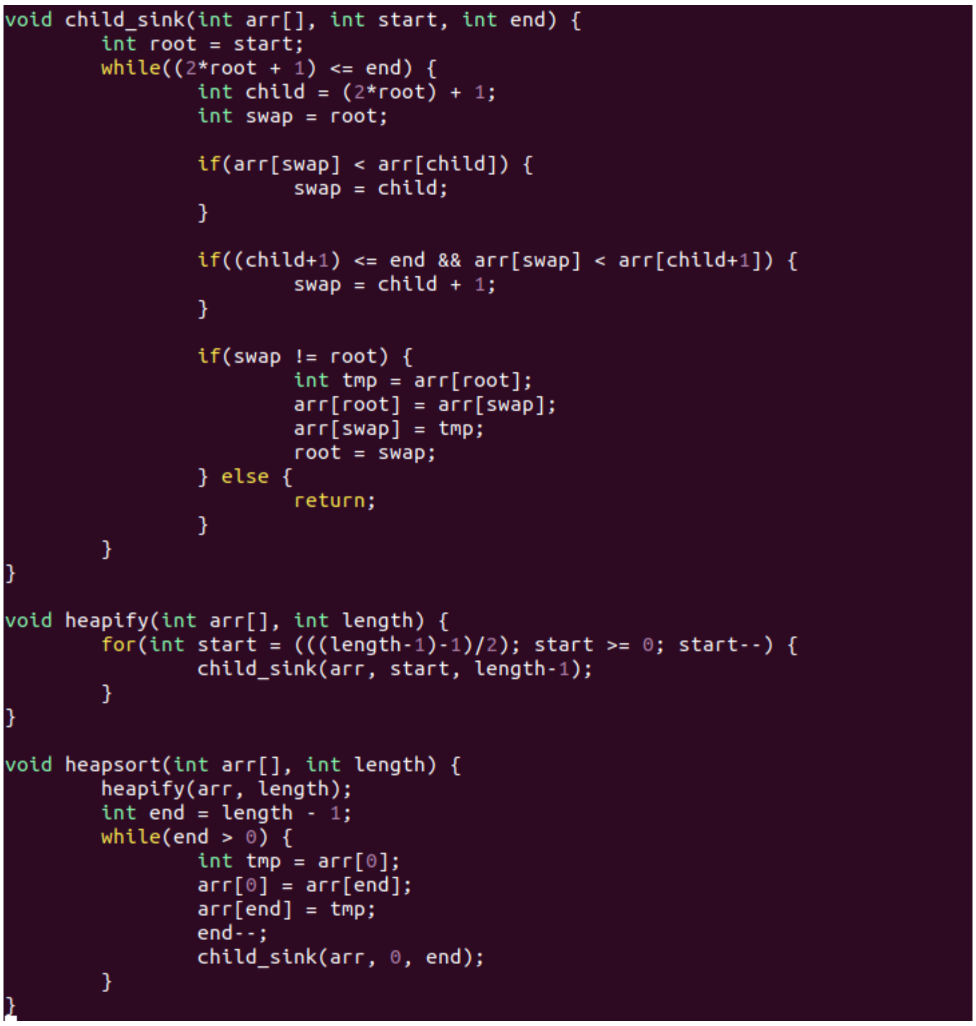


```cpp
void heapify(int arr[], int size, int mid) {
        int largest = mid;
        int l = 2*mid + 1;
        int r = 2*mid + 2;

        // Must find largest element between children and root

        if(l < size && arr[l] > arr[largest]) {
                largest = l;
        }

        if(r < size && arr[r] > arr[largest]) {
                largest = r;
        }

        if(largest != mid) {
                // Make mid the largest element
                swap(arr[mid], arr[largest]);
                heapify(arr, size, largest);    // Recursively heapify sub-tree
        }
}

void heapsort(int arr[], int n) {

        // First we build the heap

        for(int i = n/2 - 1; i >= 0; i--) {     // Decrement from end of left half
                heapify(arr, n, i);             // Create sub-heaps
        }

        for(int i = n-1; i >= 0; i--) {
                swap(arr[0], arr[i]);           // Move current root to end of array
                heapify(arr, i, 0);             // Create heap of reduced heap
        }
}
```

Figure 2: Heap Sort implementation in C++

2.2 Variant

The algorithm from Figure 2 can be modified as per the mentioned requirements. A snapshot of this modified implementation is illustrated in Figure 3. This is also implemented in C++.



```
void child_sink(int arr[], int start, int end) {
        int root = start;
        while((2*root + 1) <= end) {
                int child = (2*root) + 1;
                int swap = root;

                if(arr[swap] < arr[child]) {
                        swap = child;
                }

                if((child+1) <= end && arr[swap] < arr[child+1]) {
                        swap = child + 1;
                }

                if(swap != root) {
                        int tmp = arr[root];
                        arr[root] = arr[swap];
                        arr[swap] = tmp;
                        root = swap;
                } else {
                        return;
                }
        }
}

void heapify(int arr[], int length) {
        for(int start = (((length-1)-1)/2); start >= 0; start--) {
                child_sink(arr, start, length-1);
        }
}

void heapsort(int arr[], int length) {
        heapify(arr, length);
        int end = length - 1;
        while(end > 0) {
                int tmp = arr[0];
                arr[0] = arr[end];
                arr[end] = tmp;
                end--;
                child_sink(arr, 0, end);
        }
}
```

Figure 3: Heap Sort Variant implementation in C++

2.3 Performance (BONUS)

When we come down to comparing the two variants of Heap Sort, we get the following approximate table -

$sizeof(n)$	Heap Sort	Heap Sort Variant
1000	0.00022	0.000187
10000	0.00197	0.001826
100000	0.02149	0.021038
1000000	0.24785	0.274699

As evident from the table above, as the size of the input array increases, the run time of the algorithm also increases. For smaller input sizes, the algorithm tends to run quicker. The algorithm has been timed using the <ctime> library from C.

All necessary files for this assignment are included in the root directory of this submission.