Heap Sorts

110191019 YU-TING CHUNG

Introduction

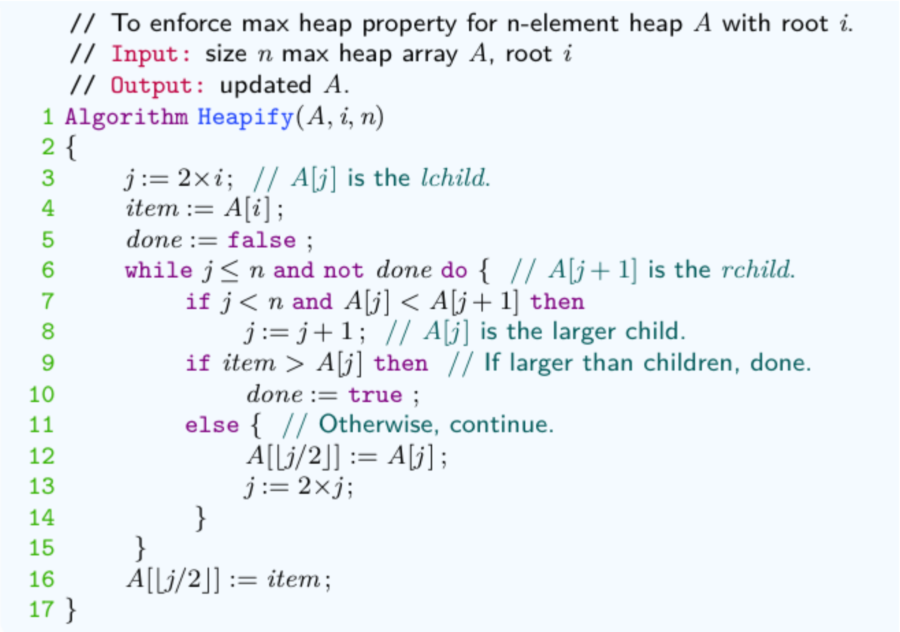
Heap sort is a highly efficient comparison-based sorting algorithm that utilizes a binary heap data structure to sort elements. It is classified as a selection sort algorithm and works by repeatedly extracting the largest (or smallest) element from a heap and placing it at the end of the array.

This assignment requires the implementation of two versions of heap sort: the standard heap sort and the bottom-up heap sort. Additionally, to evaluate the impact of comparison operations on execution time, two versions of each algorithm will be implemented—one using the standard strcmp function and another using compare function. The performance of these implementations will be analyzed by using multiple input datasets, and execution times will be compared to assess efficiency.

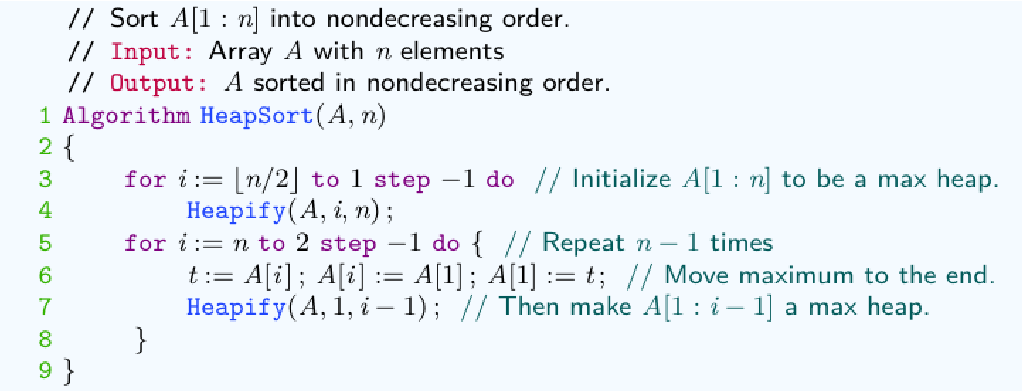
Approach

- Heap Sort

  1. Algorithm:



```
// To enforce max heap property for n-element heap $A$ with root $i$.
// Input: size $n$ max heap array $A$, root $i$
// Output: updated $A$.
Algorithm Heapify($A, i, n$)
{
    $j := 2 \times i$;  // $A[j]$ is the $lchild$.
    $item := A[i]$;
    $done :=$ false ;
    while $j \leq n$ and not $done$ do {  // $A[j+1]$ is the $rchild$.
        if $j < n$ and $A[j] < A[j+1]$ then
            $j := j + 1$;  // $A[j]$ is the larger child.
        if $item > A[j]$ then  // If larger than children, done.
            $done :=$ true ;
        else {  // Otherwise, continue.
            $A[\lfloor j/2 \rfloor] := A[j]$;
            $j := 2 \times j$;
        }
    }
    $A[\lfloor j/2 \rfloor] := item$;
}
```



```
// Sort $A[1:n]$ into nondecreasing order.
// Input: Array $A$ with $n$ elements
// Output: $A$ sorted in nondecreasing order.
Algorithm HeapSort($A, n$)
{
    for $i := \lfloor n/2 \rfloor$ to 1 step $-1$ do  // Initialize $A[1:n]$ to be a max heap.
        Heapify($A, i, n$);
    for $i := n$ to 2 step $-1$ do {  // Repeat $n-1$ times
        $t := A[i]$; $A[i] := A[1]$; $A[1] := t$;  // Move maximum to the end.
        Heapify($A, 1, i-1$);  // Then make $A[1:i-1]$ a max heap.
    }
}
```

The algorithm operates by first transforming the input array into a max heap to ensure that the largest element is always at the root. Then, the largest element is swapped with the last element and removed from the heap, while the remaining elements are restructured to maintain the heap property. This process is repeated until all elements are sorted.

2. Proof of correctness:

At the start of the algorithm, the input array is first transformed into a max heap. This ensures that the largest element is always at the root of the heap. In each step of the sorting process, the root element which is the largest is swapped with the last unsorted element in the array, and the heap size is reduced by one. The heap property is then restored by performing the Heapify operation on the new root.

Since the Heapify function ensures that the heap property is maintained at every step, and since each extraction step correctly places the next largest element in its correctly sorted position, the array is fully sorted when the heap size is reduced to one.

Given that every element is correctly placed in each iteration, heap sort always produces the correct sorted order. Hence, proved.

3. Time and space complexities:

Time complexity:

Heap Sort consists of two main steps:

(1) Building the heap: O(n log n)

Heap Sort uses the top-down approach to build the heap by inserting each element. Each insertion takes O(log n) time, and with n elements, the total time complexity is O(n log n).

(2) Heapify and extraction: O(n log n)

After extracting the root, the heap needs to be adjusted using heapify, which takes O(log n) time. The following table explains why each data entry can move up to ($\log_2$ n) levels.

Number of layers	Number of nodes/ Number of data points	Number of layers
1	$2^0 = 1$	$\log_2 2$
2	$2^0+2^1 = 3$	$\log_2 4$
3	$2^0+2^1+2^2 = 7$	$\log_2 8$
...	...	...
k	$2^0+2^1+2^2+\ldots +2^{k-1}=2^k-1$	$\log_2 2^k$
	n	$\log_2 n$

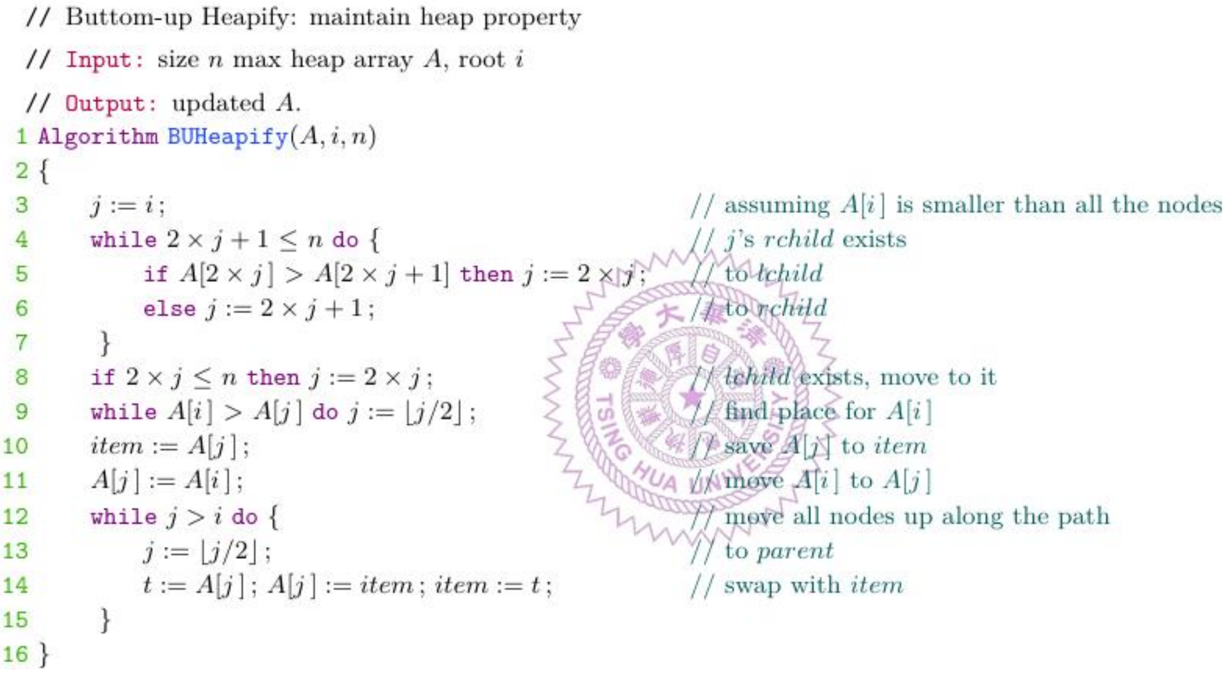
Since this process is repeated n times, the total time complexity is O(n log n).

Overall Time Complexity: O(n log n)+ O(n log n)= O(n log n)

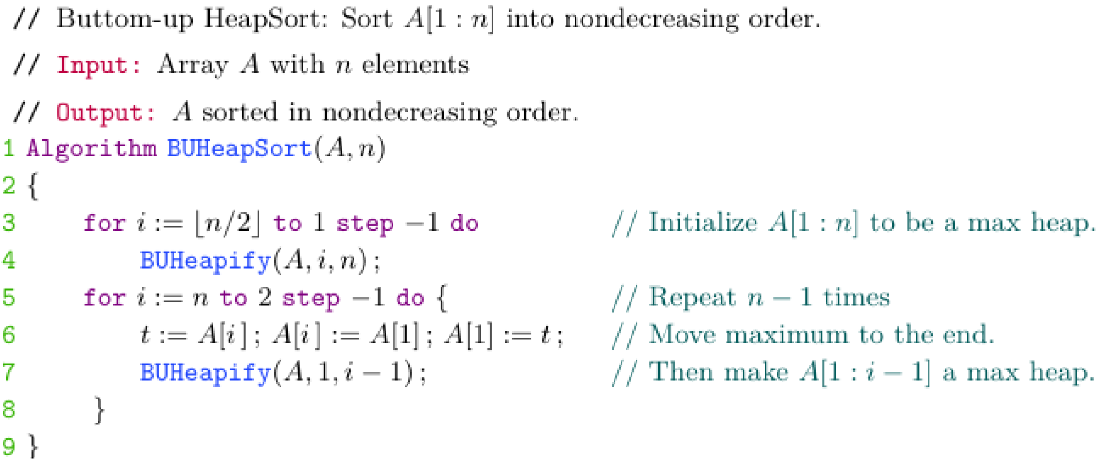
Space complexity: $\log_2 n+5$ ➔ O(log n)

- Bottom-up heap sort

1. Algorithm:



```
// Buttom-up Heapify: maintain heap property
// Input: size $n$ max heap array $A$, root $i$
// Output: updated $A$.
Algorithm BUHeapify($A, i, n$)
{
    $j := i$;                                          // assuming $A[i]$ is smaller than all the nodes
    while $2 \times j + 1 \le n$ do {                  // $j$'s $rchild$ exists
        if $A[2 \times j] > A[2 \times j + 1]$ then $j := 2 \times j$;   // to $lchild$
        else $j := 2 \times j + 1$;                    // to $rchild$
    }
    if $2 \times j \le n$ then $j := 2 \times j$;      // $lchild$ exists, move to it
    while $A[i] > A[j]$ do $j := \lfloor j/2 \rfloor$; // find place for $A[i]$
    $item := A[j]$;                                    // save $A[j]$ to $item$
    $A[j] := A[i]$;                                    // move $A[i]$ to $A[j]$
    while $j > i$ do {                                 // move all nodes up along the path
        $j := \lfloor j/2 \rfloor$;                    // to $parent$
        $t := A[j]$; $A[j] := item$; $item := t$;      // swap with $item$
    }
}
```



```
// Buttom-up HeapSort: Sort $A[1:n]$ into nondecreasing order.
// Input: Array $A$ with $n$ elements
// Output: $A$ sorted in nondecreasing order.
Algorithm BUHeapSort($A, n$)
{
    for $i := \lfloor n/2 \rfloor$ to 1 step $-1$ do      // Initialize $A[1:n]$ to be a max heap.
        BUHeapify($A, i, n$);
    for $i := n$ to 2 step $-1$ do {                      // Repeat $n-1$ times
        $t := A[i]$; $A[i] := A[1]$; $A[1] := t$;         // Move maximum to the end.
        BUHeapify($A, 1, i-1$);                           // Then make $A[1:i-1]$ a max heap.
    }
}
```

Bottom-up heap sort is a variation of heap sort that constructs the heap more efficiently by starting the process from the lowest levels of the tree rather than the root. This reduces the number of comparisons and swaps, and it leads to improved performance in certain cases.

2. Proof of correctness:

At the start of the algorithm, the input array is first transformed into a max heap using a bottom-up approach. This process ensures that the heap property is established efficiently by starting heapification from the lowest levels of the tree and moving upward.

In each step of the sorting phase, the largest element (located at the root) is swapped with the last unsorted element, and the heap size is reduced by one. The bottom-up heapification process is then applied to restore the heap property efficiently.

Since the bottom-up Heapify function (BUHeapify) ensures that the heap property is maintained while minimizing the number of comparisons, and since each extraction step correctly places the next largest element in its correctly sorted position, the array is fully sorted when the heap size is reduced to one.

As every element is correctly placed in each iteration, the bottom-up heap sort algorithm always produces the correct sorted order.

Hence, proved.

3. Time and space complexities:

   Time complexity:

   Bottom-up Heap Sort consists of two main steps:

   (1) Building the heap (O(n)):

In the bottom-up approach, we start from the last non-leaf node and move upward. The heapify operation takes O(log k) time, where k is the height of the node in the tree.

Since the bottom-up method processes nodes with decreasing height, the total time to build the heap is O(n), which is an improvement over the O(n log n) time in the top-down approach.

(2) Heapify and extraction (O(n log n)):

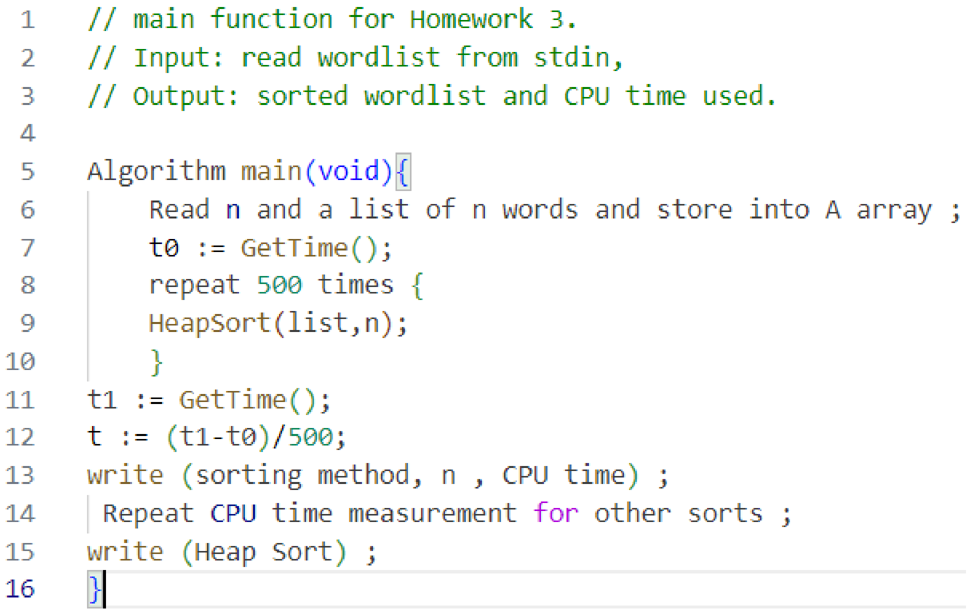
After the heap is built, the heap sort proceeds with extracting the root and heapifying the remaining elements. As with the top-down approach, each extraction and heapify operation takes O(log n) time, and we perform this n times.

Overall Time Complexity: O(log n)+ O(n log n)= O(n log n)

Space complexity: $\log_2 n+4$ ➔ O(log n)

- Main function

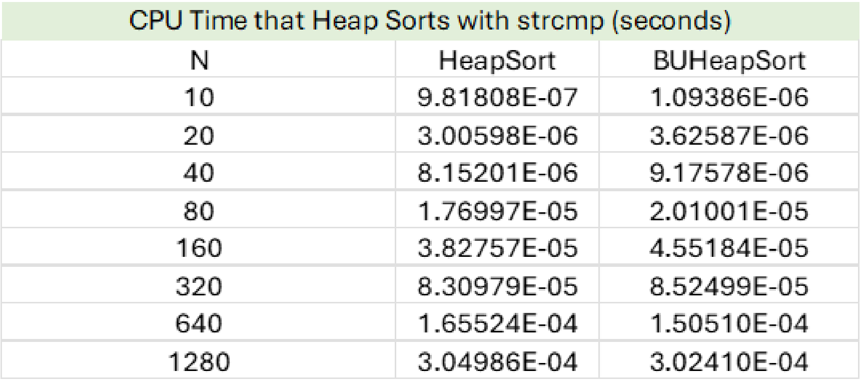
This pseudocode measures the CPU time for sorting a list of words using the HeapSort algorithm. It reads ‘n’ words, sorts the list 500 times, and records the start and end times. The total time for 500 iterations is used to calculate the average CPU time per sort. The result, including the sorting method, number of words, and average CPU time, is printed. The same process can be repeated for other sorting algorithms to compare performance.



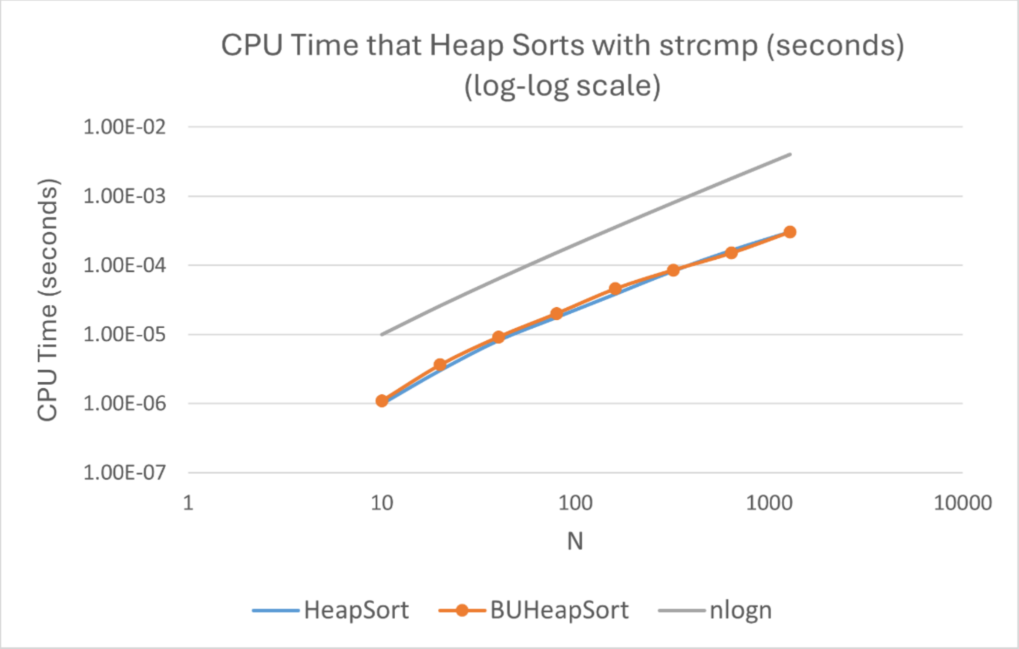
```
// main function for Homework 3.
// Input: read wordlist from stdin,
// Output: sorted wordlist and CPU time used.

Algorithm main(void){
    Read n and a list of n words and store into A array ;
    t0 := GetTime();
    repeat 500 times {
    HeapSort(list,n);
    }
t1 := GetTime();
t := (t1-t0)/500;
write (sorting method, n , CPU time) ;
 Repeat CPU time measurement for other sorts ;
write (Heap Sort) ;
}
```

Results and observations

1. Results of heap sorts with strcmp:



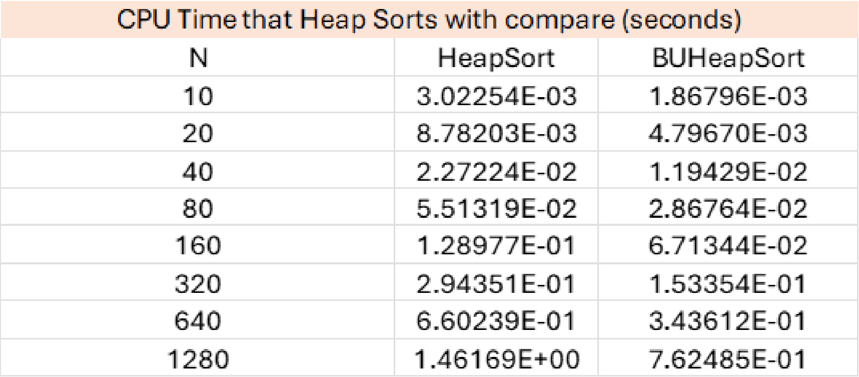
CPU Time that Heap Sorts with strcmp (seconds)		
N	HeapSort	BUHeapSort
10	9.81808E-07	1.09386E-06
20	3.00598E-06	3.62587E-06
40	8.15201E-06	9.17578E-06
80	1.76997E-05	2.01001E-05
160	3.82757E-05	4.55184E-05
320	8.30979E-05	8.52499E-05
640	1.65524E-04	1.50510E-04
1280	3.04986E-04	3.02410E-04



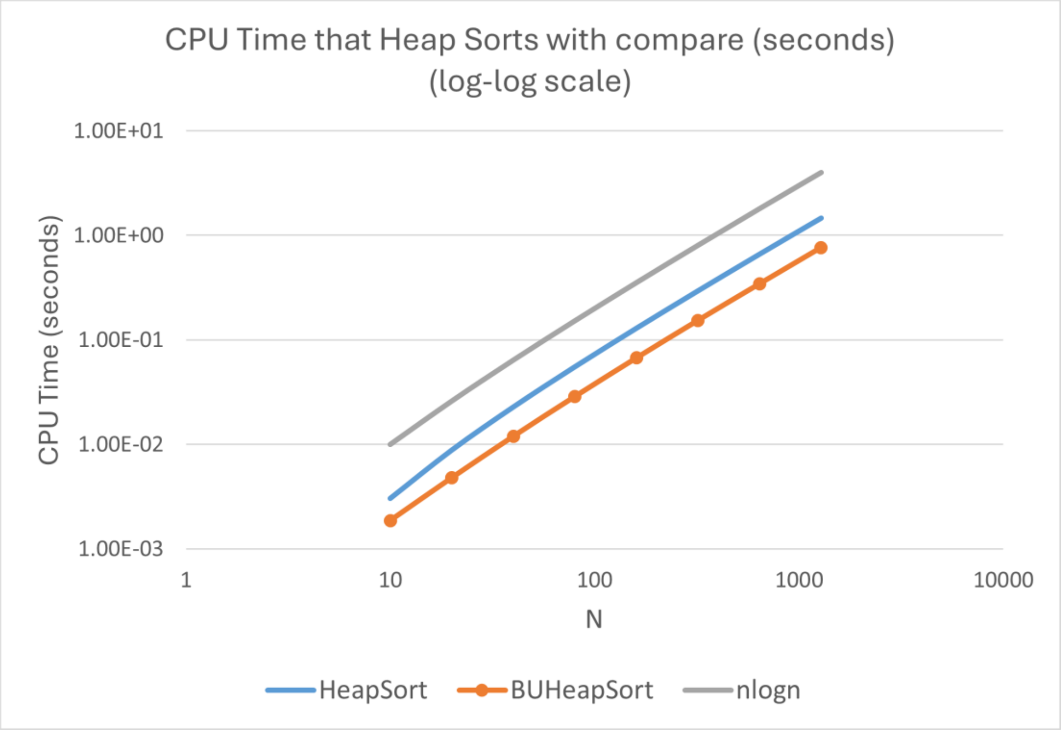
2. Observation of heap sorts with strcmp:

   BUHeap sort is a little faster than Heap sort, but when the amount of data gets more, both of their efficiency is almost the same. I think that BUHeap being a little faster is because its time complexity of building the heap is lower than the one of Heap sort. Actually, both of these two algorithms are more efficient than the sorts algorithms that we did in homework 1 because their time complexity is O(nlogn) which is much lower than those sorts that we saw in homework 1.

3. Results of heap sorts with compare:



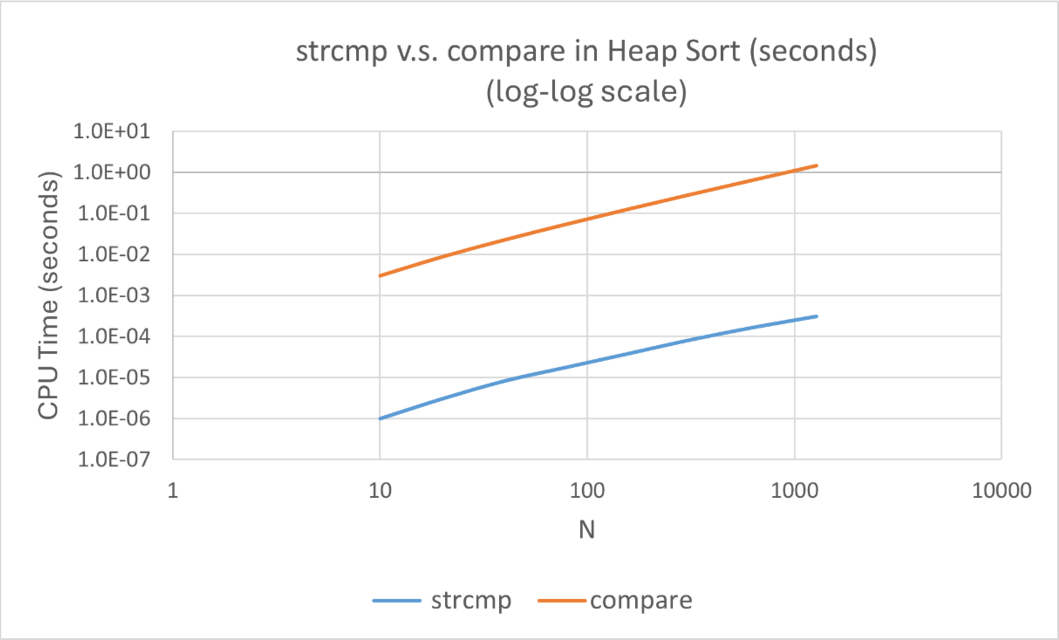
CPU Time that Heap Sorts with compare (seconds)		
N	HeapSort	BUHeapSort
10	3.02254E-03	1.86796E-03
20	8.78203E-03	4.79670E-03
40	2.27224E-02	1.19429E-02
80	5.51319E-02	2.86764E-02
160	1.28977E-01	6.71344E-02
320	2.94351E-01	1.53354E-01
640	6.60239E-01	3.43612E-01
1280	1.46169E+00	7.62485E-01

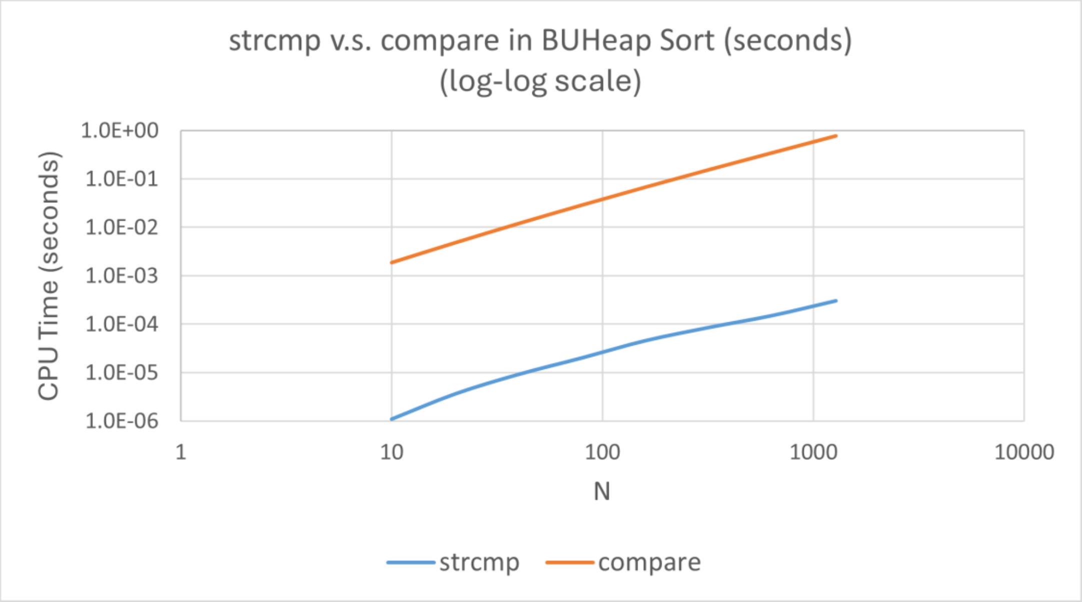


4. Observation of heap sorts with compare:

From the chart, Bottom-Up Heap Sort (BUHeapSort) has a lower CPU time compared to standard Heap Sort. This is because Bottom-Up Heap Sort reduces unnecessary comparisons and swaps during the heapify process. Heap Sort performs compares and swaps elements layer by layer. Bottom-Up Heap Sort skips some comparisons by directly finding the appropriate insertion position for the current node instead of moving it down step by step. This significantly reduces the total number of comparisons. Since the compare function includes nanosleep(10ns), each comparison has an additional cost. By reducing the number of comparisons, Bottom-Up Heap Sort can significantly lower the total execution time and make it much faster than Heap Sort in this case.

5. Strcmp v.s compare





6. Observation:

The compare function calls nanosleep(&tim, &tim2) before executing strcmp(str1, str2). This line of code makes the program pause (sleep) for 10 nanoseconds (tv_nsec = 10), so it increases the execution time of compare.

From the log-log scale in the chart, compare consistently takes about an order of magnitude (roughly **1000 times**) longer than strcmp.

Conclusion

1. Time complexity and space complexity:

	HeapSort	BUHeapSort
Time	O(n log n)	O(n log n)
Space	O(log n)	O(log n)

2. When we use strcmp function, the BUHeap sort is a little faster than Heap sort, but when the amount of data gets more, both of their

efficiency is almost the same.

3. Bottom-Up Heap Sort performs better than standard Heap Sort when we use the compare function. Since the compare function has a high overhead due to the added sleep delay. The smaller number of comparisons of Bottom-Up Heap Sort results in a significant reduction in total CPU time.

4. The compare function is significantly slower than strcmp due to the added nanosleep(10 ns), which introduces a fixed delay.