EE3980 Algorithms

Homework 5. Better Sorts

106061151 劉安得

1. Introduction

Sorting algorithms is to rearrange a set of items into an increasing or decreasing order. In this homework, we implement three sorting algorithms, heap sort (Algorithm 2.2.19), merge sort (Algorithm 3.2.1) and quick sort (Algorithm 3.2.5) and compare their efficiency using the data sets in hw01.

Heap Sort use max heap to perform sort function. A max heap is a complete binary tree with the property that the value at each node is at least as large as the values at its children (if they exist).

Merge Sort and Quick Sort are two examples of divide and conquer approach. The method of Divide and Conquer is split the input into k distinct subsets, yielding k subproblems. These k subproblems are solved individually. And then combine the subsolutions into a solution of the whole problem. Merge Sort partitions A into two sets and each set is sorted into nondecreasing order by divide and conquer. And then merge the two sets together. Quick Sort partition A by a pivot and sort them separately

2. Approach

A. HeapSort

```
// 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 := n/2 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.
    }
}

// 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×i; // A[j] is the lchild.
    item := A[i];
    done := false;
    while j <= 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[j/2] := A[j];
            j := 2×j;
        }
    }
    A[j/2] := item;
}
```

I. Correctness

Starting from the deepest internal nodes down to the root, perform Heapify on these internal nodes. Leave nodes have no lchild nor rchild, and thus no need to perform Heapify on them. After these operations, the array satisfies max heap property. And then, one can remove the maximum element and then perform Heapify to maintain the max heap property. This process repeats until the entire A array is sorted.

II. Time Complexity

Assume that element comparison(strcmp) dominates the CPU time.

Statement	s/e
`Algorithm Heapify(A, i, n)`	0
`{`	0
`  j := 2×i;`	0
`  item := A[i];`	0
`  done := false;`	0
`  while j <= n and not done do {`	0
`  if j < n and A[j] < A[j + 1] then`	1
`  j := j + 1;`	0
`  if item > A[j] then`	1
`  done := true;`	0
`  else {`	0
`  A[j/2] := A[j];`	0
`  j := 2×j;`	0
`  }`	0
`  }`	0
`  A[j/2] := item;`	0
`}`	0
Total	

$$T_{Heapify}(i,n) \le \lceil lg(n+1) \rceil - \lfloor lg(i) \rfloor \le lg(n) + 1 = \text{O(lg(n))}$$

Assume that element comparison(strcmp) dominates the CPU time.

Statement	s/e	Freq.	Total steps
`Algorithm HeapSort(A, n)`	0	-	0
`{`	0	-	0
`  for i := n/2 to 1 step -1 do`	0	n/2+1	0
`  Heapify(A, i, n);`	$T_{heap}(i, n)$	n/2	$\Sigma T_{heap}(i, n)$
`  for i := n to 2 step -1 do {`	0	n	0
`  t := A[i]; A[i] := A[1]; A[1] := t;`	0	n-1	0
`  Heapify(A, 1, i - 1);`	$T_{heap}(1, i-1)$	n-1	$\Sigma T_{heap}(1, i-1)$
`  }`	0	-	0
`}`	0	-	0
Total			

$$T(n) \leq \sum_{j=1}^{n/2} lg(n) + 1 + \sum_{j=1}^{n-1} lg(i) + 1 \leq \sum_{j=1}^{n/2} lg(n) + 1 + \sum_{j=1}^{n-1} lg(n) + 1$$

$$= (\frac{3}{2}n - 1) \times lg(n) \cong \frac{3}{2}n \times lg(n) \ = \text{O(n*lg(n))}$$

III. Space Complexity

- $S_{Heapify}(n)$

  n for array *A,* one for each variable: *i*, *j*, *n*, *item* and *done*

  $S_{Heapify}(n) = n+5 = \Theta(n)$

- S(n)

  n for array *A*, 5 for Heaptify(), and one for each variable: *n*, *i* and *t*

  $S(n) = n+8 = \Theta(n)$

B. Merge Sort

```
// Sort A[low : high] into nondecreasing order.
// Input: array A, int low, high
// Output: A rearranged.
Algorithm MergeSort(A, low, high)
{
    if (low < high) then {
        mid := (low + high)/2;
        MergeSort(A, low, mid);
        MergeSort(A, mid + 1, high);
        Merge(A, low, mid, high);
    }
}

// Merge sorted A[low : mid] and A[mid + 1 : high] to nondecreasing
order.
// Input: A[low : high], int low, mid, high
// Output: A[low : high] sorted.
Algorithm Merge(A, low, mid, high)
{
```

```
    h := low; i = low; j := mid + 1; // Initialize looping indices.
    while ((h <= mid) and (j <= high)) do { // Store smaller one to B[i].
        if (A[h] <= A[j]) then { // A[h] is smaller.
            B[i] := A[h]; h := h + 1;
        } else { // A[j] is smaller.
            B[i] := A[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then { // A[j : high] remaining.
        for k := j to high do {
            B[i] = A[k]; i := i + 1;
        }
    }
    else { // A[h : mid] remaining.
        for k := h to mid do {
            B[i] := A[k]; i := i + 1;
        }
    }
    for k := low to high do A[k] := B[k]; // Copy B to A.
}
```

I. Correctness

A is partitioned into two sets and each set is sorted into nondecreasing order by divide and conquer. And then Merge() function merge the two sets together, and B is copied into A to get the sorted result.

II. Time Complexity

Assume that element comparison(strcmp) dominates the CPU time, and n is the size of array A[low: high].

Statement	s/e
`Algorithm Merge(A, low, mid, high)`	0
`{`	0
`    h := low; i = low; j := mid + 1;`	0
`    while ((h <= mid) and (j <= high)) do {`	0

`if (A[h] <= A[j]) then {`	1
`B[i] := A[h]; h := h + 1;`	0
`} else {`	0
`B[i] := A[j]; j := j + 1;`	0
`}`	0
`i := i + 1;`	0
`}`	0
`if (h > mid) then {`	0
`for k := j to high do {`	0
`B[i] = A[k]; i := i + 1;`	0
`}`	0
`}`	0
`else {`	0
`for k := h to mid do {`	0
`B[i] := A[k]; i := i + 1;`	0
`}`	0
`}`	0
`for k := low to high do A[k] := B[k];`	0
`}`	0
Total	

$$\left\lfloor \frac{n}{2} \right\rfloor \leq T_{Merge}(n) \leq n - 1 = \Theta(n)$$

Assume that element comparison(strcmp) dominates the CPU time, and n is the size of array A[low: high].

Statement	s/e	Freq.		Total steps	
		n > 1	n = 1	n > 1	n = 1
`Algorithm MergeSort(A, low, high)`	0	-	-	0	0
`{`	0	-	-	0	0
`if (low < high) then {`	0	1	1	0	0
`mid := (low + high)/2;`	0	1	0	0	0
`MergeSort(A, low, mid);`	T(n/2)	1	0	T(n/2)	0
`MergeSort(A, mid + 1, high);`	T(n/2)	1	0	T(n/2)	0
`Merge(A, low, mid, high);`	$T_{Merge}(n)$	1	0	$T_{Merge}(n)$	0
`}`	0	-	-	0	0
`}`	0	-	-	0	0

Total			…	0

$$T(n) = \begin{cases} 0, if\ n = 1 \\ 2\mathrm{T}\left(\frac{n}{2}\right) + \mathrm{T}_{Merge}(\mathrm{n}), otherwise \end{cases}$$

$$\text{If n} = 2^k,\ T(n) \leq 2\mathrm{T}\left(\frac{n}{2}\right) + n - 1 \leq 2(2\mathrm{T}\left(\frac{n}{4}\right) + \frac{n}{2} - 1) + n - 1 = 4\mathrm{T}\left(\frac{n}{4}\right) + 2n - 3$$

$$\leq \ldots \leq 2^k\mathrm{T}(1) + kn - 2k + 1 = n \times lg(n) - 2lg(n) + 1$$

$$T(n) \geq 2\mathrm{T}\left(\frac{n}{2}\right) + \frac{n}{2} \geq 2(2\mathrm{T}\left(\frac{n}{4}\right) + \frac{n}{4}) + \frac{n}{2} = 4\mathrm{T}\left(\frac{n}{4}\right) + n$$

$$\geq \ldots \geq 2^k\mathrm{T}(1) + k\frac{n}{2} = \frac{1}{2}n \times lg(n)$$

$$\frac{1}{2}n \times lg(n) \leq T(n) \leq n \times lg(n) - 2lg(n) + 1$$

$$\mathrm{T(n)} = \Theta(\mathrm{n*lg(n)})$$

III. Space Complexity

- $\mathrm{S_{Merge}(n)}$

  2n for array *A* and *B,* one for each variable: *h*, *i*, *j*, *k*, *low*, *mid,* and *high*

  $\mathrm{S_{Merge}(n)} = 2n+7 = \Theta(n)$

- S(n)

  2n for array *A* and *B*, 7 for Merge()

  And for each recursive function call, one for each variable: *low*, *mid,* and *high*

  Assuming $n = 2^k$, $S(n) = 2n+7+\lg(n)*3 = \Theta(n)$

C. Quick Sort

```
// Sort A[low : high] into nondecreasing order.
// Input: A[low : high], int low, high
// Output: A[low : high] sorted.
Algorithm QuickSort(A, low, high)
{
    if (low < high) then {
```

```
        mid := Partition(A, low, high + 1);
        QuickSort(A, low, mid - 1);
        QuickSort(A, mid + 1, high);
    }
}

// Partition A into A[low : mid - 1] <= A[mid] and A[mid + 1 : high] >=
A[mid].
// Input: A, int low, high
// Output: j that A[low : j - 1] <= A[j] <= A[j + 1 : high].
Algorithm Partition(A, low, high)
{
    v := A[low]; i := low; j := high; // Initialize
    repeat { // Check for all elements.
        repeat i := i + 1; until (A[i] >= v); // Find i such that A[i] >=
v.
        repeat j := j - 1; until (A[j] <= v); // Find j such that A[j] <=
v.
        if (i < j) then Swap(A, i, j); // Exchange A[i] and A[j].
    } until (i >= j);
    A[low] := A[j] ; A[j] = v; // Move v to the right position.
    return j;
}

Algorithm Swap(A, i, j)
{
    t := A[i]; A[i] := A[j]; A[j] := t;
}
```

I. Correctness

Note that after Partition() is executed, A[i] <= A[j], if i < j; and A[i] >= A[j], if i > j. Therefore, the problem can be divided and conquer to two subproblem, which are sorting A[low : j-1] and A[j+1 : high] separately. And then QuickSort() can be applied to A[low : j − 1] and A[j + 1 : high].

II. Time Complexity

Assume that element comparison(strcmp) dominates the CPU time, and n is the size of array A[low: high].

Statement	s/e
`Algorithm Partition(A, low, high)`	0
`{`	0
`    v := A[low]; i := low; j := high;`	0
`    repeat {`	0
`        repeat i := i + 1; until (A[i] >= v);`	1
`        repeat j := j - 1; until (A[j] <= v);`	1
`        if (i < j) then Swap(A, i, j);`	0
`    } until (i >= j);`	0
`    A[low] := A[j] ; A[j] = v;`	0
`    return j;`	0
`}`	0
`Algorithm Swap(A, i, j)`	0
`{`	0
`    t := A[i]; A[i] := A[j]; A[j] := t;`	0
`}`	0
Total	

$$T_{Partition}(n) = n$$

Assume that element comparison(strcmp) dominates the CPU time, and n is the size of array A[low: high].

Statement	s/e	Freq.		Total steps	
n		>1	=1	> 1	=1
`Algorithm QuickSort(A, low, high)`	0	-	-	0	0
`{`	0	-	-	0	0
`    if (low < high) then {`	0	1	1	0	0
`        mid := Partition(A, low, high + 1);`	n+1	1	0	n+1	0
`        QuickSort(A, low, mid - 1);`	T(mid-low)	1	0	T(mid-low)	0
`        QuickSort(A, mid + 1, high);`	T(high-mid)	1	0	T(high-mid)	0
`    }`	0	-	-	0	0

}	0	- -	0	0
Total			…	0

$$T(n) = \begin{cases} 0, if\ n = 1 \\ \mathrm{T(mid - low) + T(high - mid)} + n + 1, otherwise \end{cases}$$

i. Best Case (If $n = 2^k-1$):

$$T(n) \geq 2T\left(\frac{\mathrm{n}-1}{2}\right) + n + 1 \geq 2\left(2T\left(\frac{\mathrm{n}-3}{4}\right) + \frac{\mathrm{n}-1}{2} + 1\right) + n + 1$$

$$= 4\mathrm{T}\left(\frac{\mathrm{n}-3}{4}\right) + 2\mathrm{n} + 2$$

$$\geq \ldots \geq 2^{k-1}\mathrm{T}(1) + \mathrm{n(k-1)} + 2(\mathrm{k}-1) = \mathrm{n} \times \lg(\mathrm{n}+1) - \mathrm{n} + 2\lg(\mathrm{n}+1) - 2 \cong n \times lg(n)$$

$$= \mathrm{O(n*lg(n))}$$

ii. Worst Case:

$$T(n) = T(n-1) + n + 1 = T(n-2) + 2n + 1 = \cdots = T(1) + \sum_{i=3}^{n+1} i = \frac{(n-1)(n+4)}{2}$$

$$\cong \frac{1}{2}n^2 = \Theta(\mathrm{n}^2)$$

iii. Average Case:

$$T(n) = \mathrm{n} + 1 + \frac{1}{n}\sum_{k=1}^{n} T(k-1) + T(n-k)$$

$$nT(n) = \mathrm{n(n+1)} + 2(T(0) + T(1) + \ldots + T(n-1)) \ \ldots (1)$$

$$(n-1)T(n-1) = \mathrm{n(n-1)} + 2(T(0) + T(1) + \ldots + T(n-2)) \ \ldots (2)$$

$$nT(n) - (n-1)T(n-1) = 2n + 2T(n-1) \ \ldots (1) - (2)$$

$$nT(n) = (n+1)T(n-1) + 2n$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \ldots = \frac{T(1)}{2} + \sum_{3}^{n+1} \frac{2}{k}$$

$$\sum_{3}^{n+1} \frac{2}{k} \leq \int_{2}^{n+1} \frac{2}{x} dx = 2\lg(\mathrm{n}+1) - 2$$

$$T(n) = (n+1)(2\lg(n+1) - 2) \cong 2n \times lg(n) = O(n*\lg(n))$$

III. Space Complexity

- $S_{Partition}(n)$

  n for array *A,* one for each variable: *i*, *j*, *v*, *low* and *high*

  $S_{Partition}(n) = n+5 = \Theta(n)$

- S(n)

  n for array *A*, 7 for Merge()

  And for each recursive function call, one for each variable: *low*, *mid,* and *high*

  Let the stack space needed by the QuickSort() is S(n)

  i. Worst Case:

  $S_W(n) = n + 3 + S_W(n-1) = \ldots = n + 3n = 4n = O(n)$

  ii. Best Case:

  $S_B(n) = n + 3 + S_B((n-1)/2) = \ldots = n + \lg(n)*3 = O(n)$

  iii. Average Case:

  Because both worst case and best case are O(n), the space complexity of average case is O(n) either

D. Main Function

```
// main function for Homework 5.
// 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 An array;
```

```
    t0 := GetTime();
    repeat 5000 times{
        Copy array A to array list;
        HeapSort(list, n);
    }
    t1 := GetTime();
    t := (t1 - t0)/500;
    write (sorting method, n, CPU time);
    Repeat CPU time measurement for other sorts;
    write (sorted list);
}
```

I. Keys of Implementation

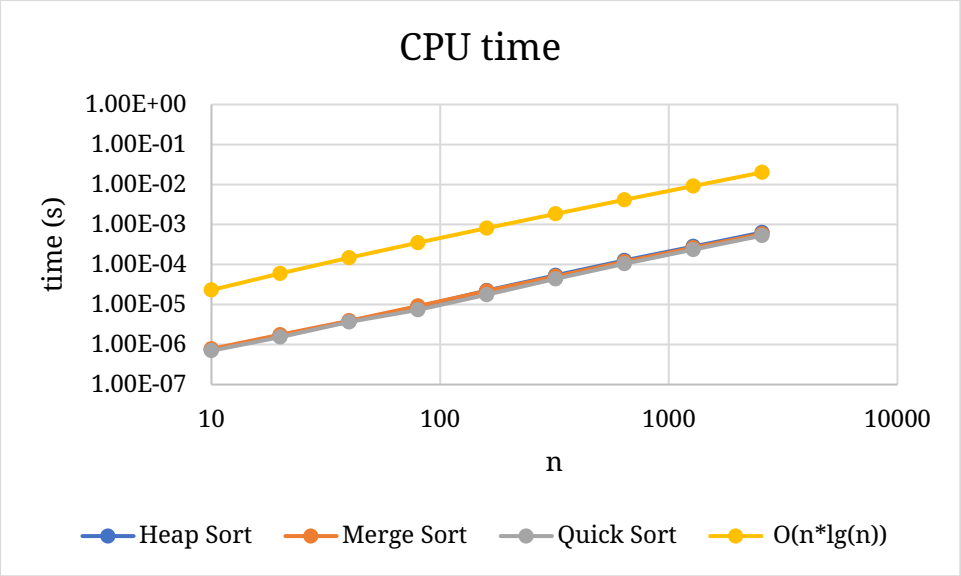
Because these three algorithms are very fast, I repeat 5000 times to measure the CPU time.

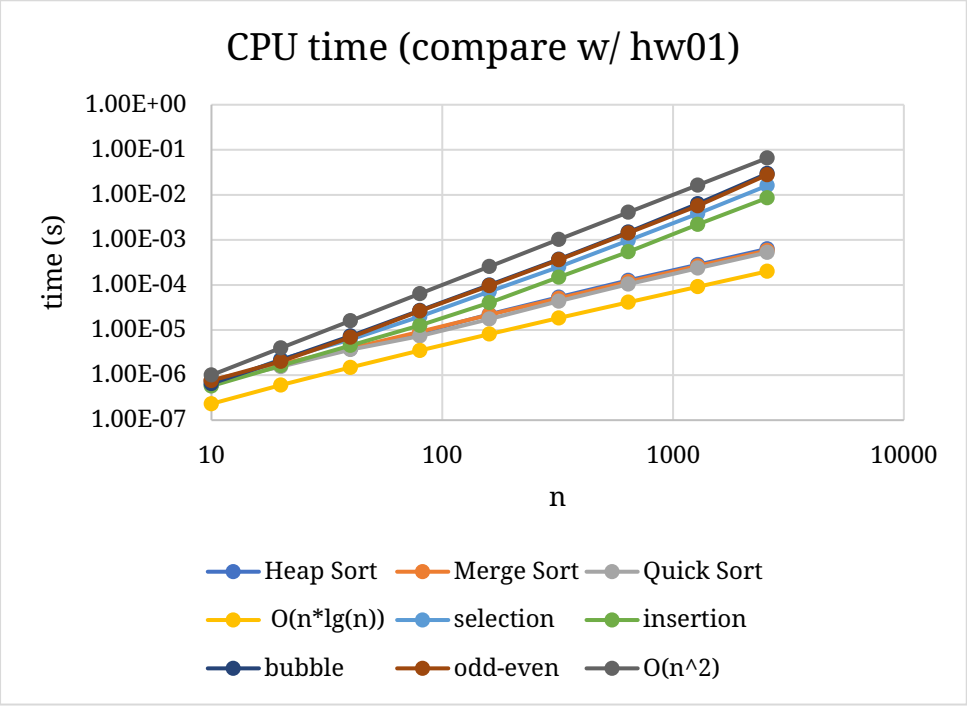
And in each iteration, we need to copy the array first, because after sorting, the array will be non-decreasing order, and it cannot be used to measure CPU time again.

3. Result & Observations

A. CPU Time (Units: s)

N	Heap Sort	Merge Sort	Quick Sort
10	7.4501e-07	7.7877e-07	7.00998e-07
20	1.68462e-06	1.75743e-06	1.53923e-06
40	3.79238e-06	3.92199e-06	3.66821e-06
80	8.67701e-06	9.16262e-06	7.3658e-06
160	2.23032e-05	2.18824e-05	1.75518e-05
320	5.3525e-05	5.0321e-05	4.36478e-05
640	0.000126114	0.000117786	0.000104347
1280	0.000282756	0.000263676	0.000236304
2560	0.00063223	0.000582277	0.000527268





B. Conclusions/ Observations

The result of analysis matches the plot, the time complexity of these three sorts is O(n*lg(n)). All of them are good sorting algorithm. Compare with the four quadratic sorts in hw01, these three sorts in this homework perform much better when n is large.

Because heap sort did not use recursive function its space complexity is the smallest.