0.1 Sorting (Merge sort, quick sort, heap sort)

- *Algorithm:* Sorting (Merge sort (algo. 1), quick sort (algo. 3), heap sort (algo. 4))
- *Input:* An unsorted array
- *Complexity:* $\mathcal{O}(n \log n)$ for all of three
- *Data structure compatibility:* Heap for heap sort, array for merge sort and quick sort
- *Common applications:* Foundation for algorithms (i.e. binary search requires sorted sequence), Data processing in excels and tables

Sorting (Merge sort, quick sort, heap sort)

A sorting algorithm places elements of an sequence in a certain order.

Description

Problem Description

Sorting is a fundamental problem in algorithms. Quite a few search or graph algorithm requires a sorted sequence as input. When studying sorting algorithm, the concept of an **in place** algorithm is needed. An **in place** algorithm is an algorithm that requires $\mathcal{O}(1)$ extra space.

Algorithm Description

A sorting probem is defined as followed: [1]
Input: A sequence of n numbers $(a_1, a_2, \cdots, a_n)$.
Output: A permutation $(a_1^{'}, a_2^{'}, \cdots, a_n^{'})$ of the input sequence such that $a_1^{'} \leq a_2^{'} \leq \cdots \leq a_n^{'}$

Merge sort

Merge sort is a divide-and-conquer algorithm. It keeps merging two sorted sequences into a new sorted sequences. Let $T(i)$ be the time of sorting i elements. Since merging two sequence together takes $\mathcal{O}(N)$ time, scanning at most all the elements.

$$T(N) = 2T(N/2) + N;$$

According to Master Theorem, the complexity of merge sort can be expressed as

$$T(N) = \mathcal{O}(N \log N)$$

Since merge sort takes $\mathcal{O}(N)$ extra place to store the stored sequence, it is not an **in place** algorithm. Another implementation of merge sort takes $\mathcal{O}(1)$ extra place, which is **in place**.

Algorithm 1: *mergeSort*(A, l, r)

Input : A subsequence of A: $A[l : r+1]$
Output: A sorted sequence C with $r - l + 1$ numbers

if $l \geq r$ **then**
  return
end if
$mid \leftarrow (l+r)/2$
$mergeSort(A, l, mid)$
$mergeSort(A, mid+1, r)$
`/* Merge the left sequence and right sequence togther. */`
$i = j = k = 0$
while $i < (mid - l + 1)$ ***and*** $j < (r - mid)$ **do**
  if $A[i] \leq B[j]$ **then**
    $C[k++] = A[j++]$
  end if
  else
    $C[k++] = B[j++]$
  end if
end while
if $i = (mid - l + 1)$ **then**
  Append $B[j : r+1]$ to C.
end if
else
  Append $A[i : mid+1]$ to C.
end if
return C

Algorithm 2: *inPlaceMergeSort*(A, l, r)

Input : A subsequence of A: $A[l : r+1]$
Output: Sorted sequence A

if $l \geq r$ **then**
  return
end if
MergeSort(A,l,$n/2$)
MergeSort(A,$n/2+1$,r)
$i \leftarrow l$
$j \leftarrow n/2+1$
$tmp \leftarrow j$
while $i < r$ **do**
  while $a[i] < a[j]$ **do**
    i++
  end while
  while $a[i] > a[j]$ **do**
    j++
  end while
  swap **a[i:index]** with **a[index:j]**
  $i \leftarrow i + j - tmp$
end while
return A

Quick sort

Quick sort is a divide-and-conquer algorithm. It works by placing one of the elements into its right place every time. This element is called a **pivotat**. The running time varies between $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$ [3] depending on the pivotat selected. Let $T(i)$ be the time of sorting i elements. Partition the sequence around *pivot* takes $\mathcal{O}(N)$ time, as the worst case is to scan all the elements. Therefore,

$$T(N) = 2T(N/2) + cN$$

According to Master Theorem, the average running time, thus the time complexity of quickSort sort can be expressed as

$$T(N) = \mathcal{O}(N \log N)$$

Quick sort is an **in place** algorithm since $\mathcal{O}(1)$ extra place is used.

Algorithm 3: $quickSort(A, l, r)$

Input : A subsequence of A: $A[l : r+1]$
Output: Sorted sequence A

if $l \geq r$ **then**
  return
end if
$pivotIndex \leftarrow$ a random number between l and r
swap $A[pivotIndex]$ and $A[0]$
$pivot \leftarrow A[0]$
$i \leftarrow 1$
$j \leftarrow N - 1$
`/* Partition the sequence around pivot */` **while** $i < j$ **do**
  while $A[i] < pivot$ **do**
    $i{+}{+}$
  end while
  while $A[j] \geq pivot$ **do**
    $j{-}{-}$
  end while
  if $i < j$ **then**
    swap $A[i]$ with $A[j]$
  end if
end while
swap $A[0]$ and $A[j]$
$quickSort(A, l, j-1)$
$quickSort(A, j+1, r)$
return A

Heap Sort

Heap sort is based on data structure **heap**. A heap is based on a complete binary tree, has the property that for any node other than the root [1]

$$A[\lfloor i/2 \rfloor] \leq A[i]$$

Procedures that are used in heap sort shows in the following. [2]

1. The *maintain* procedure is to maintain the property of a heap.

2. The *buildHeap* procedure is to produces a heap from an unsorted array.

Since the height of a complete bianry tree of N elements is $\Theta(\log N)$, *maintain* takes $\mathcal{O}(\log N)$ running time. For one complete heap sort, there are $\mathcal{O}(N)$ loops. Therefore, the time complexity of heap sort can be expressed as

$$T(N) = \mathcal{O}(N) \times \mathcal{O}(\log N) = \mathcal{O}(N \log N)$$

Algorithm 4: *heapSort*(A)

Input : An unsorted sequence A
Output: None

```
Function maintain(A, i):
    l ← 2i
    r ← 2i + 1
    if l ≤ A.heap_size and A[l] > A[i] then
        largest ← l
    end if
    else
        largest ← i
    end if
    if r ≤ A.heap_size and A[r] > A[largest] then
        largest ← r
    end if
    if largest ≠ i then
        swap A[i] with A[largest]
        maintain(A, largest)
    end if
end
Function buildHeap(A, i):
    A.heap_size ← A.length
    for i = ⌊A · length /2⌋ downto 1 do
        maintain(A,i)
    end for
end
buildHeap (A)
for i = A.length downto 2 do
    swap A[1] with A[i]
    A.heap_size − −
    maintain(A,i)
end for
return
```

References.

[1] Leiserson Charles E. Cormen Thomas H. and etc. *Introduction to Algorithms – Third Edition*. The MIT Press, 2009 (cit. on pp. 1, 3).

[2] Xiaofeng Gao. *VE281 – Data Structure and Algorithms (lecture slides)*. 2019 (cit. on p. 3).

[3] Manuel. *VE477 – Introduction to Algorithms (lecture slides)*. 2019 (cit. on p. 3).