CS 313
- Kangmei Yang
### Selection Sort

Select the appropriate value for each position:
- starting from index 0, it should contain the minimum value (from index 0 to the end)
- linear search for that minimum value in the range
- if not in the correct position, swap
- move on to next index 1
- repeat all steps for index 0 – capacity -1

**Sample 4, 6, 2, 7, 1, 3, 3**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>4</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time complexity:
index 0 – n, index 1 – n-1, index 2 – n-2, all index – O(n²)
Insertion Sort

Insert each value into previous sort list
- for each value v at index i
- compare with the value before it
- if that value is larger move down one position
- repeat until reach the first position or the value before it is smaller, insert value v
- move to next value, repeat

Sample 4, 6, 2, 7, 1, 3, 3

Time complexity:
for each index, how many values move down?
0 if value is sorted, \(O(n)\) if value is unsorted
Best case \(O(n)\), Average/Worst case \(O(n^2)\)
Sort with Priority Queue

Sort with the help of priority queue

- add all values to priority queue, then dequeue all

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>2</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Con: extra array for priority queue

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>6</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

1

2 3

7 64 3

Time complexity:
Enqueue n times + Dequeue n times
O(n log n)
Heap Sort

- In place sort
- Construct current array to heap
  - bottom up heapify - bubble down
    (starting from leaf nodes, all subtrees are heap)
- Re-use the same array to store final result
  - Once dequeue, the array has one unused entry
    - Place the dequeue value to that position
      (last position is free, then second last, so on)
    - Fill up from last position (maximum value)
- Use max-heap structure
  (0 indexed, parent index is (i - 1)/2)
  (left child index is i * 2 + 1)
  (right child index is (i + 1) * 2)
Heap Sort

bottom up heapify
- all leaf nodes are max-heap
- starting from last internal node, bubble down
  (all it’s subtree are max-heap)
- continue with all internal nodes, finish after the root

Time Complexity: \(\frac{1}{2}\) leaf nodes – no swap,
\(\frac{1}{4}\) internal with height 1 – 1 swap, etc...
1 root with height \(O(\log n)\) – \(\log n\) swap
Heapify

Time complexity Proof:
Worst Case: Each internal node, bubble all the way down to a leaf node, assuming the path is: Right -> Left -> Left -> all the way Left
#Swap is $O(\text{edge})$, besides root, 1 node = 1 edge,
#edge is $O(n)$, thus heapify time complexity $O(n)$
Sample bottom-up to maxheap:

Visit each edge at most once
Heap Sort

in place dequeue
- place the root to the last position
- dequeue root
  (replace with last element, bubble down)
- ignore last position as part of heap
- continue with heap size to 0

Time Complexity:
Each dequeue perform bubble down \(O(\log n)\)
n nodes – \(O(n \log n)\)
Heap Sort

swap root with size – 1
size--
bubble down root value
repeat
Divide-Conquer

- Recursively break down the problem into two or more sub-problems
- Until each one is simple enough to solve
- Then combine the sub-problems solutions

-Merge Sort
  - Divide the list into two halves
  - Sort the two halves recursively
  - Merge the two sorted halves
Merge Sort

- Merge Sort
- Divide the list into two halves
- Sort the two halves recursively
- Merge the two sorted halves

4 6 2 7 1 3 3 5
Divide-Conquer

-Merge Sort
  -Divide the list (Easy) $O(1)$
  -Merge the two sorted halves (Heavy) $O(n)$

Runtime
$$a = 2, b = 2, c = 1, \log_2 2 = 1 \rightarrow O(n \log n)$$

-Quick Sort
  -Divide the list (Heavy) $O(n)$
  -Pick a pivot value
  -Put all value smaller than the pivot in the first half and larger values in the second half
  -Combine the two sorted halves (Easy) $O(1)$

Runtime
$$a = 2, b = 2, c = 1, \log_2 2 = 1 \rightarrow O(n \log n)$$

Con: $b$ not always equal to 2.
Quick Sort

- Quick Sort
- Partition the list into two halves base on pivot
- Sort the two halves recursively
- Combine the two sorted halves
Quick Sort

- How do we pick the pivot?
  - Ideally it’s the median value, but we can’t get a median value with $O(n)$ or better
    - first value $\times$, last value $\times$
  - random value, median of 3 (first, middle, last)

Swap pivot with last entry, values smaller than pivot start from front of the list, values larger than pivot start from the end, swap if values are out of place

Partition is costly, to improve, quick sort switch to insertion sort when the list is short – $O(n)$
Non-comparison sort

Counting Sort
- sort the values in a range
- ideal for short range

Steps:
- Count occurrence of each value, store to count
- Modify count, to store the sum of previous count (store ending index)
- Put the input in corresponding index, decrease index

Sample: range 1 – 8

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Time complexity: $O(n + \text{range})$