Sorting

CSE 326
Data Structures
Unit 15

Reading:
Sections 7.1-7.3 Bubble and Insert sort,
7.5 Heap sort,
Section 3.2.6 Radix sort,
Section 7.6 Mergesort,
Section 7.7 Quicksort,
Section 7.8 Lower bound

Consistent Ordering

• The comparison function must provide a consistent *ordering* on the set of possible keys
  › You can compare any two keys and get back an indication of  \( a < b, a > b, \) or \( a = b \)
  › The comparison functions must be consistent
    • If \( \text{compare}(a, b) \) says \( a < b \), then \( \text{compare}(b, a) \) must say \( b > a \)
    • If \( \text{compare}(a, b) \) says \( a = b \), then \( \text{compare}(b, a) \) must say \( b = a \)

Why Sort?

• Sorting algorithms are among the most frequently used algorithms in computer science
• Allows binary search of an N-element array in \( O(\log N) \) time
• Allows \( O(1) \) time access to \( k \)th largest element in the array for any \( k \)
• Allows easy detection of any duplicates
Evaluating a Sort Algorithm: Time

- How fast is the algorithm?
  - The definition of a sorted array A says that for any i<j, A[i] < A[j]
  - This means that you need to at least check on each element at the very minimum, i.e., at least O(N)
  - And you could end up checking each element against every other element, which is O(N²)
  - The big question is: How close to O(N) can you get?

Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - Is copying needed? O(n) additional space
  - In-place sorting – no copying – O(1) additional space
  - Somewhere in between for “temporary”, e.g. O(log n) space
  - External memory sorting – data so large that does not fit in memory

Stability

- Stability: Does it rearrange the order of input data records which have the same key value (duplicates)?
  - E.g. Phone book sorted by name. Now sort by county – is the list still sorted by name within each county?
  - Extremely important property for databases
  - A stable sorting algorithm is one which does not rearrange the order of duplicate keys

Example

Stable Sort

```
5a 8 3a 5b 4 3b 2 3c
2 3a 3b 3c 4 5a 5b 8
```

Unstable Sort

```
5a 8 3a 5b 4 3b 2 3c
2 3c 3b 3a 4 5a 5b 8
```
Bubble Sort

- “Bubble” elements to their proper place in the array by comparing elements \(i\) and \(i+1\), and swapping if \(A[i] > A[i+1]\)
  - Bubble every element towards its correct position
    - last position has the largest element
    - then bubble every element except the last one towards its correct position
    - then repeat until done or until the end of the quarter, whichever comes first ...

Bubblesort

\[
bubble(A[1..n]: \text{integer array}, n: \text{integer}): \{
  i, j: \text{integer};
  \text{for} \ i = 1 \ \text{to} \ n-1 \ \text{do}
  \begin{align*}
    &\text{for} \ j = 2 \ \text{to} \ n-i+1 \ \text{do} \\
    &  \quad \text{if} \ A[j-1] > A[j] \ \text{then} \ \text{SWAP}(A[j-1], A[j]);
  \end{align*}
\}
\]

\[
\text{SWAP}(a,b): \{
  t: \text{integer};
  t:=a; a:=b; b:=t;
\}
\]

\(i=1: \text{Largest element is placed at last position}\)
\(i=k: k^{th} \text{Largest element is placed at } k^{th} \text{ to last position}\)

Bubblesort (recursive)

\[
bubble(A[1..n]: \text{integer array}, n: \text{integer}): \{
\}
\]

Put the largest element in its place
Put 2nd largest element in its place

Two elements done, only n-2 more to go ...

Bubble Sort: Just Say No

- “Bubble” elements to to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
- We bubblize for i=1 to n (i.e, n times)
- Each bubblization is a loop that makes n-i comparisons
- This is O(n^2)

Insertion Sort

- What if first k elements of array are already sorted?
  - 4, 7, 12, 5, 19, 16
- We can shift the tail of the sorted elements list down and then insert next element into proper position and we get k+1 sorted elements
  - 4, 5, 7, 12, 19, 16

Insertion Sort

```java
InsertionSort(A[1..N]: integer array, N: integer) {
  i, j, temp: integer;
  for i = 2 to N {
    temp := A[i];
    j := i-1;
    while j > 1 and A[j-1] > temp {
      A[j] = temp;
    }
  }
}
```

- Is Insertion sort in place? Stable? Running time = ?
- Have we used something similar before?
Insertion Sort Characteristics

- In place and Stable
- Running time
  - Worst case is $O(N^2)$
    - reverse order input
    - must copy every element every time
- Good sorting algorithm for almost sorted data
  - Each item is close to where it belongs in sorted order.

Inversions

- An inversion is a pair of elements in wrong order
- By definition, a sorted array has no inversions
- So you can think of sorting as the process of removing inversions in the order of the elements
### Inversions

- A single value out of place can cause several inversions

![Inversion Example](image)

- Our simple sorting algorithms so far swap adjacent elements and remove just one inversion at a time
  
  › Their running time is proportional to number of inversions in array

- Given N distinct keys, the maximum possible number of inversions is

\[
(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}
\]

### Reverse order

- All values out of place (reverse order) causes numerous inversions

![Reverse Order Example](image)

### Inversions and Adjacent Swap Sorts

- "Average" list will contain half the max number of inversions = \(\frac{(n-1)n}{4}\)

  › So the average running time of Insertion sort is \(\Theta(N^2)\)

- Any sorting algorithm that only swaps adjacent elements requires \(\Omega(N^2)\) time because each swap removes only one inversion (lower bound)
Heap Sort

- We use a Max-Heap
- Root node = A[1]
- Keep track of current size N (number of nodes)

Using Binary Heaps for Sorting

- Build a max-heap
- Do N DeleteMax operations and store each Max element as it comes out of the heap
- Data comes out in largest to smallest order
- Where can we put the elements as they are removed from the heap?

1 Removal = 1 Addition

- Every time we do a DeleteMax, the heap gets smaller by one node, and we have one more node to store
  - Store the data at the end of the heap array
  - Not "in the heap" but it is in the heap array

Repeated DeleteMax

N = 3

N = 2
Heap Sort is In-place

- After all the DeleteMaxs, the heap is gone but the array is full and is in sorted order

---

Bucket Sort: Sorting Integers

- The goal: sort N numbers, all between 1 to k.
- Example: sort 8 numbers 3,6,7,4,11,3,5,7. All between 1 to 12.
- The method: Use an array of k queues. Queue j (for 1 ≤ j ≤ k) keeps the input numbers whose value is j.
- Each queue is denoted ‘a bucket’.
- Scan the list and put the elements in the buckets.
- Output the content of the buckets from 1 to k.

---

Heapsort: Analysis

- Running time
  - time to build max-heap is O(N)
  - time for N DeleteMax operations is N O(log N)
  - total time is O(N log N)
- Can also show that running time is Ω(N log N) for some inputs,
  - so worst case is Θ(N log N)
  - Average case running time is also O(N log N)
- Heapsort is in-place but not stable (why?)

---

Bucket Sort: Sorting Integers

- Example: sort 8 numbers 3,6,7,4,11,3,9,7 all between 1 to 12.
- Step 1: scan the list and put the elements in the queues

  1 2 3 4 5 6 7 8 9 10 11 12
  3 4 6 7 9 11

- Step 2: concatenate the queues

  3 3 4 6 7 9 11 → 3,3,4,6,7,7,9,11
- Time complexity: O(n+k)
Radix Sort: Sorting integers

- Historically goes back to the 1890 census.
- Radix sort = multi-pass bucket sort of integers in the range 0 to $B^p-1$
- Bucket-sort from least significant to most significant “digit” (base B)
- Requires $P(B+N)$ operations where $P$ is the number of passes (the number of base B digits in the largest possible input number).
- If $P$ and $B$ are constants then $O(N)$ time to sort!

Radix Sort Example

Input data

<table>
<thead>
<tr>
<th>478</th>
<th>537</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>38</td>
<td>123</td>
<td>67</td>
</tr>
</tbody>
</table>

Bucket sort by 1's digit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>132</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>721</td>
<td>132</td>
<td>537</td>
<td>67</td>
<td>478</td>
</tr>
</tbody>
</table>

This example uses $B=10$ and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

Radix Sort Example

After 1st pass

<table>
<thead>
<tr>
<th>3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>123</td>
</tr>
<tr>
<td>123</td>
<td>67</td>
</tr>
<tr>
<td>537</td>
<td>478</td>
</tr>
<tr>
<td>38</td>
<td>9</td>
</tr>
</tbody>
</table>

Bucket sort by 10's digit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>09</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
</tr>
</tbody>
</table>

After 2nd pass

<table>
<thead>
<tr>
<th>3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>123</td>
</tr>
<tr>
<td>123</td>
<td>67</td>
</tr>
<tr>
<td>537</td>
<td>478</td>
</tr>
<tr>
<td>38</td>
<td>9</td>
</tr>
</tbody>
</table>

Bucket sort by 100's digit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>09</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
</tr>
</tbody>
</table>

After 3rd pass

<table>
<thead>
<tr>
<th>3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>123</td>
</tr>
<tr>
<td>123</td>
<td>67</td>
</tr>
<tr>
<td>537</td>
<td>478</td>
</tr>
<tr>
<td>38</td>
<td>9</td>
</tr>
</tbody>
</table>

Bucket sort by 1000's digit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>003</td>
<td>123</td>
<td>478</td>
<td>537</td>
<td>221</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>478</td>
</tr>
</tbody>
</table>

Invariant: after k passes the low order k digits are sorted.
Properties of Radix Sort

• Not in-place
  › needs lots of auxiliary storage.
• Stable
  › equal keys always end up in same bucket in the same order.
• Fast
  › Time to sort \(N\) numbers in the range 0 to \(B^p-1\) is \(O(P(B+N))\) (\(P\) iterations, \(B\) buckets in each)

“Divide and Conquer”

• Very important strategy in computer science:
  › Divide problem into smaller parts
  › Independently solve the parts
  › Combine these solutions to get overall solution
• Idea 1: Divide array into two halves, \textit{recursively} sort left and right halves, then \textit{merge} two halves  Mergesort
• Idea 2: Partition array into items that are “small” and items that are “large”, then recursively sort the two sets  Quicksort

Mergesort

\[
\begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

• Divide it in two at the midpoint
• Conquer each side in turn (by recursively sorting)
• Merge two halves together

Mergesort Example

\[
\begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 \\
\end{array}
\]
Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Merging

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Merging

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Merging

```
2 4 8 9 1 3 5 6
```

Auxiliary array

Merging

```
2 4 8 9 1 3 5 6
```

Auxiliary array
Merging

```c
Merge(A[], T[] : integer array, left, right : integer) : {
    mid, i, j, k, l, target : integer;
    mid := (right + left)/2;
    i := left; j := mid + 1; target := left;
    while i < mid and j < right do
        else T[target] := A[j]; j := j + 1;
        target := target + 1;
    if i > mid then //left completed//
        for k := left to target-1 do A[k] := T[k];
    if j > right then //right completed//
        k := mid; l := right;
        while k > i do A[l] := A[k]; k := k-1; l := l-1;
        for k := left to target-1 do A[k] := T[k];
}
```

Recursive Mergesort

```c
Mergesort(A[], T[] : integer array, left, right : integer) : {
    if left < right then
        mid := (left + right)/2;
        Mergesort(A,T,left,mid);
        Mergesort(A,T,mid+1,right);
        Merge(A,T,left,right);
}
MainMergesort(A[1..n]: integer array, n : integer) : {
    T[1..n]: integer array;
    Mergesort[A,T,1,n];
}
```

Iterative Mergesort

```c
Merge by 1
Merge by 2
Merge by 4
Merge by 8
```
Iterative Mergesort

Iterative Mergesort

IterativeMergesort(A[1..n]: integer array, n : integer) : {
//precondition: n is a power of 2
  i, m, parity : integer;
  T[1..n]: integer array;
  m := 2; parity := 0;
  while m <= n do
    for i = 1 to n - m + 1 by m do
      if parity = 0 then Merge(A, T, i, i + m - 1);
      else Merge(T, A, i, i + m - 1);
      parity := 1 - parity;
    m := 2 * m;
    if parity = 1 then
      for i = 1 to n do A[i] := T[i];
  }
}

How do you handle non-powers of 2?
How can the final copy be avoided?

Mergesort Analysis

- Let T(N) be the running time for an array of N elements
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
- Each recursive call takes T(N/2) and merging takes O(N)

Mergesort Recurrence Relation

- The recurrence relation for T(N) is:
  > T(1) ≤ a
    - base case: 1 element array constant time
  > T(N) ≤ 2T(N/2) + dN
    - Sorting N elements takes
      – the time to sort the left half
      – plus the time to sort the right half
      – plus an O(N) time to merge the two halves
- T(N) = ?
**Mergesort Analysis**

**Upper Bound**

\[ T(n) \leq 2T(n/2) + dn \]

Assuming \( n \) is a power of 2

\[ \leq 2(2T(n/4) + dn/2) + dn \]

\[ = 4T(n/4) + 2dn \]

\[ \leq 4(2T(n/8) + dn/4) + 2dn \]

\[ = 8T(n/8) + 3dn \]

\[ \vdots \]

\[ \leq 2^k T(n/2^k) + kdn \]

\[ = nT(1) + kdn \quad \text{if} \quad n = 2^k \quad n = 2^k, \; k = \log n \]

\[ \leq cn + dn \log_2 n \]

\[ = O(n \log n) \]

---

**Properties of Mergesort**

- Not in-place
  - Requires an auxiliary array (\( O(n) \) extra space)
- Stable
  - Make sure that left is sent to target on equal values.
- Iterative Mergesort reduces copying.

---

**Quicksort**

- Quicksort uses a divide and conquer strategy, but does not require the \( O(N) \) extra space that MergeSort does
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in \( O(1) \) time

---

**“Four easy steps”**

- To sort an array \( S \)
  1. If the number of elements in \( S \) is 0 or 1, then return. The array is sorted.
  2. Pick an element \( v \) in \( S \). This is the pivot value.
  3. Partition \( S-\{v\} \) into two disjoint subsets, \( S_1 = \{ \text{all values } x \leq v \} \), and \( S_2 = \{ \text{all values } x \geq v \} \).
  4. Return QuickSort(\( S_1 \)), \( v \), QuickSort(\( S_2 \))
The steps of QuickSort

1. Select pivot value
2. Partition S
3. QuickSort(S₁) and QuickSort(S₂)
4. Voila! S is sorted

Details, details

- Implementing the actual partitioning
- Picking the pivot
  - want a value that will cause |S₁| and |S₂| to be non-zero, and close to equal in size if possible
- Dealing with cases where an element equals the pivot

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - the elements in left sub-array are ≤ pivot
  - elements in right sub-array are ≥ pivot
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions

Partitioning: Choosing the pivot

- One implementation (there are others)
  - median3 finds pivot and sorts left, center, right
    - Median3 takes the median of leftmost, middle, and rightmost elements
    - An alternative is to choose the pivot randomly (need a random number generator; “expensive”)
    - Another alternative is to choose the first element (but can be very bad. Why?)
  - Swap pivot with next to last element
Partitioning in-place

› Set pointers i and j to start and end of array
› Increment i until you hit element A[i] > pivot
› Decrement j until you hit element A[j] < pivot
› Swap A[i] and A[j]
› Repeat until i and j cross
› Swap pivot (at A[N-2]) with A[i]

Example

Choose the pivot as the median of three

Choose the pivot as the median of three

Median of 0, 6, 8 is 6. Pivot is 6

Median of 0, 6, 8 is 6. Pivot is 6

Place the largest at the right and the smallest at the left.
Swap pivot with next to last element.

Example

Example

Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap
Recursive Quicksort

Quicksort(A[]): integer array, left, right : integer): {
  pivotindex : integer;
  if left + CUTOFF ≤ right then
    pivot := median3(A, left, right);
    pivotindex := Partition(A, left, right-1, pivot);
    Quicksort(A, left, pivotindex - 1);
    Quicksort(A, pivotindex + 1, right);
  else
    Insertionsort(A, left, right);
}

Don’t use quicksort for small arrays.  CUTOFF = 10 is reasonable.

Quicksort Best Case Performance

• Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  › T(0) = T(1) = O(1)
    • constant time if 0 or 1 element
  › For N > 1, 2 recursive calls plus linear time for partitioning
    › T(N) = 2T(N/2) + O(N)
      • Same recurrence relation as Mergesort
    › T(N) = O(N log N)

Properties of Quicksort

• Not stable because of long distance swapping.
• No iterative version (without using a stack).
• Pure quicksort not good for small arrays.
• “In-place”, but uses auxiliary storage because of recursive call (O(logn) space).
• O(n log n) average case performance, but O(n^2) worst case performance.

Quicksort Worst Case Performance

• Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  › T(N) ≤ a for N ≤ C
  › T(N) ≤ T(N-1) + bN
    ≤ T(N-2) + b(N-1) + bN
    ≤ T(C) + b(C+1) + ... + bN
    ≤ a + b(C + (C+1) + (C+2) + ... + N)
  › T(N) = O(N^2)
• Fortunately, average case performance is O(N log N) (see text for proof)
How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in $O(N \log N)$ best case running time
- Can we do any better?
- No, if sorting is comparison-based.
- We saw that radix sort is $O(N)$ but it is only for integers from bounded-range.

Permutations

- How many possible orderings can you get?
  - Example: a, b, c ($N = 3$)
  - $(a \ b \ c), (a \ c \ b), (b \ a \ c), (b \ c \ a), (c \ a \ b), (c \ b \ a)$
  - $6$ orderings $= 3 \cdot 2 \cdot 1 = 3!$ (i.e., “3 factorial”)
  - All the possible permutations of a set of 3 elements
- For $N$ elements
  - $N$ choices for the first position, $(N-1)$ choices for the second position, …, $(2)$ choices, $1$ choice
  - $N(N-1)(N-2)\cdots(2)(1) = N!$ possible orderings

Sorting Model

- Recall the basic assumption: we can only compare two elements at a time
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given $N$ elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c ($N = 3$)

Decision Tree

The leaves contain all the possible orderings of a, b, c
Decision Trees

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
    - i.e., the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for N distinct elements?
    - N!, i.e., a leaf for each possible ordering
- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

Decision Trees and Sorting

- Every comparison-based sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - i.e., by making comparisons
  - Each decision reduces the possible solution space by one half
- Run time is \( \geq \) maximum no. of comparisons
  - maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

Decision Tree Example

![Diagram of a decision tree example]

- 3! possible orders
- Actual order: a < b < c

How many leaves on a tree?

- Suppose you have a binary tree of height d.
  - How many leaves can the tree have?
    - \( d = 1 \) at most 2 leaves,
    - \( d = 2 \) at most 4 leaves, etc.
**Lower bound on Height**

- A binary tree of height $d$ has at most $2^d$ leaves
  - depth $d = 1$ 2 leaves, $d = 2$ 4 leaves, etc.
  - Can prove by induction
- Number of leaves, $L \leq 2^d$
- Height $d \geq \log_2 L$
- The decision tree has $N!$ leaves
- So the decision tree has height $d \geq \log_2(N!)$

**log($N!$) is $\Omega(N \log N)$**

\[
\log(N!) = \log(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)) = \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2} \\
\geq \frac{N}{2} \log \frac{N}{2} \\
\geq \frac{N}{2} (\log N - 2) = \frac{N}{2} \log N - \frac{N}{2} \\
= \Omega(N \log N)
\]

**Summary of Sorting**

- Sorting choices:
  - $O(N^2)$ – Bubblesort, Insertion Sort
  - $O(N \log N)$ average case running time:
    - Heapsort: In-place, not stable.
    - Mergesort: $O(N)$ extra space, stable.
  - Run time of any comparison-based sorting algorithm is $\Omega(N \log N)$