$O(1)$ (constant time) < $O(logn)$ (logarithmic time) < $O(n)$ (linear time) < $O(n\ log\ n)$ (quasilinear time) < $O(n^2)$ (quadratic time) < $O(n^3)$ (cubic time) < $O(2^n)$ (exponential time) < $O(n!)$ (exponential time)

Bubble Sort: $O(n^2)$

1. 比较相邻的元素。如果第一个比第二个大（升序排序），就交换它们。
2. 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

Selection Sort: $O(n^2)$

1. 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。
2. 以此类推，直到所有元素均排序完毕。

Insertion Sort: $O(n^2)$

插入排序（Insertion Sort）是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描（对于单向链表则从前向后扫描），找到相应位置并插入。插入排序在实现上，在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

1. 将整个数列分为「已排序」和「未排序」两部分。初始时，「已排序」部分只有一个元素。
2. 从「未排序」的元素中，逐一取出元素插入到「已排序」部分，其位置是通过与「已排序」部分的元素逐一比较确定的。插入过程中，「已排序」部分的元素根据比较结果后移，为新元素腾出插入位置。
3. 重复上述过程，直到「未排序」部分元素全部插入「已排序」部分。

Merge Sort: Divide & Conquer, $O(n\ log\ n)$

1. **分解**：首先，列表从中间被分成两半，创建两个子列表。这一过程会递归继续，直到子列表只包含一个元素停止。一个元素的列表被认为是已排序的。
2. **合并**：接着，开始合并这些子列表，以创建多个较小的已排序列表，最终合并成一个单一的已排序列表。在合并过程中，会持续地将最小的元素从子列表中选出来，放入新的已排序列表中，直到所有的子列表都为空。

Quick Sort: $O(n\ log\ n)$, 最坏情况为$O(n^2)$，但通常比此更快

1. **选择基准值：** 在列表中选择一个元素作为"基准"（pivot）。
2. **分区操作：** 所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准的后面（相同的数可以到任何一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. **递归地对基准值前后的子序列进行分区操作：** 递归地将前两步骤应用于更小的子序列。

Random Quick Sort:

其选择一个随机的基准值，而不是固定的值（第一个元素，最后一个元素，或中间的元素）

引入随机选择基准的优势在于，它降低了算法在面对某些特定数据集时表现不佳的可能性。

随机快速排序的平均时间复杂度通常仍为 $O(n\ log\ n)$，这使其成为非常有效的排序算法。然而，需要注意的是，在最坏的情况下，其时间复杂度仍然为 $O(n^2)$。不过，通过随机化基准元素的选择，这种最坏情况出现的概率会大大降低。

Counting Sort: $O(n)$，但是对于数值差距大的数据，其需要大量的空间

计数排序一种非比较整数排序算法，意味着它不会比较待排序的元素。这种方法是基于对一组对象计数来确定排序后的位置。它适用于小范围的整数排序，并且效率非常高，时间复杂度可以达到 O(n)。但是，它不适用于数值差距大的数据集，因为它需要的额外存储空间与待排序数据的具体值（不是数据的数量）成正比。

1. **找出待排序的数组中最大和最小的元素**。
2. **统计数组：** 建立一个统计数组，利用索引来表示原始数组中的元素，用统计数组中的值来表示原始数组中对应元素的数量。具体而言，首先初始化一个计数数组，数组长度为原始数组最大元素和最小元素的差+1，所有位置的值都设为 0。然后遍历原始数组，每遇到一个数，就在计数数组对应的位置上加 1。
3. **累加数组：** 对统计数组做变形处理，累加前面的数，使得统计数组每个索引位置的值是对应值及其之前的所有数值的总和。这一步使得统计数组包含了位置信息，对于每个数而言，统计数组中对应值的部分表示了它在输出数组中应处的位置。
4. **输出结果：** 最后，倒序遍历原始数组，从统计数组找到正确位置，输出到结果数组。同时，每放置一个数到排序后的数组中，就将其对应的计数减少。

Radix Sort:

基数排序（Radix Sort）是一种非比较型整数排序算法，其方法是通过分配和收集来利用整数的内部结构实现排序。基数排序不是通过比较数值来排序，而是通过按数字级别分配桶来实现排序，每个级别的排序可以采用不同的排序算法。一般情况下，对于十进制

数，就会基于数位来进行排序，先从最低有效数字（比如个位）开始，一直到最高有效数字。
以 LSD 为例，即现根据最低有效位排序，然后逐渐向最高有效位过度：

1. **创建桶：** 首先，初始化一个足够数量的桶，每个桶代表一个特定的数位（例如，十进制数有 10 个桶，从 0 到 9）。
2. **分配过程：** 按照从最低有效位开始的顺序，将每个数分配到对应的桶中。例如，如果是个位数，那么数字 "25" 将会被放入 "5" 号桶。
3. **收集过程：** 在每个位的分配结束后，从桶中按照顺序收集数值，这一过程会保持之前位数的排序状态。
4. **重复过程：** 然后，重复分配和收集的步骤，针对更高的位数，直至最高有效位。

- $O(nk)$, n 为元素的数量，k 为数字的最大长度

Array Operation

`get(i)`: return `A[i]`, $O(1)$
`search(v)`: check each index one by one to see if `A[i] == v`, best case $O(1)$, worst case $O(n)$
`insert(i, v)`: shift item from index i to the end, to `i+1` until `end+1`, then set `A[i] = v`, best case $O(1)$, worst case $O(n)$
`remove(i)`: shift item from i+1 to end, to i to end-1, overwriting old `A[i]`, best case $O(1)$, worst case $O(n)$

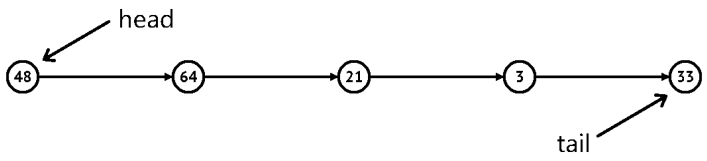
Linked List

Additional Variables:

- `head`: pointer points to `a0`
- `val`: the current number of elements N in the linked list
- `tail`: pointer points to `a(n-1)`

Operations:

- `get(i)`: only keep the head and tail pointers, the list traversal is needed. So it is much slower than array. Its running time is $O(n)$
- `search(v)`: can only search from head, since only `next` pointer to point the next element but not previous. $O(n)$
- `insert(i, v)`: four possibilities
  1. Insert to the head (before the current first item), $O(1)$
  2. An empty linked list, $O(1)$
  3. The position beyond the last (the current tail), $O(1)$
  4. The other positions of the linked list, $O(N)$
- `remove(i)`: three possibilities:
  1. The head, $O(1)$
  2. The tail, $O(N)$, since tail pointer need to be updated to the i-1 element, go to this element needs $O(N)$
  3. The other positions of the linked list, $O(N)$



Stack

LIFO
`push`, `pop` are both $O(1)$

- `push` 会将元素加在栈顶
- `pop` 会将栈顶的元素移除

Postfix expression

- push operand to stack, pop first 2 operand if an operator pushed in
- 4 1 2 9 3 / * + 5 * +
  - (9/3 * 2 + 1) * 5 + 4 = 39

Queue

FIFO
`push`, `pop` are both $O(1)$

- `push` 会将元素加在末尾
- `pop` 会将第一个元素移除

Doubly Linked List

Add `prev` pointer to linked list, make faster

Double-Ended Queue (Deque)

Can only search from head/tail, insert new item to head/tail, remove item from head/tail