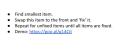
*Sorting Summary

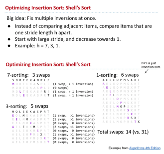
Mechanic Properties

假设我们有一个长为 N 的数组。

Algorithm	Algorithm Logic	Best Case Runtime	Worst Case Runtime	Memory	In Place	Stable

Selection Sort
选择排序
	将数组看成已排序+未排序的两部分。
	找出最小的元素，和数组未排序部分的首位元素交换，已排序部分扩容。
	
		$\Theta(N^2)$
		Cost Model: 交换次数
		外层循环遍历不同的未排序数组，内层循环找出每个未排序数组的最小元素。
		因为查找最小元素势必要事先查看所有元素，所以内层循环 $\Theta(N)$ ，外层循环也是 $\Theta(N)$ 。
			$\Theta(N^2)$
			*
				$\Theta(1)$
				不需要额外的数据结构
					Yes	No
						比如数组 $[3_1, 3_2, 2, 1, 4]$ 其中 3_1 表示第一个 3 ， 3_2 表示第二个 3 ，我们期望在排序后下标 $1, 2$ 的顺序不变。但是根据算法， 3_1 和 1 交换， 3_2 和 2 交换，于是结果变成 $[1, 2, 3_2, 3_1, 4]$ ，不满足 `Stable` 条件。
						其他例子
						`Selection Sort: 3a, 3b, 3c, 1`
						`[3a 3b 3c *1*]`
						`1 [3b 3c 3a]`
						`1 3b [3c 3a]`
						`1 3b 3c [3a]`

Insertion Sort 插入排序	将数组看成已排序+未排序的两部分。 对未排序元素中的首位元素在已排序元素中找到自己的位置插入。	$\Theta(N)$ Cost Model: 交换次数 当数组元素本来就已经排好序。 此时我们不需要额外的时间来寻找插入的位置。	$\Theta(N^2)$ *	$\Theta(1)$ 不需要额外的数据结构，只依赖于元素交换	Yes	Yes 因为当待插入元素大于等于左侧元素时，就不会在与其交换了。比如 $[1, 2, 3_1, 4, 3_2, 5]$ ，此时 3_2 在和 4 交换后就不会在和 3_1 交换了。满足 `Stable` 的条件。

Shell's Sort 希尔排序	作为 `Insertion Sort` 的衍生。 	$\Theta(N)$	$\Omega(NlogN)$ (在CS170中证明)	$\Theta(1)$ In Place 实现即可	Yes	Yes
Heap Sort 堆排序	**阶段一:** 首先将数组进行 `Bottom Up Heapification`, 阶段二: 逐个 `Pop largest item`。同时 `re-heapify` 剩余的 `heap` 直到 `Heap Size=1`	$\Theta(N)$ Cost Model: 交换次数。 阶段一耗时 $\Theta(N)$。 当数组所有元素都相等时。此时阶段二每个元素的 `re-heapify` 的时间为 $\Theta(1)$。	$\Theta(NlogN)$ * 阶段一耗时 $\Theta(N)$。 阶段二每个元素的 `re-heapify` 的时间为 $\Theta(logN)$。	`In-place`: $\Theta(1)$ `Naive`: $\Theta(N)$, 需要创建额外的数组来存放 `removeMin()` 的元素。	`In-place Implementation`: Yes `Naive Implementation`: No	No 假设数组为 $[1_a, 1_b, 1_c]$ 则阶段一 `Heapify` 的结果是 $[1_a, 1_b, 1_c]$ 阶段二的步骤如下: $[1_b, 1_c], 1_a$ $[1_c], 1_b, 1_a$ $1_c, 1_b, 1_a$

Merge Sort 归并排序	分而治之的思路。	$\Theta(NlogN)$ 求解 $T(N) = 2T(N/2) + N$ 即可。	$\Theta(NlogN)$ *	$\Theta(N)$ 需要很多额外的数组来存放 `Merge` 的结果。	Generally No. But can be. `In-place Mergesort` 实现起来非常复杂。	Yes 在 `Merge` 的过程中，假设我们有两个待合并的数组：$[1_1, 3, 4]$，$[1_2, 5, 6]$ 只要规定如果第一个数组的元素小于等于第二个数组的元素，就把第一个数组中的元素放到 `Merged` 结果中即可，结果为 $[1_1, 1_2, 3, 4, 5, 6]$。符合 `Stable` 条件。

Deterministic Quick Sort
 	
快速排序
	选取 `Pivot`，然后将剩余元素根据大小关系防止在 `Pivot` 的左右两侧。然后分而治之。	$\Theta(NlogN)$
		
		如果使用 `Hoare's Partitioning`，则数组元素全部相同时为 `Best Case`。
		
		数组随机时，不论使用什么 `Partitioning` 策略，都是 `Best Case`
			$\Theta(N^2)$ *
			
			如果使用 `3-scan Paritioning`，则数组元素全部相同是为 `worst case`。
			
			数组已经排好序(不全相同)时，是 `Worst Case`。
				`3-scan`：$\Theta(N)$
				
				`Hoare`：$\Theta(1)$
					`3-scan`：No
					
					`Hoare`：Yes
						No
						
						`Quicksort: 3, 5a, 2, 5b, 1`
						`[-3- *5a* 2 5b ~1~]`
						`[-3- 1 2 5b 5a]`
						`[-3- 1 *2* ~5b~ 5a]`
						`[-3- 1 2 *~5b~* 5a]`
						`[-3- 1 ~2~ *5b* 5a]`
						`"L" and "R" pointers cross, swap pivot.`
						`[1 2] 3 [5b 5a]`
						
						`[-1- 2] 3 [-5b- 5a]`
						`[-1- *~2~*] 3 [-5b- *~5a~*]`
						`[-1- ~~ *2*] 3 [-5b- ~~ *5a*]`
						`[-1-] [-2-] 3 [-5b-] [5a]`
						`1 2 3 5b 5a`

Random Quick Sort 随机快速排序	略，主要是通过随机选择 `Pivot` 或者在排序前对数组进行 `Random Shuffle` 。	$\Theta(logN)$	$\Theta(NlogN)$ * expected	$\Theta(logN)$ Call Stack		
Counting Sort 计数排序	**阶段一:** 构建大小为 R 的字典，统计每个不同的 `Key` 出现的次数。 **阶段二:** 根据阶段一构建的字典，计算每一个 `Key` 的元素的起始位置字典，大小也为 R 。 **阶段三:** 根据阶段二构建的起始位置字典，从原数组开始遍历，将元素插入到起始位置并更新起始位置。	$\Theta(N+R)$ N 是遍历原数组的时间 R 是构建字典的时间和更新字典的时间	$\Theta(N+R)$	$\Theta(N+R)$ N 是结果数组的大小， R 是构建字典数组的大小。	No	Yes

Radix Sort – LSD 基数排序	如果比较的是字符串，则对右侧进行填补 `\0` 使得所有字符串长度一致。 如果比较的是数字，则高位补零直到所有数字的位数一致。 然后从最低位开始调用 `Counting Sort` 。	$\Theta(W(N+R))$	$\Theta(W(N+R))$	$\Theta(N+R)$ 不同的 `Digit` 可以复用。	No	Yes

Radix Sort – MSD 基数排序	填补方式同上。 从最高位开始调用 `Counting Sort`，如果最高位已经可以判断排序结果，就不需要继续调用。 否则，对最高位元素进行分组，每组内各自调用 `Counting Sort`。 最后将结果合并起来。	$\Theta(N+R)$ 首位排序就可以确定元素大小的情况，比如 $[1234, 2426, 4523, 3234]$ 只需要比较首位即可。	$\Theta(W(N+R))$	$\Theta(N+WR)$	No	Yes

N: Number of keys. R: Size of alphabet. W: Width of longest key.
*: Assumes constant compareTo time.

Heapification

Method	Algorithmic Logic	Best Case Runtime	Worst Case Runtime

Bottom-up Heapification	反向遍历 `Heap Array`，对每个元素调用 `sink()` 使得以这个元素为根元素的 `Sub Heap` 满足 `Heap Invariant`。	$\Theta(N)$，此时每一个 `sink()` 操作都没有 `swap` 的介入。	$\Theta(N)$ 推导如下, 假设 `Array Size` 为 N，则我们一共有 log_2N 层。我们假设 `Leaf Nodes` 层为 $i=0$，则第 i 层有 2^{log_2N-i-1} 个节点，每个节点在 `sink()` 的过程中至多需要 i 次 `swap` 操作，所以总共的 `swap` 操作是: $$\begin{aligned} \sum_{i=0}^{log_2N} i \times 2^{log_2N-i-1} &= \sum_{i=0}^{log_2N} i \times \frac{N}{2^{i+1}} \\ &= \frac{N}{4}\sum_{i=0}^{log_2N} i \times (\frac{1}{2})^{i-1} \end{aligned}$$ 令 $x = \frac{1}{2}$，则: $$\begin{aligned} \frac{N}{4}\sum_{i=0}^{log_2N} i \times (\frac{1}{2})^{i-1} &= \frac{N}{4}\sum_{i=0}^{log_2N} i(x)^{i-1} \\ &= \frac{N}{4}\frac{d}{dx}\sum_{i=0}^{log_2N} x^i \\ &\leq \frac{N}{4}\frac{d}{dx}\sum_{i=0}^{\infty} x^i \\ &= \frac{N}{4}\frac{d}{dx}\frac{1}{(1-x)} \\ &= \frac{N}{4}\frac{1}{(1-(\frac{1}{2}))^2} \\ &= \Theta(N) \end{aligned}$$

Top-down Heapification	逐个插入数组中的元素进入 `Heap` 。	$\Theta(N)$ ，所有元素都相同的时候，这样在 `swim` 的时候不需要任何 `swap` 操作。	$\Theta(NlogN)$ 推导如下，我们假设从上开始，第 i 层($i \geq 0$)的每个新插入元素在 `swim` 的过程中总共需要 i 次 `swap` 操作，而第 i 层总共有 2^i 个元素，所以每一层需要 $i \times 2^i$ 次 `swap` 操作，而我们一共有 $log_2 N$ 层，所以时间复杂度是: $$\sum_{i=0}^{log_2 N} i \times 2^i \leq log_2 N \times \sum_{i=0}^{log_2 N} 2^i = \Theta(NlogN)$$ 因为 $\sum_{i=0}^{log_2 N} \in \Theta(NlogN)$ 。

总的来说， `Heapfication` 操作的算法复杂度是 $\Omega(N)$ 。因为至少需要遍历所有的元素，但是 `swap` 操作的多少视元素自身的性质决定。

另外， `Heapfication` 操作的算法复杂度是 $O(NlogN)$ ，这对应了 `Top-down Heapification` 中的 `Worst Case Runtime` 。

Algorithmic Lower Bound

Comparison Sort

所有的 `Comparison Sort` 的 `Lower Bound` 是 $\Theta(NlogN)$

Notice that the worst-case runtime in the comparison sorts on an N element array listed above are lower bounded by $\Theta(N\log N)$. Can there be a sort that runs faster than $\Theta(N\log N)$ in the worst-case?

Yes, if we can avoid sorts that require comparisons, otherwise no. Given N elements, there are $N!$ possible permutations. Using a comparison sort, we will need at least $log_2(N!) \in \Omega(N\log N)$ comparisons. This is because one comparison could eliminate at most half of the possible permutations–when comparing two elements A and B, if you decide A should come first, then you've eliminated all the other permutations where B came first. However, with counting sorts, we can avoid the need for comparisons, and get a runtime that is linear with respect to the number of elements in the list, though its runtime is greatly dependent on other factors like radix and word size.

完整证明方法如下:

1. 因为对于一个含有 N 个元素的数组来说，元素有 $N!$ 种排列方式，每一次比较都能排除一般的排列可能性，所以我们至少需要 $log_2(N!)$ 次比较来实现排序，如果我们的 `Cost Model` 是 `Number of Comparison` 的话，那么算法复杂度就至少是 $log_2(N!)$ 。
2. 证明 $log_2(N!) \in \Omega(NlogN)$ ：因为 $N! > (\frac{N}{2})^{\frac{N}{2}}$ ，所以两边取 `log` 可得 $log(N!) > \frac{N}{2}log(\frac{N}{2})$ ，所以 $log_2(N!) \in \Omega(NlogN)$ 。
3. 证明 $log_2(N!) \in O(NlogN)$ ：因为
$$log_2(N!) = log_2(N) + log_2(N-1) + \cdots + log_2(1) < log_2(N) + log_2(N) + \cdots + log_2(N) = NlogN$$
，所以 $log_2(N!) \in O(NlogN)$
4. 综上所述， $log_2(N!) \in \Theta(NlogN)$

所以所有的 `Comparison Sort` 算法的复杂度至少是 $log_2(N!)$ ，也就是 $\Theta(NlogN)$ 。