Sorting

Selection Sort

- for each index find the smallest remaining element.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} n-1-(i+1)+1$$

$$= \sum_{i=0}^{n-2} n-1-i \qquad (n-1)+(n-2)+(n-3)+\ldots+2+1$$

$$= \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \in \Theta(n^2)$$

SELECTIONSORT(A)

```
1  for i = 0 to A.length − 2
2        min = i
3        for j = i + 1 to A.length − 1
4              if A[j] < A[min]
5                    min = j
6        exchange A[i] and A[min]
```

Input Size: number of elements in A

$n = A.length$

Basic op? comparison between $A[j]$ and $A[min]$

Best/Worst/Average? No

In-place

An in-place sorting algorithm does not require a linear amount of additional memory.

Stable Sorting

A sorting algorithm is stable if whenever there are two elements with the same key, R and S, in the input with R appearing before S in the input array. R will appear before S in the output array.

INSERTIONSORT(A)

```
1  for j = 1 to A.length − 1
2      key = A[j]
3      i = j − 1
4      while i ≥ 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

Input size? n = A.length

Basic op? comparison between A[i] and key

Best/Worst/Average?

Best: Already sorted array

Worst: Reverse sorted array

Insertion Sort

- insert an element into the correct position in an already sorted list

Best case complexity

$$\sum_{j=1}^{n-1} 1 = n-1 \in \Theta(n)$$

Worst case complexity

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Average case

$$\Theta(n^2)$$

Merge Sort

- it is easier to combine two sorted arrays than it is to sort the whole array.

```
MERGESORT(A)
1  n = A.length
2  if n = 1
3        return A
4  mid = ⌊n/2⌋
5  B = MERGESORT(A[0...mid − 1])
6  C = MERGESORT(A[mid...n − 1])
7  return MERGE(B, C)
```

Input size? $n = A.length$

Basic op? comparison between $B[i]$ and $C[j]$

$$T(n) = 2T\left(\frac{n}{2}\right) + an$$

$$T(1) = 0$$

```
MERGE(B, C)
1   i, j = 0
2   A = []
3   while i < B.length and j < C.length
4       if B[i] ≤ C[j]
5           A.add(B[i])
6           i = i + 1
7       else
8           A.add(C[j])
9           j = j + 1
10  if i = B.length
11      A.add(C[j...])
12  else
13      A.add(B[i...])
14  return A
```

Best: $\frac{n}{2}$ $\in \Theta(n)$

worst: $n-1$

$$T(n) = 2T\left(\frac{n}{2}\right) + an$$

$$T(1) = 0$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + a\frac{n}{2}$$

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + a\frac{n}{2}\right] + an$$

$$= 4T\left(\frac{n}{4}\right) + an + an$$

$$= 8T\left(\frac{n}{8}\right) + 3an$$

$$= 16T\left(\frac{n}{16}\right) + 4an$$

$$= 2^i T\left(\frac{n}{2^i}\right) + ian$$

$$= 2^{\lg(n)} T(1) + an \lg n$$

$$= an \lg n \in \Theta(n \log n)$$

Problem of the day

Pancake Sorting

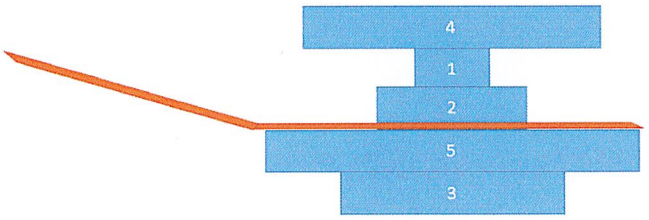
- Given an array of integers, sort the array using a given **Flip** operation: $flip(array, end)$
  $\Rightarrow$ reverse the array elements from 0 to end (inclusive).
- Example: consider the array $A = [4, 1, 2, 5, 3]$

4	1	2	5	3

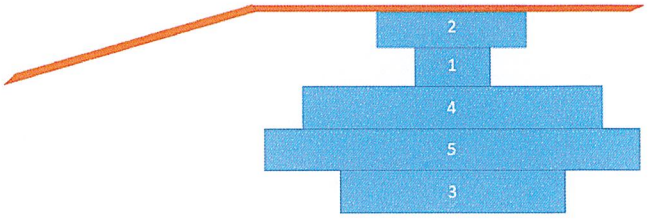
$\rightarrow flip(A, 2) \rightarrow$

2	1	4	5	3

- This is called Pancake Sorting because the **Flip** operation is analogous to flipping pancakes.



is flipped into



```
PancakeSort(A)
1  for i = n − 1 to 0
2      maxIndex = i
3      for j = 0 to i
4          if A[j] ≥ A[maxIndex]
               // Find the index of the largest remaining element
5              maxIndex = j
       // Flip the largest remaining element to the top
6      Flip(A, maxIndex)
       // Flip the largest remaining element to the correct position
7      Flip(A, i)
8  return A
```