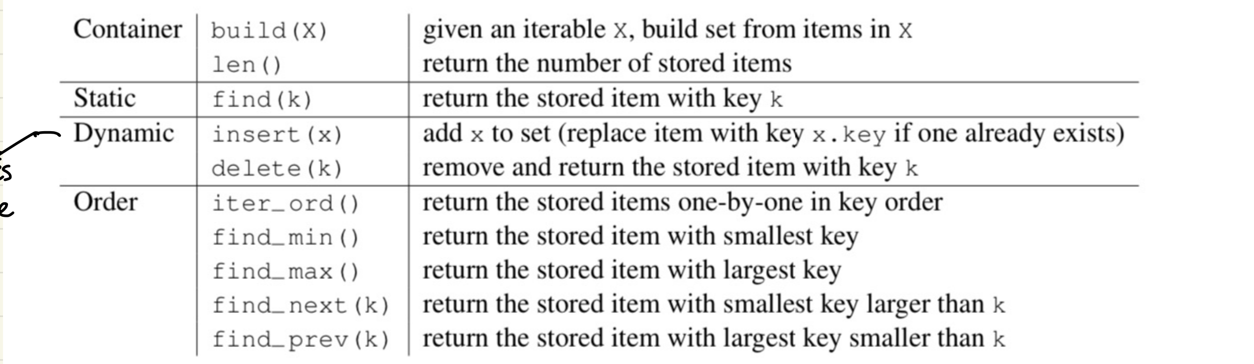
Lecture 3: Mon, April 29, 2024

* Sets

(A) Set Interface → "program specification, what it does (not how)"



Container	build(X)
	len()
		given an iterable X, build set from items in X
		return the number of stored items

Static	find(k)	return the stored item with key k

Dynamic	insert(x)
	delete(k)
		add x to set (replace item with key x.key if one already exists)
		remove and return the stored item with key k

Order	iter_ord()
	find_min()
	find_max()
	find_next(k)
	find_prev(k)
		return the stored items one-by-one in key order
		return the stored item with smallest key
		return the stored item with largest key
		return the stored item with smallest key larger than k
		return the stored item with largest key smaller than k

changes the size

(B) Data Structures implementing Sets

Approach 1: Store it in an Unorded Array. → no rule based on key in which to store x.

↳ Highly ineficient. find(x) $\in O(n)$, since we have to iterate through the entire thing.

↳ In fact all operations of the interface will take $O(n)$. by the same argument.

Approach 2: Store it in a key-ordered Array.

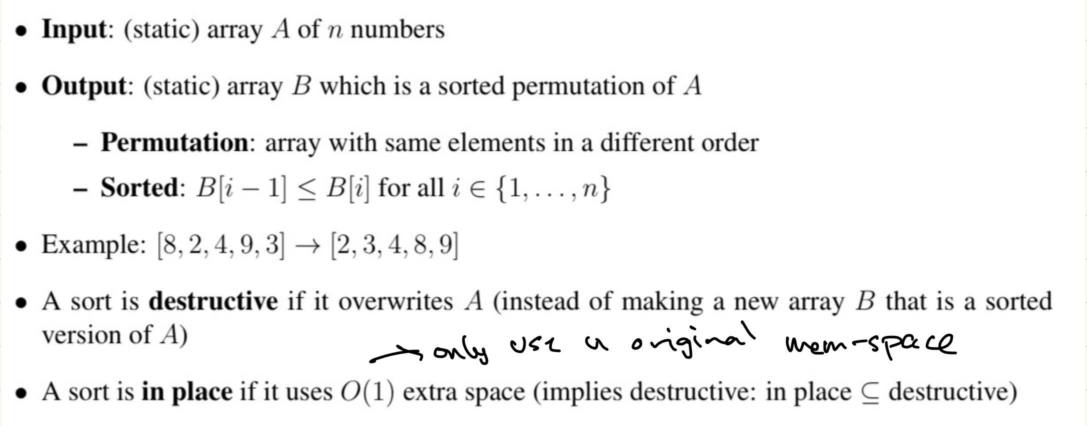
↳ improves find-min & find_max @ $O(1)$ since u just get 1st or last element.

↳ find(x) also is improved to $\log(n)$ by implementing binary-search. (same w/ find-prev/next)

↳ Although build(x) becomes less efficient @ $O(\underbrace{n}_{\text{store}}\underbrace{\log(n)}_{\text{sort}})$

↳ Hence how to construct a sorted array efficiently?

(C) Sorting



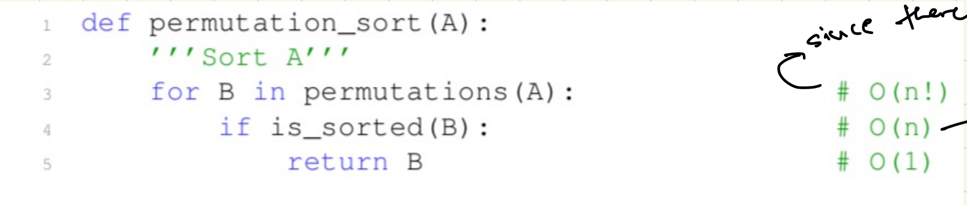
- **Input**: (static) array A of n numbers
- **Output**: (static) array B which is a sorted permutation of A
  - **Permutation**: array with same elements in a different order
  - **Sorted**: $B[i-1] \le B[i]$ for all $i \in \{1, \ldots, n\}$
- Example: $[8, 2, 4, 9, 3] \to [2, 3, 4, 8, 9]$
- A sort is **destructive** if it overwrites A (instead of making a new array B that is a sorted version of A)

→ only use u original mem-space

- A sort is **in place** if it uses $O(1)$ extra space (implies destructive: in place $\subseteq$ destructive)

Sorting Algo 1: Permutation Sort

↳ list every possible perm, check each perm if it's sorted.



```
def permutation_sort(A):
    '''Sort A'''
    for B in permutations(A):           # O(n!)
        if is_sorted(B):                # O(n)
            return B                    # O(1)
```

since there are n! perms.

loop through this perm.

↳ very inefficient! $\in \Omega(n \cdot n!)$ → exponential!

↳ omega instead of big-O because we're not sure how the permutations (A) is implemented but we know it should take at least n! b.c. there are n! perms.

↳ meaning can be worse...

Sorting Algo 2: Selection Sort

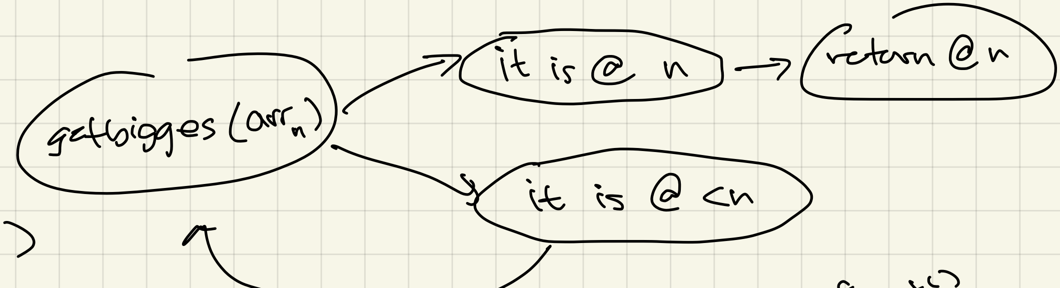
↳ loops thru array and selects highest element each pass through and puts it into a new (or not) ordered array.

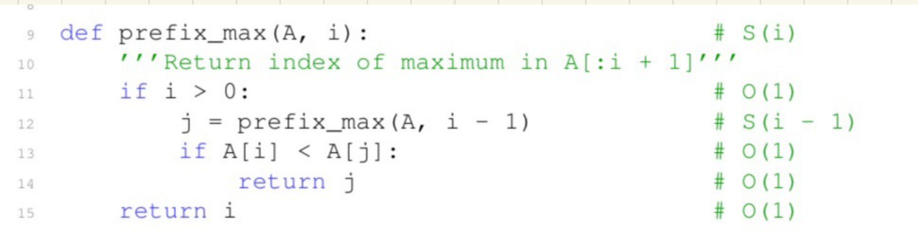
↳ can do it in-place: delete, insert. (or swap)

① Find biggest w/ index ≤ n,
② swap to n.
③ sort 1 ... n−1

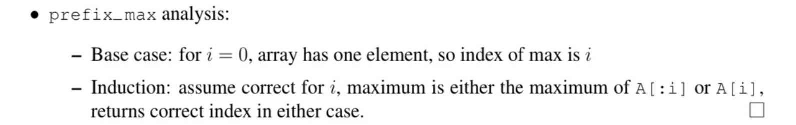
looping shorter & shorter arr.

recursive implementation (theoretical only don't do this!)





```
def prefix_max(A, i):                                # S(i)
    '''Return index of maximum in A[:i + 1]'''
    if i > 0:                                        # O(1)
        j = prefix_max(A, i - 1)                     # S(i - 1)
        if A[i] < A[j]:                              # O(1)
            return j                                 # O(1)
    return i                                         # O(1)
```



- prefix_max analysis:
  - Base case: for $i = 0$, array has one element, so index of max is i
  - Induction: assume correct for i, maximum is either the maximum of A[:i] or A[i], returns correct index in either case. □

Runtime of the selection_sort()

$S(\ ) = ?$

if $S(1) \longrightarrow \Theta(1)$

if $S(n) \longrightarrow S(n-1) + \Theta(1)$

$= ?$

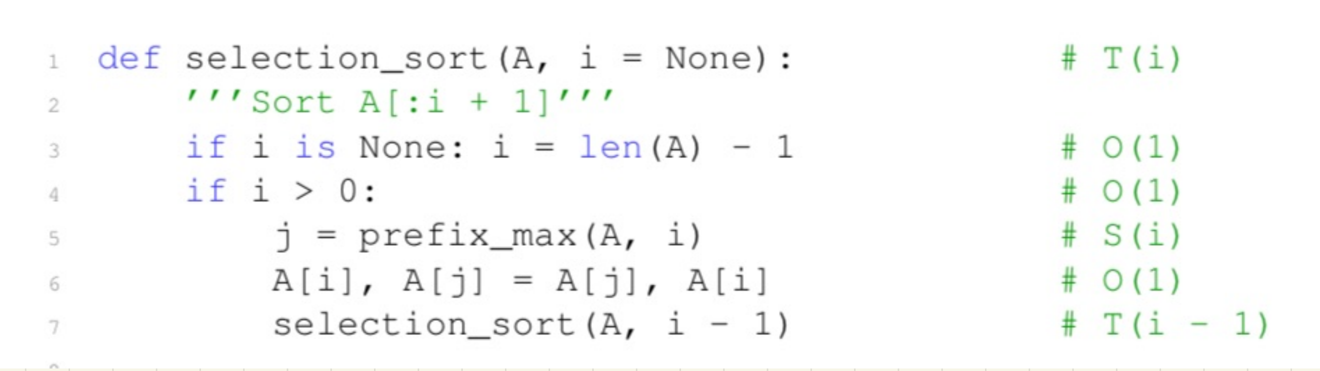
Try substitution method: $S(n) \overset{?}{=} cn$

subtract cn on both sides ↳

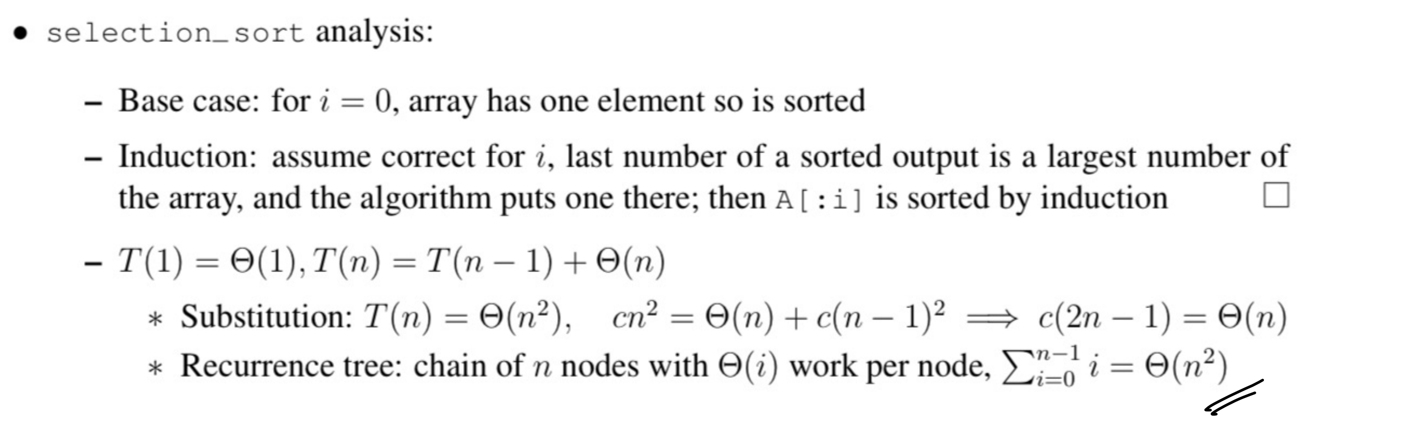
$cn = c(n-1) + \Theta(1)$

$0 = 1 + \Theta(1)$ ✓✓

then $S(n) = \Theta(n)$



```
def selection_sort(A, i = None):                  # T(i)
    '''Sort A[:i + 1]'''
    if i is None: i = len(A) - 1                  # O(1)
    if i > 0:                                     # O(1)
        j = prefix_max(A, i)                      # S(i)
        A[i], A[j] = A[j], A[i]                   # O(1)
        selection_sort(A, i - 1)                  # T(i - 1)
```



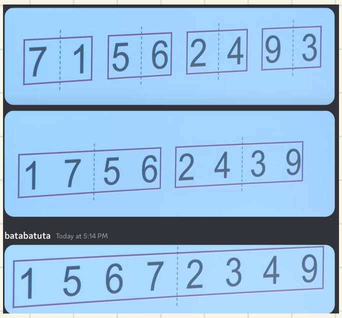
- `selection_sort` analysis:
  - Base case: for $i = 0$, array has one element so is sorted
  - Induction: assume correct for i, last number of a sorted output is a largest number of the array, and the algorithm puts one there; then `A[:i]` is sorted by induction □
  - $T(1) = \Theta(1), T(n) = T(n-1) + \Theta(n)$
    - Substitution: $T(n) = \Theta(n^2)$, $\quad cn^2 = \Theta(n) + c(n-1)^2 \implies c(2n-1) = \Theta(n)$
    - Recurrence tree: chain of n nodes with $\Theta(i)$ work per node, $\sum_{i=0}^{n-1} i = \Theta(n^2)$ //

Sorting Algo 3: Insertion Sort

↳ skipped ... similar to selection sort he said...

↳ check at home

Sorting Algo 4: Merge Sort



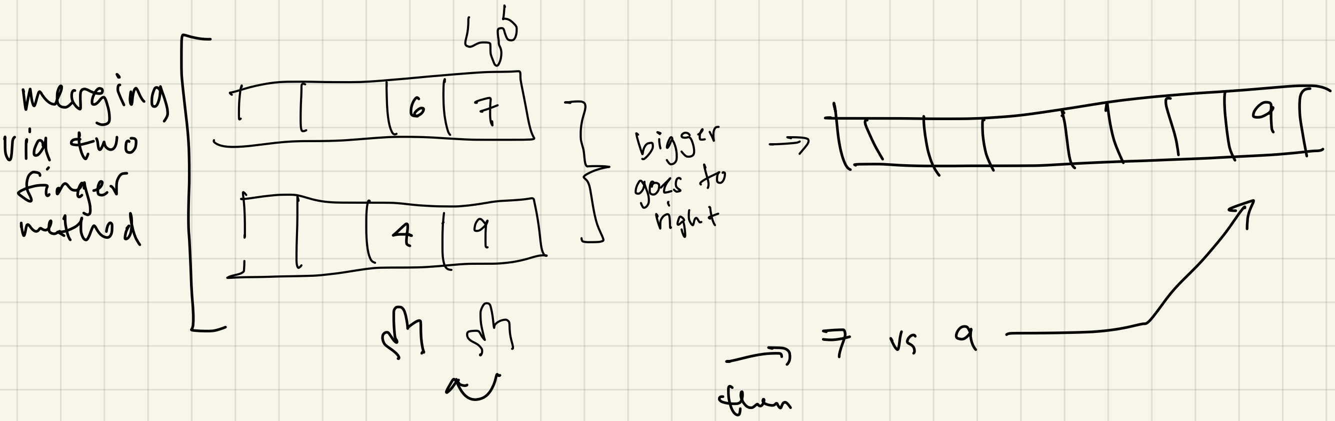
7 1 5 6 2 4 9 3

1 7 5 6 2 4 3 9

batabatuta Today at 5:14 PM

1 5 6 7 2 3 4 9

↳ take pairs and merge sort elements in that pair. Then take pairs of pairs and merge_sort the pair of pairs and so on...



then

→ 7 vs 4 (hence 7 goes to right after 9).

```
def merge_sort(A, a = 0, b = None):                  # T(b - a = n)
    '''Sort A[a:b]'''
    if b is None: b = len(A)                         # O(1)
    if 1 < b - a:                                    # O(1)
        c = (a + b + 1) // 2                         # O(1)
        merge_sort(A, a, c)                          # T(n / 2)
        merge_sort(A, c, b)                          # T(n / 2)
        L, R = A[a:c], A[c:b]                        # O(n)
        merge(L, R, A, len(L), len(R), a, b)         # S(n)

def merge(L, R, A, i, j, a, b):                      # S(b - a = n)
    '''Merge sorted L[:i] and R[:j] into A[a:b]'''
    if a < b:                                        # O(1)
        if (j <= 0) or (i > 0 and L[i - 1] > R[j - 1]): # O(1)
            A[b - 1] = L[i - 1]                      # O(1)
            i = i - 1                                # O(1)
        else:                                        # O(1)
            A[b - 1] = R[j - 1]                      # O(1)
            j = j - 1                                # O(1)
        merge(L, R, A, i, j, a, b - 1)               # S(n - 1)
```

- merge analysis:
  - Base case: for $n = 0$, arrays are empty, so vacuously correct
  - Induction: assume correct for n, item in A[r] must be a largest number from remaining prefixes of L and R, and since they are sorted, taking largest of last items suffices; remainder is merged by induction □
  - $S(0) = \Theta(1), S(n) = S(n-1) + \Theta(1) \implies S(n) = \Theta(n)$ (linear time to merge...)
- merge_sort analysis: → we call merge_sort twice each time on a list that's half the size.
  - Base case: for $n = 1$, array has one element so is sorted
  - Induction: assume correct for $k < n$, algorithm sorts smaller halves by induction, and then merge merges into a sorted array as proved above. □
  - $T(1) = \Theta(1), T(n) = 2T(n/2) + \Theta(n)$
    - Substitution: Guess $T(n) = \Theta(n \log n)$
      $cn \log n = \Theta(n) + 2c(n/2)\log(n/2) \implies cn\log(2) = \Theta(n)$
      $= cn(\log n - \log 2) + \Theta(n)$
    - Recurrence Tree: complete binary tree with depth $\log_2 n$ and n leaves, level i has 2^i nodes with $O(n/2^i)$ work each, total: $\sum_{i=0}^{\log_2 n}(2^i)(n/2^i) = \sum_{i=0}^{\log_2 n} n = \Theta(n \log n)$