Lecture 3: Sorting

Set Interface (L03-L08)

For python

Container	`build(X)`	given an iterable `X`, build set from items in `X`
	`len()`	return the number of stored items
Static	`find(k)`	return the stored item with key `k`
Dynamic	`insert(x)`	add `x` to set (replace item with key `x.key` if one already exists)
	`delete(k)`	remove and return the stored item with key `k`
Order	`iter_ord()`	return the stored items one-by-one in key order
	`find_min()`	return the stored item with smallest key
	`find_max()`	return the stored item with largest key
	`find_next(k)`	return the stored item with smallest key larger than `k`
	`find_prev(k)`	return the stored item with largest key smaller than `k`

- Storing items in an array in arbitrary order can implement a (not so efficient) set
- Stored items sorted increasing by key allows:
  - faster find min/max (at first and last index of array)
  - faster finds via binary search: $O(\log n)$

Set	Operations $O(\cdot)$				
Data Structure	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$

- But how to construct a sorted array efficiently?

Sorting

- Given a sorted array, we can leverage binary search to make an efficient set data structure.
- **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**: $B[i-1] \leq B[i]$ for all $i \in \{1, \ldots, n\}$
- Example: $[8, 2, 4, 9, 3] \rightarrow [2, 3, 4, 8, 9]$
- A sort is **destructive** if it overwrites A (instead of making a new array B that is a sorted version of A)
- A sort is **in place** if it uses $O(1)$ extra space (implies destructive: in place $\subseteq$ destructive)

Permutation Sort

- There are $n!$ permutations of A, at least one of which is sorted
- For each permutation, check whether sorted in $\Theta(n)$
- Example: $[2, 3, 1] \rightarrow \{[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]\}$

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

- `permutation_sort` analysis:
  - Correct by case analysis: try all possibilities (Brute Force)
  - Running time: $\Omega(n! \cdot n)$ which is **exponential** :(

Solving Recurrences

equations that discribes the running time of recursive algorithms

- **Substitution**: Guess a solution, replace with representative function, recurrence holds true
- **Recurrence Tree**: Draw a tree representing the recursive calls and sum computation at nodes
- **Master Theorem**: A formula to solve many recurrences (R03)

time complexity of a problem of size n terms

Selection Sort

- Find a largest number in prefix `A[:i + 1]` and swap it to `A[i]`
- Recursively sort prefix `A[:i]`
- Example: $[8, 2, 4, 9, 3]$, $[8, 2, 4, 3, 9]$, $[3, 2, 4, 8, 9]$, $[3, 2, 4, 8, 9]$, $[2, 3, 4, 8, 9]$

$1+2+\cdots+n$
$=\Theta(n^2)$

```
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)
```

To find the biggest element.

```
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:
  - 1 element in the array
  - 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. either is in the end or not. □
  - run time; order $= n$
  - $S(1) = \Theta(1), S(n) = S(n-1) + \Theta(1)$
    * Substitution: $S(n) = \Theta(n), \quad cn = \Theta(1) + c(n-1) \implies 1 = \Theta(1)$
    * Recurrence tree: chain of n nodes with $\Theta(1)$ work per node, $\sum_{i=0}^{n-1} 1 = \Theta(n)$
- `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, $\sum_{i=0}^{n-1} i = \Theta(n^2)$

Insertion Sort

- Recursively sort prefix `A[:i]`
- Sort prefix `A[:i + 1]` assuming that prefix `A[:i]` is sorted by repeated swaps
- Example: $[8, 2, 4, 9, 3]$, $[2, 8, 4, 9, 3]$, $[2, 4, 8, 9, 3]$, $[2, 4, 8, 9, 3]$, $[2, 3, 4, 8, 9]$

```
def insertion_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)
        insertion_sort(A, i - 1)              # T(i - 1)
        insert_last(A, i)                     # S(i)

def insert_last(A, i):                        # S(i)
    '''Sort A[:i + 1] assuming sorted A[:i]'''
    if i > 0 and A[i] < A[i - 1]:             # O(1)
        A[i], A[i - 1] = A[i - 1], A[i]       # O(1)
        insert_last(A, i - 1)                 # S(i - 1)
```

- `insert_last` analysis:
  - Base case: for $i = 0$, array has one element so is sorted
  - Induction: assume correct for i, if `A[i] >= A[i - 1]`, array is sorted; otherwise, swapping last two elements allows us to sort `A[:i]` by induction $\square$
  - $S(1) = \Theta(1), S(n) = S(n-1) + \Theta(1) \implies S(n) = \Theta(n)$
- `insertion_sort` analysis:
  - Base case: for $i = 0$, array has one element so is sorted
  - Induction: assume correct for i, algorithm sorts `A[:i]` by induction, and then `insert_last` correctly sorts the rest as proved above $\square$
  - $T(1) = \Theta(1), T(n) = T(n-1) + \Theta(n) \implies T(n) = \Theta(n^2)$

Pair in boxes of 2 elements
Sort inside of the box
pair in boxes of 4 elements
sort inside of the box
⋮

pair in 2^n elements
sort inside the box
sort each (recursively)

Merge Sort

- Recursively sort first half and second half (may assume power of two) The time spended is in $T(n) = O(n \log n)$
- Merge sorted halves into one sorted list (two finger algorithm)
- Example: $[7, 1, 5, 6, 2, 4, 9, 3]$, $[1, 7, 5, 6, 2, 4, 3, 9]$, $[1, 5, 6, 7, 2, 3, 4, 9]$, $[1, 2, 3, 4, 5, 6, 7, 9]$

```
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 □
  - $S(0) = \Theta(1), S(n) = S(n-1) + \Theta(1) \implies S(n) = \Theta(n)$
- `merge_sort` analysis: ↳ worst-case - already sorted
  - 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 $T(n) = \Theta(n \log n)$
      $cn \log n = \Theta(n) + 2c(n/2)\log(n/2) \implies cn \log(2) = \Theta(n)$
    - ∗ 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: $\sum_{i=0}^{\log_2 n}(2^i)(n/2^i) = \sum_{i=0}^{\log_2 n} n = \Theta(n \log n)$

Sets and sorting

Interface: collections of operations.
Data structure: way to store data that supports a set of operations.
Always have tradeoffs - the goal or the circumstances chooses the sorting.

SET - A set interface is a “container” of objects that is dynamic you can keep editing or adding.
set = all of students (key) that are associated with a number ID (satellite data)

build(A)- given an iterable A, build a sequence from items in A.
len() - return the number of stored items.
find(K)- return the stored number with key k - does k exist in the list - is personalizable by using this structure _min
delete(k)- remove or return key

SORTING -
items sorted by key - smallest id number to the highest.
binary search - look left or the right takes log(n) time.

Destructive - overwrites the input array
In place - Uses O(1) extra space.

input - array of n numbers/ keys A
output - sorted array B

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	build(X)	find(k)	insert(x) delete(k)	find_min() find_max()	find_prev(k) find_next(k)
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$

Permutation sort : Enumerate the permutations - □(n!)
check if permutations is sorted

for I = 1 to n-1 // to the whole list -
b[i]<= b[i+1] // see if what you have is lower them what you see

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

Selection Sort -
1- Find item (i) in the list index <= 1
2- Swap
3- sort list
82493
find the biggest and strike to the end by swiping the 9 and 3
8 2 4 3 | 9
swap 8 with 3, so on...

1-