

## Lecture 3: Sorting

### Sorting

- Given a sorted array, we can leverage binary search to make an efficient set data structure.
- But how can we sort an array?
- **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  $i \in \{1, \dots, n\}$
- Example:  $[8, 2, 4, 9, 3] \rightarrow [2, 3, 4, 8, 9]$
- A sort is **destructive** if  $B = A$
- A sort is **in place** if it uses  $O(1)$  extra space (implies 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:  $[2, 3, 1] \rightarrow \{[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]\}$

```

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

```

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

### Solving Recurrences

- **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 (R02)

## 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 def selection_sort(A, i = None):                # T(i)
2     '''Sort A[:i + 1]'''
3     if i is None: i = len(A) - 1                # O(1)
4     if i > 0:                                    # O(1)
5         j = prefix_max(A, i)                    # S(i)
6         A[i], A[j] = A[j], A[i]                 # O(1)
7         selection_sort(A, i - 1)                # T(i - 1)
8
9 def prefix_max(A, i):                            # S(i)
10    '''Return index of maximum in A[:i + 1]'''
11    if i > 0:                                     # O(1)
12        j = prefix_max(A, i - 1)                # S(i - 1)
13        if A[i] < A[j]:                          # O(1)
14            return j                             # O(1)
15    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.  $\square$
  - $S(1) = \Theta(1), S(n) = S(n - 1) + \Theta(1)$ 
    - \* 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  $\square$
  - $T(1) = \Theta(1), T(n) = T(n - 1) + \Theta(n)$ 
    - \* 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:  $[8, 2, 4, 9, 3]$ ,  $[2, 8, 4, 9, 3]$ ,  $[2, 4, 8, 9, 3]$ ,  $[2, 4, 8, 9, 3]$ ,  $[2, 3, 4, 8, 9]$

```

1 def insertion_sort(A, i = None):                # T(i)
2     '''Sort A[:i + 1]'''
3     if i is None: i = len(A) - 1                # O(1)
4     if i > 0:                                    # O(1)
5         insertion_sort(A, i - 1)                # T(i - 1)
6         insert_last(A, i)                        # S(i)
7
8 def insert_last(A, i):                           # S(i)
9     '''Sort A[:i + 1] assuming sorted A[:i]'''
10    if i > 0 and A[i] < A[i - 1]:                 # O(1)
11        A[i], A[i - 1] = A[i - 1], A[i]          # O(1)
12        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  $A[i] \geq A[i - 1]$ , 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  $\square$
  - $T(1) = \Theta(1), T(n) = T(n - 1) + \Theta(n) \implies T(n) = \Theta(n^2)$

## Merge Sort

- Recursively sort first half and second half (may assume power of two)
- 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]

```

1 def merge_sort(A, a = 0, b = None):                                # T(b - a = n)
2     '''Sort A[a:b]'''
3     if b is None: b = len(A)                                       # O(1)
4     if 1 < b - a:                                                  # O(1)
5         c = (a + b + 1) // 2                                       # O(1)
6         merge_sort(A, a, c)                                         # T(n / 2)
7         merge_sort(A, c, b)                                         # T(n / 2)
8         L, R = A[a:c], A[c:b]                                       # O(n)
9         merge(L, R, A, len(L), len(R), a, b)                       # S(n)
10
11 def merge(L, R, A, i, j, a, b):                                    # S(b - a = n)
12     '''Merge sorted L[:i] and R[:j] into A[a:b]'''
13     if a < b:                                                       # O(1)
14         if (j <= 0) or (i > 0 and L[i - 1] > R[j - 1]):          # O(1)
15             A[b - 1] = L[i - 1]                                     # O(1)
16             i = i - 1                                              # O(1)
17         else:                                                        # O(1)
18             A[b - 1] = R[j - 1]                                     # O(1)
19             j = j - 1                                              # O(1)
20         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  $\square$
  - $S(0) = \Theta(1), S(n) = S(n - 1) + \Theta(1) \implies S(n) = \Theta(n)$
- merge\_sort analysis:
  - 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.  $\square$
  - $T(1) = \Theta(1), T(n) = 2T(n/2) + \Theta(n)$ 
    - \* 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:  $\sum_{i=0}^{\log_2 n} (2^i)(n/2^i) = \sum_{i=0}^{\log_2 n} n = \Theta(n \log n)$