CSC111 Lecture 19: Recursive Sorting Algorithms, Part 1

Hisbaan Noorani

March 22, 2021

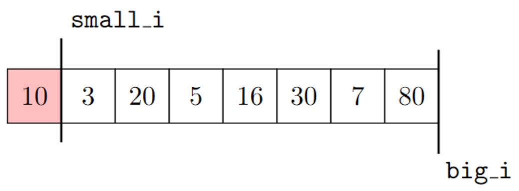
Contents

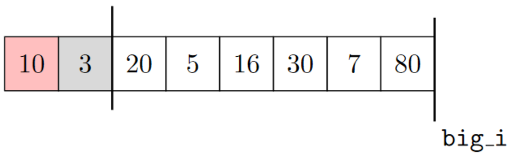
1 Exercise 1: In-place partitioning

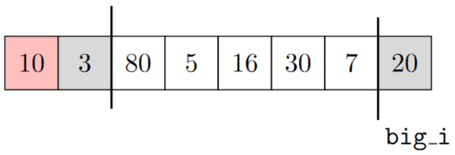
We're going to study how to implement an *in-place* version of `_partition` that mutates `lst` directly, without creating any new lists. This is the key step to implementing an in-place version of quicksort.

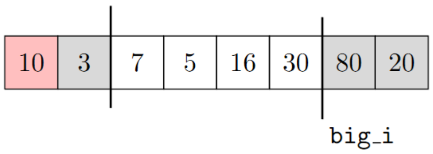
```python
def _in_place_partition(lst: list) -> None:
    """Mutate lst so that it is partitioned with pivot lst[0].

    Let pivot = lst[0]. The elements of lst are moved around so that lst looks like

        [x1, x2, ... x_m, pivot, y1, y2, ... y_n],

    where each of the x's is <= pivot, and each of the y's is > pivot.

    >>> lst = [10, 3, 20, 5, 16, 30, 7, 100]
    >>> _in_place_partition(lst)  # pivot is 10
    >>> lst[3]  # the 10 is now at index 3
    10
    >>> set(lst[:3]) == {3, 5, 7}
    True
    >>> set(lst[4:]) == {16, 20, 30, 100}
    True
    """
```

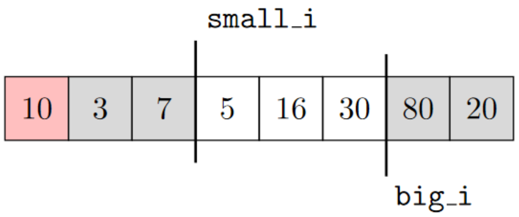
For your reference, there are the diagrams of the example we saw in lecture:

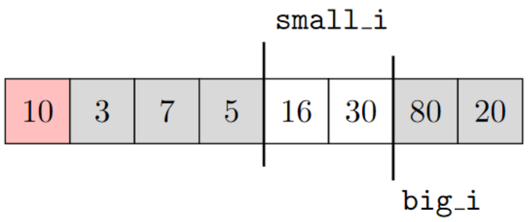


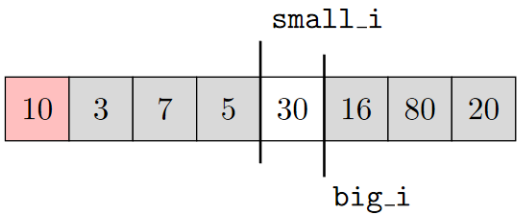

small_i
10
3
20
5
16
30
7
80
big_i

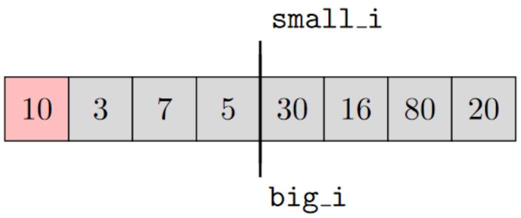

small_i
10
3
80
5
16
30
7
20
big_i

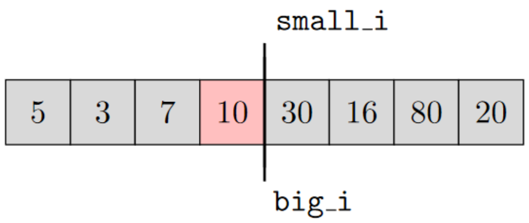

small_i
10
3
7
5
16
30
80
20
big_i


small_i
10
3
7
5
16
30
80
20
big_i


small_i
10
3
7
5
16
30
80
20
big_i


small_i
10
3
7
5
30
16
80
20
big_i


small_i
10
3
7
5
30
16
80
20
big_i



1. The key idea to implement this function is to divide up `lst` into three parts using indexes `small_i` and `big_i`:

   - `lst[1:small_i]` contains the elements that are known to be < `lst[0]` (the "smaller partition")
   - `lst[big_i:]` contains the elements that are known to be > `lst[0]` (the "bigger partition")
   - `lst[small_i:big_i]` contains the elements that have not yet been compared to the pivot (the "unsorted section")

   `_in_place_partition` uses a loop to go through the elements of the list and compare each one to the pivot, building up either the "smaller" or the "larger partition" and shrinking the "unsorted section."

   (a) When we first begin the algorithm, we know that `lst[0]` is the pivot, but have not yet compared it against any other list element. What should the initial values of `small_i` and `big_i` be?

   `small_i = 1`

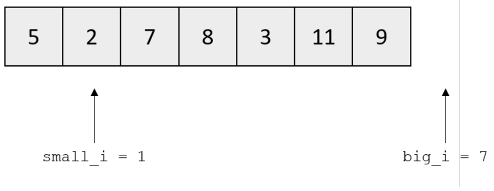
   `big_i = len(lst)`

   (b) We need to check the items one at a time, until every item has been compared against the pivot. What is the relationship between `small_i` and `big_i` when we have finished checking every item?

   Once we have finished checking every single item, `small_i = big_i`. If we were to do some sort of recursive implementation (which we don't need to do) then this would likely be our base case.

   (c) On each loop iteration, the element `lst[small_i]` is compared to the pivot.

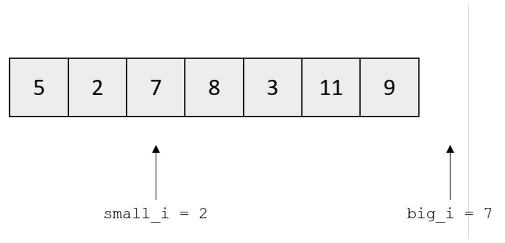
   - Suppose `lst[small_i]` is less than or equal to the pivot, as in the diagram below.

   Which partition should `lst[small_i]` belong to? Write the code to update `small_i` to add this element to the correct the partition.
   `lst[small_i]` should be a part of the < partition. In this case, we will only increment the `small_i` variable.

```python
small_i += 1
```

- Now suppose lst[small_i] is greater than the pivot, as in the diagram below.



Which partition should lst[small_i] belong to? Write the code to swap this element and update small_i or big_i to add this element to the correct partition.
lst[small_i] should be a part of the > partition. In this case, we will swap the items.

```python
lst[small_i], lst[big_i - 1] = lst[big_i - 1], lst[small_i]
big_i -= 1
```

2. Next, implement _in_place_partition. Make sure you're confident in your answers to the previous questionsbefore writing the main loop!

```python
def _in_place_partition(lst: list) -> None:
    # Initialize variables
    pivot = lst[0]
    small_i = 1
    big_i = len(lst)

    # The main loop
    while small_i < big_i:
        # Loop invariants
        assert all(lst[j] <= pivot for j in range(1, small_i))
        assert all(lst[j] > pivot for j in range(big_i, len(lst)))

        if lst[small_i] <= pivot:
            # Increase the "smaller" partition
            small_i +=1
        else:
            # Swap lst[small_i] to back and increase the "bigger" partition
            lst[small_i], lst[big_i - 1] = lst[big_i], lst[small_i]
            big_i -= 1

    # At this point, all elements have been compared.
    # The final step is to move the pivot into its correct position in the list
    # (after the "smaller" partition, but before the "bigger" partition).
    lst[0], lst[small_i - 1] = lst[small_i - 1], lst[0]
    # Could have used big_i as well, since they're equal
```

3. We're going to need to make this helper a bit more general in order to use it in our a modified quicksort algorithm. If you still have time, complete the following modifications to your implementation of `_in_place_partition`.

   (a) Modify your implementation so that it returns the *new index of the pivot* in the list (so that when `_in_place_partition` is called in quicksort, we know where the partitions start and end).

```python
def _in_place_partition(lst: list) -> None:
    # Initialize variables
    pivot = lst[0]
    small_i = 1
    big_i = len(lst)

    while small_i < big_i:
        assert all(lst[j] <= pivot for j in range(1, small_i))
        assert all(lst[j] > pivot for j in range(big_i, len(lst)))

        if lst[small_i] <= pivot:
            small_i +=1
        else:
            lst[small_i], lst[big_i - 1] = lst[big_i], lst[small_i]
            big_i -= 1

    lst[0], lst[small_i - 1] = lst[small_i - 1], lst[0]
    return small_i - 1
```

   (b) Modify your implementation so that it takes in two additional parameters `b` and `e`, so that the function only partitions the range `lst[b:e]` rather than the entire list.

```python
def _in_place_partition(lst: list, b: int, e: int) -> int:
    # Initialize variables
    pivot = lst[b]
    small_i = b + 1
    big_i = e

    while small_i < big_i:
        # assert all(lst[j] <= pivot for j in range(1, small_i))
        # assert all(lst[j] > pivot for j in range(big_i, len(lst)))

        if lst[small_i] <= pivot:
            small_i +=1
        else:
            lst[small_i], lst[big_i - 1] = lst[big_i], lst[small_i]
            big_i -= 1

    lst[b], lst[small_i - 1] = lst[small_i - 1], lst[b]
    return small_i - 1
```