CSC111 Lecture 18: Iterative Sorting Algorithms, Part 2

Hisbaan Noorani

March 17, 2021

Contents

1 Exercise 1: Implementing `_insert`

In lecture, we implemented the `insertion_sort` algorithm using a helper, `_insert`.

```python
def insertion_sort(lst: list) -> None:
    """Sort the given list using the insertion sort algorithm.
    """
    for i in range(len(lst)):
        # Invariants:
        # - lst[:i] is sorted
        _insert(lst, i)


def _insert(lst: list, i: int) -> None:
    """Move lst[i] so that lst[:i + 1] is sorted.

    Preconditions:
        - 0 <= i < len(lst)
        - lst[:i] is sorted

    >>> lst = [7, 3, 5, 2]
    >>> _insert(lst, 1)
    >>> lst
    [3, 7, 5, 2]
    """
    j = i
    while j == 0 or lst[j] < lst[j - 1]:
        # Do the swap
```

```
        lst[j], lst[j - 1] = lst[j - 1], lst[j]
        j = j - 1
```

One way to efficiently implement `_insert` is to repeatedly swap the element at index `i` with the one to its left until it reaches its correct spot in the sorted list.

2	5	7	9	**4**	... (other unsorted items)

2	5	7	**4**	9	... (other unsorted items)

2	5	**4**	7	9	... (other unsorted items)

2	**4**	5	7	9	... (other unsorted items)

2	**4**	5	7	9	... (other unsorted items)

Using this idea, implement the `_insert` helper function. *Hint*: this is similar to a function from this week's prep.

2 Exercise 2: Running-time analysis

Your implementation of `_insert` (or the one we saw in class) has a spread of running times, since the number of loop iterations depends on the value of `lst[i]` and the other list items before it. This means that we'll need to do a worst-case running-time analysis for our code.

1. Find (with analysis) a good asymptotic upper bound on the worst-case running time of `_insert`, in terms of n, the size of the input list, and/or i, the value of the second argument.

   Assume a reversed list. This means that the list will be in the form $[n, n-1, \ldots, 1, 0]$.

   This means that we will always need to swap a given element to the start of the list.

   This gives `_insert` an upper bound for a worst case running time of $\mathcal{O}(i)$

2. Find an input family for `_insert` whose asymptotic running time matches the upper bound you found in the previous question. To save time, you do *not* need to analyse the running time for this input family, just describe what the input family is. This lets you conclude a Theta bound for the worst-case running time of `_insert`.

   Let $i \in \mathbb{N}$ and assume $i < n$.

   Let `lst` $= [4, 3, 2, 1, 0]$

   This input family gives `_insert` a lower bound for a worst case running time of $\Omega(i)$

   This, combined with the upper bound gives a worst case running time of $\Theta(i)$

3. Now, refer to the `insertion_sort` implementation at the top of this exercise. Find (with analysis) a good asymptotic upper bound on the worst-case running time of `_insert`, in terms of n, the size of the input list.

   The `for` loop runs n times, for $i = 0, 1, \ldots, n-1$.

   Each iteration takes at most i steps.

   Since i is increasing each iteration, this leads to an upper bound for a worst case running time of $\mathcal{O}(n^2)$.

4. Finally, find an input family for `insertion_sort` whose asymptotic running time matches the upper bound you found in the previous question. To save time, you do *not* need to analyse the running time for this input family, just describe what the input family is.

   This lets you conclude a Theta bound for the worst-case running time of `insertion_sort`.

   Hint: look carefully at the property you used for the input family in Question 2, and try to pick a list so that this property holds for *every* index `i`.

   Let $n \in \mathbb{N}$.

   Let `lst` $= [n-1, n-2, \ldots, 1, 0]$.

   In this case, for every $i \in \mathbb{N}$ with $i < n$, `lst[i]` is always less than every element in `lst[:i]`.

   This leads to a lower bound for a worst case running time of $\Omega(n^2)$.

   This, in combination with the upper bound gives a worst case running time of $\Theta(n^2)$.

3 Exercise 3: Saving `key` values

Here is the start of a *memoized* version of insertion sort that we started in lecture. Your task: complete this algorithm by implementing a new `_insert_memoized` helper function.

Hint: this function is very similar to `_insert_key`, except for the places where `key` is called.

```python
def insertion_sort_memoized(lst: list, key: Optional[Callable] = None) -> None:
    """Sort the given list using the insertion sort algorithm.

    If key is not None, sort the items by their corresponding return value when passed to key.
    Use a dictionary to keep track of "key" values, so that the function is called only once per
    list element.

    Note that this is a *mutating* function.

    >>> lst = ['cat', 'octopus', 'hi', 'david']
    >>> insertion_sort_memoized(lst, key=len)
    >>> lst
    ['hi', 'cat', 'david', 'octopus']
    >>> lst2 = ['cat', 'octopus', 'hi', 'david']
    >>> insertion_sort_memoized(lst2)
    >>> lst2
    ['cat', 'david', 'hi', 'octopus']
    """
```

```python
    # Use this variable to keep track of the saved "key" values
    # across the different calls to _insert.
    key_values = {}

    for i in range(0, len(lst)):
        _insert_memoized(lst, i, key, key_values)

# Define the _insert_memoized helper below.
# Hint: You'll need to modify the _insert_key helper function to take an
# additional dictionary argument.

def _insert_memoized(lst: list, i: int, key: Optional[Callable] = None, key_values: dict) -> None:
    for j in range(i, 0, -1):  # This goes from i down to 1
        if key is None:
            if lst[j - 1] <= lst[j]:
                return
            else:
                # Swap lst[j - 1] and lst[j]
                lst[j - 1], lst[j] = lst[j], lst[j - 1]
        else:
            if lst[j - 1] not in key_values:
                key_values[lst[j - 1]] = key(lst[j - 1])
            if lst[j] not in key_values:
                key_values[lst[j]] = key(lst[j])

            if key_values[lst[j - 1]] <= key_values(lst[j]):
                return
            else:
                # Swap lst[j - 1] and lst[j]
                lst[j - 1], lst[j] = lst[j], lst[j - 1]#+end_src
```

4 Additional exercises

1. Implement “key” and memoized versions of the selection sort algorithm from last class.