11.1 - Week 11 - Applied - Theory

Objectives of this Applied Session

Objectives of this Applied Session

- To be able to implement multiple recursive functions
- To be able to analyse the functionality of recursive functions, and analyse their complexity
- To understand recursive sorting algorithms Quicksort and Mergesort
- To be able to analyse their complexity

Getting Familiar with Recursion

Question 1 *Submitted Oct 11th 2022 at 8:17:03 am*

Below is a jumbled up version of Recursive Binary Search. Given an input list, some left and right bounds, and a value to look for, `rec_binary_search(lst, lbound, rbound, key)` will look for `key` within indicies `>= lbound`, and `< rbound` (Inclusive of left, Exclusive on right). The code should raise a ValueError if the key is not found.

Assume that '__' denotes an indent.

Drag the code into the correct order.

def rec_binary_search(lst, lbound, rbound, key):

if rbound == lbound:

__raise ValueError(f"{key} not found!")

mid = (rbound + lbound) // 2

if lst[mid] < key: # Check 1

__return rec_binary_search(lst, mid+1, rbound, key)

if lst[mid] > key: # Check 2

__return rec_binary_search(lst, lbound, mid, key)

return mid

Question 2 *Submitted Oct 11th 2022 at 8:04:55 am*

Consider the following implementation of the famous Fibonacci Sequence:

```
def fib(n):
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)

if __name__ == "__main__":
    print(fib(1))
    print(fib(2))
```

What will the code do when run?

- ○ Print "1" then "1"
- ○ Print "1" then "2"
- ○ Print "1" then "0"
- ○ Print "1" continuously forever
- [✓ Print "1" and Crash]
- ○ Crash
- ○ Print "1" and then idle

Question 3 *Submitted Oct 11th 2022 at 8:09:46 am*

Consider another recursive function:

```
def mystery(a, b):
    print(a, b)
    if a == 0 or b == 0:
        return a + b
    return mystery(a-1, b) + mystery(a, b-1)

if __name__ == "__main__":
    print(mystery(2, 2))
```

What will it print?

```
2 2
1 2
0 2
1 1
0 1
1 0
```

```
2 1

1 1

0 1

1 0

2 0

8
```

LinkedList recursion

Consider the standard **LinkList** class discussed in the lectures that implements a Linked List using the Node class. You can assume for this exercise our linked list only contains integers. We are now adding to this class the following method:

```
class MysteryList(LinkedList):

    def mystery(self) -> int:
        return self.mystery_aux(self.head)

    def mystery_aux(self, current: Node[int]) -> int:
        if current is None:
            return 0
        else:
            current.item += self.mystery_aux(current.link)
            return current.item
```

In `Ex2.txt`, answer the following questions:

- What does the mystery method do? Explain in terms of its effect on the value of a list with the elements 1,2,3,4,5 (Don't execute the code in Ed, try running through on paper).

- What is the best and worst complexity in Big O notation of our **mystery** method in terms of the length of the list it's operating on?

In `Ex2.py`, add a recursive method `sum_of_evens`, which should return the sum of all even index digits in the list. For example `sum_of_evens` on [1, 3, 4, 6, 8] should return `1+4+8=13`.

Recursive Power

Consider the following recursive function, which computes x to the power of y:

```
def power(x, y):
    if y == 0:
        return 1
    if y == 1:
        return x

    if y % 2 == 0:
        return power(x, y//2) * power(x, y//2)
    else:
        return power(x, y//2) * power(x, y//2) * x

print(power(3, 5))
```

Q1: What is the complexity of this function?

Q2: Is this function tail recursive?

Q3: Try modifying this function so that the runtime complexity is **O(log(y))**.

Quicksort Pivots

For each of the following pivot choices, determine the overall Best and Worst case time complexities, in Big O notation (Assuming that finding these pivots takes **O(1)**).

Question 1 *Submitted Oct 11th 2022 at 8:11:47 am*

Pivot selected is the minimum of the list.

- Best Case O(N), Worst Case O(NlogN)
- Best Case O(NlogN), Worst Case O(NlogN)
- Best Case O(NlogN), Worst Case O(N^2)
- [✓ Best Case O(N^2), Worst Case O(N^2)]

Question 2 *Submitted Oct 11th 2022 at 8:12:08 am*

Pivot selected is `lst[0]`.

- Best Case O(N), Worst Case O(NlogN)
- Best Case O(NlogN), Worst Case O(NlogN)
- [✓ Best Case O(NlogN), Worst Case O(N^2)]
- Best Case O(N^2), Worst Case O(N^2)

Question 3 *Submitted Oct 11th 2022 at 8:12:15 am*

Pivot selected is the median of the list.

- Best Case O(N), Worst Case O(NlogN)

[✓ Best Case O(NlogN), Worst Case O(NlogN)]

Best Case O(NlogN), Worst Case O(N^2)

Best Case O(N^2), Worst Case O(N^2)

Question 4 *Submitted Oct 11th 2022 at 8:12:29 am*

Pivot is selected at random.

Best Case O(N), Worst Case O(NlogN)

Best Case O(NlogN), Worst Case O(NlogN)

[✓ Best Case O(NlogN), Worst Case O(N^2)]

Best Case O(N^2), Worst Case O(N^2)

Question 5 *Submitted Oct 11th 2022 at 8:12:48 am*

Pivot is selected at random, but guaranteed to be larger than `>= 1%` of the list, and smaller than `>= 1%` of the list.

Best Case O(N), Worst Case O(NlogN)

[✓ Best Case O(NlogN), Worst Case O(NlogN)]

Best Case O(NlogN), Worst Case O(N^2)

Best Case O(N^2), Worst Case O(N^2)

Stability and Incrementality

Question 1 *Submitted Oct 11th 2022 at 8:21:51 am*

Is quicksort stable, why?

QuicksSort is not stable as it does not maintains the relative order of records in the case of equality of keys.

QuickSort swap elements accordingly to pivot's position without considering their original positions.

Question 2 *Submitted Oct 11th 2022 at 8:28:18 am*

Is quicksort incremental? Explain why / why not.

QuickSort is not incremental as the way quicksort partitions elements means that any existing sorted relation between elements is quickly destroyed. So if any new elements come into the list, placing them in the correct position with quicksort can be expensive.

Question 3 *Submitted Oct 11th 2022 at 8:27:04 am*

Is mergesort stable? What part of the definition can be changed to make it stable/unstable?

```
def merge_sort_aux(array: ArrayR, start: int, end: int, tmp: T) -> None:
    if not start == end: # 2 or more still to sort
        mid = (start + end)//2
        # split into two halves
        merge_sort_aux(array, start, mid, tmp)
        merge_sort_aux(array, mid+1, end, tmp)
        # merge
        merge_arrays(array, start, mid, end, tmp)
        # copy tmp back into the original
        for i in range(start, end+1):
        array[i] = tmp[i]

def merge_arrays(a: ArrayR, start: int, mid: int, end: int, tmp: T) -> None:
    ia = start
    ib = mid+1
    for k in range(start, end+1):
        if ia > mid: # a finished, copy b
            tmp[k] = a[ib]
            ib += 1
        elif ib > end: # b finished, copy a
```

```
            tmp[k] = a[ia]
            ia += 1
        elif a[ia] <= a[ib]: # a[ia] is the item to copy
            tmp[k] = a[ia]
            ia += 1
        else:
            tmp[k] = a[ib] # b[ib] is the item to copy
            ib += 1
```

MergeSort is stable.

Question 4 *Submitted Oct 11th 2022 at 8:28:45 am*

Is mergesort incremental? If so, explain why. If not, try to provide a slightly different approach that merge sort could take if it knew what subset of the list was already sorted.

Mergesort is not incremental, since it always splits down the middle, and has no early exit conditions.

We could however same time sorting, if we knew only the last `k` elements are unsorted, then we could do the following:

- Break the list into two parts, like merge sort does, but split into sorted vs. unsorted.
- mergesort the unsorted part
- call merge_arrays on the two lists

The complexity of this approach would be O(N + KlogK), better than O(NlogN).

K-Merge Sort

In the usual implementation of merge sort, we break our list in two, and merge the result.

In `merge.py` implement two new functions:

- `merge_k(list_of_lists, k) -> list` which merges a list of k sorted lists together.
- `k_mergesort(lst, k) -> list` which performs a merge sort, but splits the list into k parts.

What is the complexity of these two methods, in terms of `n` and `k`?

Identifying recursive breakdown of sorts.

Question

In general with recursive functions, we find a way to break the problem into smaller chunks, which can be solved separately, and then the results of this subproblems are combined.

Identify how merge and quick sort fit into this category of breaking the problem into smaller chunks and combining, and for both sorts identify which of these two steps requires more computation.

No response