# Algorithmic coding questions

compiled by D.Gueorguiev 5/9/2024

## Array Manipulation

### kth largest element of an array

#### Naive solution

def get\_largest\_element\_by\_index(array: list[int], index: int):

“““

Get the (index+1)-th largest element

:param array: list[int]

:param index: int

“““

array.sort(reverse=True)

return array[index]

**Excerpt 1**: k-th largest element implemented via sort where k=index+1

Question to candidate: what is the time complexity of this code?

Answer: time complexity O(N\*log(N)) where N = len(array)

Question to candidate: why the complexity is O(N\*log(N))?

Answer: this is the complexity of any sorting algorithm on average.

Question to candidate: what kind of default sorting algorithm Python uses internally?

Answer: Time sort which is a hybrid algorithm incorporating merge sort and insertion sort.

#### Solution using Binary Search

def get\_largest\_element\_by\_index(array: list[int], index: int):

“““

Get the (index+1) largest element from the array

:param array: list[int]

:param index: int

“““

low = min(array)

high = max(array)

while low <= high:

mid = low + (high – low) // 2

if count\_larger\_than(array, mid) > index:

low = mid + 1

else:

high = mid - 1

return high + 1

def count\_larger\_than(array, val):

“““

Get the count of all elements larger to val

:param array: list[int]

:param val: int

“““

count = 0

for i in range(len(array)):

if array[i] > val:

count+= 1

return count

**Excerpt 2**: k-th largest element implemented with Binary Search, where k=index+1

Question to candidate: what complexity has the code above? Explain why?

Answer: O(N \* log (high-low)). *Hint*: if we replace the function count\_larger\_than with a function which returns in O(1) time then the time complexity of get\_largest\_element\_by\_index will become O(log(high-low)).

Question to candidate: what is the problem with the following piece of code from Excerpt 2 and how would you rewrite it to be more efficient?

low = min(array)

high = max(array)

*Hint*: what is its time complexity?

Answer: better performing solution would be to use a single loop to find low and high.

Question to candidate: what would the function get\_largest\_element\_by\_index on Excerpt 2 above produce for an input with duplicate values? For instance, without running any code answer what output the invocation of get\_largest\_element\_by\_index([1, 5, 3, 2, 7, 9, 1, 3, 5], 2) would produce? What about get\_largest\_element\_by\_index([1, 5, 3, 2, 7, 9], 2)?

Question to candidate: Given the code shown on Excerpt 2 can you provide solution for the problem of finding the k-th ***smallest*** element with minimal code change? *Hint*: use the fact that the k-th smallest element in an array of N elements is also the N-k+1 largest element in the same array.

Answer: Expressing the idea that the smallest k-th element is also the N-k+1 largest element we replace the parameter **index** with **len(array) – 1 – index** in the RHS of the line

if count\_larger\_than(array, mid) > **index**:

in Excerpt 2.

Thus, the implementation for getting the k-th smallest element becomes:

def get\_smallest\_element\_by\_index(array: list[int], index: int):

“““

Get the (index+1) smallest element from the array

:param array: list[int]

:param index: int

“““

low = min(array)

high = max(array)

while low <= high:

mid = low + (high – low) // 2

if count\_larger\_than(array, mid) > **len(array) – 1 - index**:

low = mid + 1

else:

high = mid - 1

return high + 1

def count\_larger\_than(array, val):

“““

Get the count of all elements larger than val

:param array: list[int]

:param val: int

“““

count = 0

for i in range(len(array)):

if array[i] > val:

count+= 1

return count

**Excerpt 3**: the k-th smallest element implemented with Binary Search where k=index+1

Note: We can obtain an alternative implementation of the k-th smallest element by observing that the concept of getting the smallest k-th element is *complementary* to the concept of getting the largest k-th element. Thus, we obtain the alternative implementation by swapping everywhere the operator **>** with its complement **<=** in Excerpt 2 as shown in Excerpt 4 below.

def get\_smallest\_element\_by\_index(array: list[int], index: int):

“““

Get the (index+1) smallest element from the array

:param array: list[int]

:param index: int

“““

low = min(array)

high = max(array)

while low <= high:

mid = low + (high – low) // 2

if count\_smaller\_than\_or\_equal(array, mid) **<=** index:

low = mid + 1

else:

high = mid - 1

return high + 1

def count\_smaller\_than\_or\_equal(array, val):

“““

Get the count of all elements smaller or equal to val

:param array: list[int]

:param val: int

“““

count = 0

for i in range(len(array)):

if array[i] **<=** val:

count+= 1

return count

**Excerpt 4**: An alternative implementation for k-th smallest element using Binary Search where k=index+1

#### Solution using Priority Queue / Min Heap

import heapq

def get\_largest\_element\_by\_index(array: list[int], index: int):

“““

Get the largest element specified by index from the array

:param array: list[int]

:param index: int

“““

min\_heap = []

for num in array:

heapq.heappush(min\_heap, num)

if (len(min\_heap) > index+1):

heapq.heappop(min\_heap)

return heapq.heappop(min\_heap)

**Excerpt 5**: an implementation for the k-th largest element using priority heap

Question to candidate: Having the code above modify it to find k-th smallest element where k=index+1.

Answer:

The line heapq.heappush(min\_heap, num) will become heapq.heappush(min\_heap, **-**num) .

The line return heapq.heappop(min\_heap) will become return **-**heapq.heappop(min\_heap).

#### Solution using Quicksort-like algorithm with recursive search and Partition function

def get\_smallest\_element\_by\_index(array: list[int], left: int, right: int, index: int):

“““

Get the (index+1) smallest element from the array

:param array: list[int]

:param left: int

:param right: int

:param index: int

“““

if index >= 0 and index <= right – left:

pos = partition(array, left, right)

if pos – left == index:

return array[pos]

if pos – left > index:

return get\_smallest\_element\_by\_index(array, left, pos – 1, index)

return get\_smallest\_element\_by\_index(array, pos + 1, right, index – pos + left)

# return sys.maxsize when index is for a position out of bounds of the array

return sys.maxsize

def partition(array: list[int], left: int, right: int):

“““

Partition array by a pivot value. The pivot value is stored in the right element.

:param array: list[int]

:param left: int

:param right: int

“““

pivot = array[right]

i = left

for j in range(left, right):

if array[j] <= pivot:

array[i], array[j] = array[j], array[i]

i += 1

array[i], array[right] = array[right], array[i]

return i

**Excerpt 6**: an implementation for the k-th largest element using partition and recursive search

Question to candidate: what is the complexity if the array is already sorted?

Answer: in ascending order O(N^2), in descending order O(N).

Question to candidate: what is the complexity of this algorithm on average? How did you arrive of your estimate for the average?

Answer: Surprisingly, the time complexity on average is O(N) where N is the size of the array. For comparison compare with the average time complexity of QuickSort which is O(N\*log(N)). So where does this speedup come from? The thing is that the partition function operates on increasingly shrinking space unlike in QuickSort where it is of O(N).

Question to candidate: is there a way to improve this algorithm speeding up its execution time on average? Hint: look closely into the partition function and the way we choose the pivot.

Answer: Randomized pivot choice would speed up the execution on average.

## Sorting Algorithms

### Heapsort

we use the following map to encode the parent-child relationship essential in the heap data structure:

A diagram of a number and a diagram of a number

Description automatically generated

def heapify(arr, N, i):

largest = i # Initialize `largest` as root

l = 2 \* i + 1 # left = 2\*i + 1

r = 2 \* i + 2 # right = 2\*i + 2

# See if left child of root exists and is

# greater than root

if l < N and arr[largest] < arr[l]:

largest = l

# See if right child of root exists and is

# greater than root

if r < N and arr[largest] < arr[r]:

largest = r

# Change root, if needed

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i] # swap

# Heapify the tree rooted at `largest` which is a child of the node `i`.

heapify(arr, N, largest)

# The main function to sort an array of given size

def heapSort(arr):

N = len(arr)

# Build a maxheap.

for i in range(N//2 - 1, -1, -1):

heapify(arr, N, i)

# One by one extract elements

for i in range(N-1, 0, -1):

arr[i], arr[0] = arr[0], arr[i] # swap

heapify(arr, i, 0)

## Algorithms using Advanced Data Structures, Recursion and Memoization

### Calculator

Implement calculator which parses strings as [:expression:][:operator:][:expression:] where [:expression:] is defined recursively as:

[:expression:] = [:number:][:operator:][:expression:]

[:expression:][:operator:][:number:]

[:left\_parenthesis:][:expression:][:right\_parenthesis:]

[:number:][:operator:][:number:]

Example:

1+2-(3+4)\*5

#### Solution of the Calculator problem using stack

operands = [‘+’, ‘-‘]

stack = []

def check\_for\_matching\_parentheses(expr: str):

“””

:param expr: str

“””

count\_left\_par = 0

count\_right\_par = 0

for ch in expr:

if ch == ‘(‘:

count\_left\_par += 1

elif ch == ‘)’:

count\_right\_par += 1

return count\_left\_par == count\_right\_par

def enclosing\_parentheses\_present(expr: str):

“””

Returns true if the expression is enclosed in parentheses

Raises ValueError in case opening or closing parenthesis is missing.

:param expr: str

“””

if exp[0] == ‘(‘ and expr[-1]==’)’:

return True

elif expr[0] == ‘(‘ and expr[-1] != ’)’ or

expr[0] != ‘(‘ and expr[-1] == ’)’:

if check\_for\_matching\_parentheses(expr):

return False

else:

raise ValueError(“missing parenthesis!”)

else:

return False

def get\_top\_level\_operator\_and\_operands (expr: str): (str, str, str)

“””

:param expr: str

:return operator, operand1, operand2

“””

idx\_op\_left = -1

left\_parenthesis\_count = 0

right\_parenthesis\_count = 0

for (idx,ch) in expr:

if ch == ‘(‘:

left\_parenthesis\_count += 1

elif ch == ‘)’:

right\_parenthesis\_count += 1

elif ch in operands:

if left\_parenthesis\_count == right\_parenthesis\_count:

idx\_op\_left = idx

break

if idx\_op\_left == -1

raise ValueError(“operand not found!”)

if idx\_op\_left == len(expr)-1:

raise ValueError(“operand at the end of an expression!”)

return expr[idx\_op\_left], expr[0,idx\_op\_left], expr[idx\_op\_left+1:-1]

def parse\_expr(expr: str):

“””

computes the result of expr using global stack

expr: str

“””

global stack

if enclosing\_parentheses\_present(expr):

parse\_expr(expr[1:-1])

elif str.isnumeric():

stack.append(str)

else:

(op, expr1, expr2) = get\_top\_level\_operator\_and\_operands(expr)

stack.append(op)

parse\_expr(expr1)

parse\_expr(expr2)

def unwind\_stack(): int

“””

stack unwinding and calculation of the final result

“””

global stack

def compute\_result(expr: str): int

“””

compute the result of expr using stack

“””

parse\_expr(expr)

#### Solution of the Calculator problem using stack with memoization

Recall the solution [:expression:] is defined recursively as:

[:expression:] = [:number:][:operator:][:expression:]

[:expression:][:operator:][:number:]

[:left\_parenthesis:][:expression:][:right\_parenthesis:]

[:number:][:operator:][:number:]

//TODO: finish the section providing solution of the Calculator problem with memoization

## Dynamic Programming

### Rod Cutting Problem

Given a rod of length inches (or centimeters) and a table of prices for find the max obtainable revenue by cutting the rod and selling the pieces.

The problem must exhibit optimal substructure if in order to solve the original problem of size we solve similar problems of the same type but smaller sizes.

(1)

Execution time complexity:

(2)

The solution of (2) is given by

(3)

(3) represents exponential time complexity.

Let us assume we are given an integer n and a list p with len(p) >= n. We want to find the max revenue and a partition of n which achieves it.

in code:

def cut\_rod(n: int, p: list[int]):

if n == 0:

### Longest Common Sequence

Let us consider strings in the form where are strings or characters. We say that the tuple forms a sequence in if appears on the left of for each pair such that . Given two strings and find the length of the largest common sequence .

#### Naïve Implementation

def longest\_common\_sequence(s1: str, s2: str, m: int, n: int):

“““

Determine the length of the longest common sequence.

:param s1: str

:param s2: str

:param m: int

:param n: int

“““

if m == 0 or n == 0:

return 0

elif s1[m-1] == s2[n-1]:

return 1 + longest\_common\_sequence(s1, s2, m-1, n-1)

else:

return max(longest\_common\_sequence(s1, s2, m, n-1),

longest\_common\_sequence(s1, s2, m-1, n))

**Excerpt 7**: naïve implementation for the longest common sequence

Question to the candidate: what is the problem with the Naïve implementation?

Answer: awful time complexity which is exponential

Question to candidate: Why the time complexity is exponential, and can it be eliminated?

Answer: It is exponential because the same string fragments are searched multiple times. In this case using recursion by itself alone does not do us favor. Yes, it can be avoided by using memoization.

Details:

Let us have the following two strings “DIMIT” and “DMTI”. Let us find out how the function in Excerpt 7 will execute. For brevity we will denote the function longest\_common\_sequence on Excerpt 7 with .

#### Dynamic Programming implementation

Optimal Subproblem formulation

Initially,

### 0/1 Knapsack problem

Given are items where each item has some weight and profit associated with it. Also, it is given a bag with capacity - that is, the bag can hold at most weight in it. The task is to put such combination of items in the bag so that the profit is maximized. The constraint is that we can put an item in the bag, or we cannot put it at all, it is not possible to put only a fraction of it.

Input: N = 3, W = 4, profit = [1, 3, 4], weight = [4, 5, 1]

Let us denote with

Naïve recursive algorithm:

# A naive recursive implementation

# of 0-1 Knapsack Problem

def knapSack(W, wt, val, n):

# Base Case

if n == 0 or W == 0:

return 0

# If weight of the nth item is

# more than Knapsack of capacity W,

# then this item cannot be included

# in the optimal solution

if (wt[n-1] > W):

return knapSack(W, wt, val, n-1)

# return the maximum of two cases:

# (1) nth item included

# (2) not included

else:

return max(

val[n-1] + knapSack(

W-wt[n-1], wt, val, n-1),

knapSack(W, wt, val, n-1))

1 ) find the subset of elements , each with a weight less than . That is,

2 )

## Recursion and Backtracking

### N Queens problem

Place queens on board so that no pair of Queens attack each other.

For example, when N = 4 we have:

.Q..

...Q

Q...

..Q.

### 2 x 2 Sudoku Solver

//TODO: Finish the section on the 2 x 2 sudoku solver

## Graph Algorithms

Topological Sorting

Depth-First Traversal of Graph

Breadth-First Traversal of Graph

Detect Cycle in Directed Graph

### Reference

[[1] Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, 3rd edition, MIT, 2009](https://pd.daffodilvarsity.edu.bd/course/material/book-430/pdf_content)

[2] [What Every Computer Scientist Should Know About Floating-Point Arithmetic, David Goldberg, 1991](https://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/goldberg-floating-point.pdf)

[[3] Examples of Floating Point Problems, Julia Evans, online blog, 2023](https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/)

[[4] The K-th Largest Element Problem, Daniel Speiser, Matthijs Hollemans, The Swift Algorithm Club, 2019](https://github.com/kodecocodes/swift-algorithm-club/tree/master/Kth%20Largest%20Element)