DS project 1 document Behraz Fereshteh Saniee

File references:

1. notrecursiveOn.py
2. notrecursiveOn2.py
3. recursive.py
4. recursiveNoPrettify.py
5. randomgeneratorandchart.py
6. search.py

1: a not recursive approach for problem 1 in project with O(n) complexity using stack

2: a not recursive approach for problem 1 in project with O(n^2) complexity no stack is used

3: a recursive approach for problem 1 in project with O(n^2) complexity and O(n) pre-process

4: a recursive approach for problem 1 in project with O(n^2) complexity no pre-process

5: random array generation and chart presentation

6: three search algorithms implementation

1:

class Linkedlist:  
 def \_\_init\_\_(self):  
 self.data = ""  
 self.next = None  
  
class Stack:  
 def \_\_init\_\_(self):  
 self.list\_head = Linkedlist()  
  
 def push(self, x):  
 ll = Linkedlist()  
 ll.data = x  
 ll.next = self.list\_head  
 self.list\_head = ll  
  
 def pop(self):  
 if not self.is\_empty():  
 to\_pop = self.top()  
 self.list\_head = self.list\_head.next  
 return to\_pop  
  
 def top(self):  
 if not self.is\_empty():  
 return self.list\_head.data  
  
 def is\_empty(self):  
 if self.list\_head is None:  
 return True  
 else:  
 return False  
  
my\_str = input()  
stack = Stack()  
for i in range(len(my\_str)):  
 if my\_str[i] == ')':  
 temp = ""  
 while stack.top() != '(':  
 temp = stack.pop() + temp  
 stack.pop()  
 temp2 = ""  
 while not stack.is\_empty() and stack.top() in "0123456789":  
 temp2 = stack.pop() + temp2  
 temp \*= int(temp2)  
 stack.push(temp)  
 else:  
 stack.push(my\_str[i])  
output = ""  
while not stack.is\_empty():  
 output = stack.pop() + output  
print(output)

Stack

push() : O(1)

pop() : O(1)

top() : O(1)

is\_empty() : O(1)

Linked list node

explanation:

from start of the string push in stack until you reach a “)” then pop until you see a “(“ at top of the stack

there is no inner “(…)” in this scope this scope is stored in *temp* pop the “(“ at top of the stack

while stack is not empty and there is a digit at top of the stack pop the digit final number is stored in

*temp2* multiply(repeat) *temp* for integer value of *temp2* final result is stored in *temp* finally push *temp*

in the stack when all of the string is processed(pushed and if needed popped and concatenated) our final

output is stored in the stack but it’s reversed so we reverse and store it in *output*

complexity:

the most repeated statements are push and pop (they are in most inner loops)

anything that is pushed is eventually popped consequently the number of pops and pushes is the same

every element of string is pushed once and for every “(…)” scope there is an extra push for its content

the number of “(…)” scopes is the same as number of “(“ in the string so it’s less then the length of string

that leaves us with O(n) complexity (n is the length of string) ( O(2n + 2(less than n) + constant) = O(n) )

2:

my\_str = input()  
index = len(my\_str) - 1  
while index >= 0:  
 if my\_str[index] == '(':  
 index2 = index  
 while my\_str[index2] != ')':  
 index2 += 1  
 index3 = index - 1  
 while my\_str[index3 - 1] in "1234567890" and index3 > 0:  
 index3 -= 1  
 my\_str = my\_str[:index3] + int(my\_str[index3: index]) \* my\_str[index + 1:index2] +\ my\_str[index2 + 1:]  
 index -= 1  
print(my\_str)

explanation:

going back from the end of the string until we reach a “(“ the go ahead until we reach a “)”

find the number behind the “(“ and the new string is first part of string before the first digit of the number

and the middle part multiplied and the second part after “)” continue until there is no “(“ left (reach the

beginning of the string)

complexity:

iteration for “(“ to “)” is at most O(n)

and the is a loop from length of string to 0 so the final complexity is O(n^2)

3:

def prettify(the\_str):  
 i = 0  
 while i < len(the\_str):  
 if the\_str[i] not in "1234567890()":  
 if the\_str[i - 1] != '(' or the\_str[i + 1] != ')':  
 the\_str = the\_str[:i] + "1(" + the\_str[i] + ")" + the\_str[i + 1:]  
 i += 2  
 i += 1  
 return the\_str  
  
  
def solve(the\_str):  
 if len(the\_str) <= 1:  
 return the\_str  
 for i in range(len(the\_str)):  
 m = 0  
 if the\_str[i] == '(':  
 m += 1  
 j = i  
 while m != 0:  
 j += 1  
 if the\_str[j] == '(':  
 m += 1  
 elif the\_str[j] == ')':  
 m -= 1  
 return int(the\_str[:i]) \* solve(the\_str[i + 1:j]) + solve(the\_str[j + 1:])  
  
  
my\_str = input()  
print(solve(prettify(my\_str)))

expiation:

the prettify function inserts “1(“ + char+ “)” for every character that is not surrounded by “(“ and “)” this will

ensure us that before the first “(“ is always only a number

is solve function if the length of string is less the 2 (empty or 1 char) it will return the string this is the termination condition

for recursive function then we go from beginning of string until a “(“ is reached the we go ahead until

the corresponding “)” is seen the new string is the multiplication of solved “(…)” scope by the number before “(“ and the solve of the rest of string after “)”

complexity:

prettify is O(n)

in each call of solve there is at most O(n) iterations

there is two calls for a partition of the original string so there can be at most O(n) calls

the final complexity is O(n^2) + O(n) = O(n^2)

T(n) = T(n – a) + T(n-b) + O(n)

4:

def solve(the\_str):  
 for i in range(len(the\_str)):  
 if the\_str[i] == '(':  
 m = 1  
 j = i  
 while m != 0:  
 j += 1  
 if the\_str[j] == '(':  
 m += 1  
 elif the\_str[j] == ')':  
 m -= 1  
 k = i - 1  
 while the\_str[k] in "0123456789":  
 k -= 1  
 return ("" if solve(the\_str[:k + 1]) is None else solve(the\_str[:k + 1])) + int(  
 the\_str[k + 1:i]) \* solve(the\_str[i + 1:j]) + solve(the\_str[j + 1:])  
 return the\_str  
  
  
my\_str = input()  
print(solve(my\_str))

explanation:

the same as previous part but the prettify is removed and the return condition is that there is no “(“ in the string

and we find the first digit of the number before “(“ then the first and multiplied middle and second parts of the string are

concatenated creating the new string

complexity:

the same way as previous part the complexity is O(n^2)

T(n) = T(n - a) + T(n - b) + T(n - c) + O(n)

5 and 6:

# linear search:  
def linear\_search(array, item):  
 global counter  
 for i in range(len(array)):  
 counter += 1  
 if array[i] == item:  
 return i  
 return -1

Complexity:

The number of iterations is at most the length of string so the complexity is O(n)

T(n) = O(n)

# binary search:  
def binary\_search(array, item, begin, end):  
 global counter  
 counter += 1  
 if begin == end:  
 if array[begin] == item:  
 return begin  
 else:  
 return -1  
 mid = (begin + end) // 2  
 if item <= array[mid]:  
 return binary\_search(array, item, begin, mid)  
 else:  
 return binary\_search(array, item, mid + 1, end)

Complexity:

T(n) = T(n/2) + O(1)

T(n) = T(n/4) + 2O(1)

…

T(n) = T(n/(2^i)) + iO(1)

i can be at most lgn because T(1) will be calculated in O(1)

therefore: T(n) = O(lgn)

# ternary search:  
def ternary\_search(array, item, begin, end):  
 global counter  
 counter += 1  
 if end - begin >= 0:  
 mid1 = begin + (end - begin) // 3  
 mid2 = end - (end - begin) // 3  
 if array[mid1] == item:  
 return mid1  
 if array[mid2] == item:  
 return mid2  
 if item < array[mid1]:  
 return ternary\_search(array, item, begin, mid1 - 1)  
 elif item > array[mid2]:  
 return ternary\_search(array, item, mid2 + 1, end)  
 else:  
 return ternary\_search(array, item, mid1 + 1, mid2 - 1)  
 return -1

Complexity:

T(n) = T(n/3) + O(1)

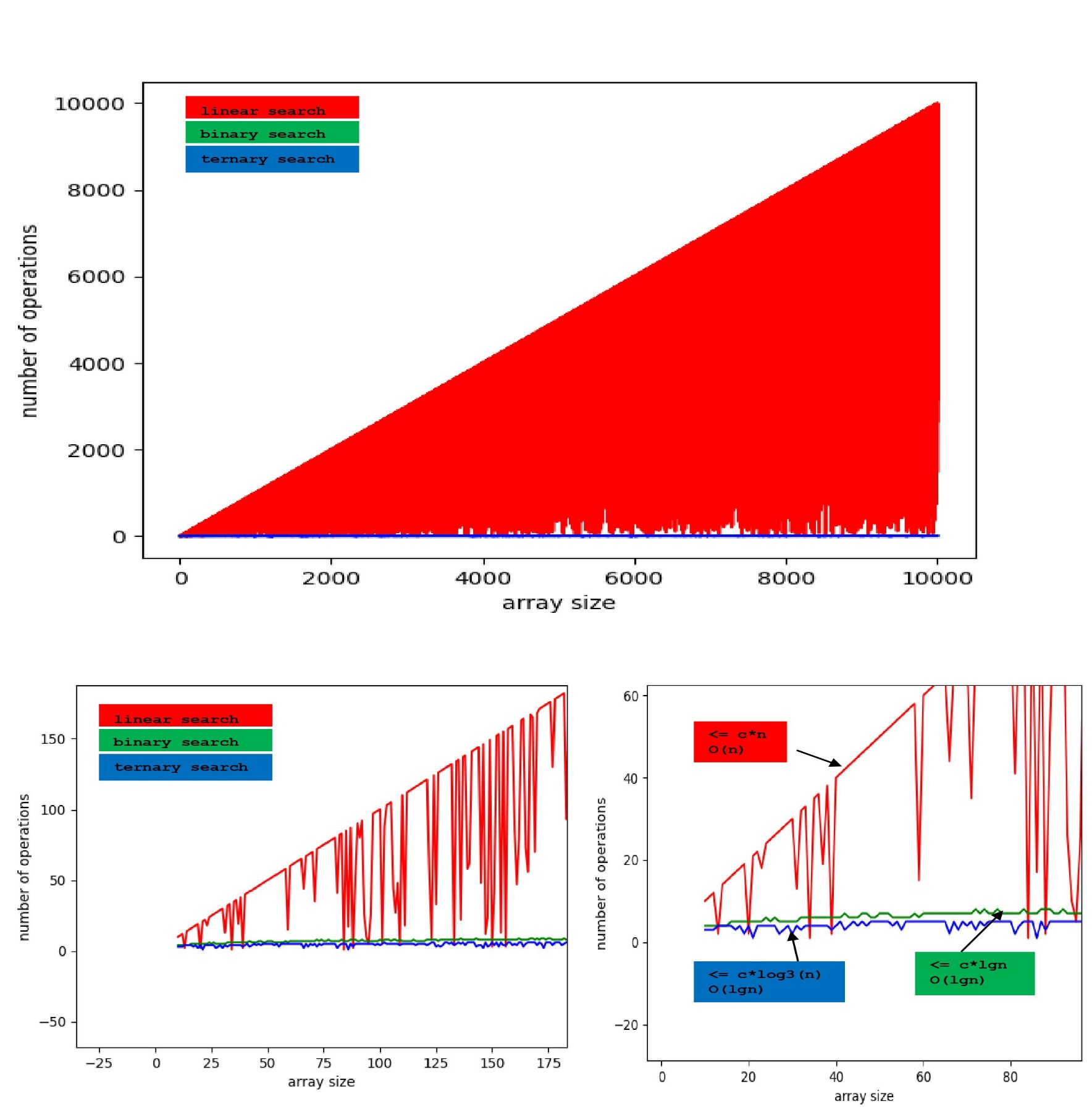
T(n) = T(n/9) + 2O(1)

…

T(n) = T(n/(3^i)) + iO(1)

i can be at most log3(n) because T(1) will be calculated in O(1)

therefore: T(n) = O(log3(n)) = O(lgn)

**Charts**:

If : f(n) = O(g(n)) then f(n) is less or equal to constant\*g(n)

