Mohammed VI Polytechnic University
Institute of Science, Technology & Innovation
Al-Khwarizmi Department
Africa Business School

Week 1 assignment

Course:
HPC

Submitted by:
Amine EL BERRI

31. März 2023

Problem 1

Given an integer n, count the number of its divisors.

Solution 1:

```python
def count_divisors(n):
    count = 0
    d = 1
    while d <= n:
        if n % d == 0:
            count += 1
        d += 1
    return count
```

Solution 2:

```python
def count_divisors(n):
    count = 0
    d = 1
    while d * d <= n:
        if n % d == 0:
            count += 1 if n / d == d else 2
        d += 1
    return count
```

Solution :

- Describing solution 1: The function count_divisors counts the number of divisors of a given integer n by looping over all possible divisors and using a counter variable to count the found divisors. since we iterate "n" times for each value of "n",the number of operations performed is directly proportional to the value of "n". Hence the algorithm has a time complexity O(n).
- Describing solution 2: This time we iterate over all possible divisors up to $\sqrt{n}$ and we fill up a counter variable each time a divisor is found, considering whether we are dealing with multiples of n . The number of operations is proportional to the square root of n. This can be expressed as O($\sqrt{n}$).
- Running the 2 programs for different values of n :

Value of n	Solution 1 timing	Solution 2 timing
100	0.000003	0.000003
1000	0.000003	0.000003
10000	0.002	0.000043
100000	0.031	0.0003
1000000	0.34	0.004

Problem 2

Big-O notation:

a) $T(n) = 3n^3 + 2n^2 + \frac{1}{2}n + 7$ Prove that : $T(n) = \mathbf{O}(n^3)$

b) Prove : $\forall k, n^k$ is not $\mathbf{O}(n^{k-1})$

Solution

a) For $n \geq 1$, we have:

$$T(n) \leq n^3(3 + 2 + \frac{1}{2} + 7)$$

Thus :

$$T(n) \leq \frac{15}{2}n^3 + 7$$

For $c = \frac{16}{2}$ and for all $n \geq 1$ we have:

$$T(n) \leq \frac{15}{2}n^3 + 7 \leq cn^3$$

Hence finally : $T(n) = \mathbf{O}(n^3)$

b) We proceed with contradiction,assume that $\forall k, n^k$ is $\mathbf{O}(n^k - 1)$. Then : $\exists c, n_0$ such that $\forall n \geq n_0$, $n^k \leq c.n^{k-1}$.

Dividing both sides by n^{k-1}, we get $n \leq c$ for all $n \geq n_0$. But this is a contradiction, since n can be arbitrarily large and c is a constant., Therefore, we have shown that $\forall k \geq 1, n^k$ is not $O(n^{k-1})$.

Problem 3

a) Given two sorted arrays, write a function (with a language of your choice) that merge the two arrays into a single sorted array.

b) Analyze the complexity of your func.on using Big-O notation.

Solution:

a) In python we proceed as follows

```python
def merge(A, B):
    i, j = 0, 0
    C = []

    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            C.append(A[i])
            i += 1
        else:
            C.append(B[j])
            j += 1

    while i < len(A):
        C.append(A[i])
        i += 1

    while j < len(B):
        C.append(B[j])
        j += 1

    return C
```

```python
A = np.array([2, 4, 6, 8, 10])
B = np.array([1, 3, 5, 7, 9])
print(A)
print(B)
```

```
[ 2  4  6  8 10]
[1 3 5 7 9]
```

```python
merge(A, B)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

b) Two input sorted lists $A = [a_1, ..., a_\alpha]$ of size α and $B = [b_1, ..., b_\beta]$ of size β can be merged into a sorted list $C = [c_1, ..., c_{\alpha]+\beta}$ of size $\alpha + \beta$ in linear time.

Proof

- Let pointers i, j, and k to current positions in A, B, and C,respectively, be initially at the first positions, $i = j = k = 1$.
- Each time the smaller of a_i and b_j is copied to c_k, and the pointers k and either i or j are incremented by 1:

$$\begin{cases} a_i \geq b_j \Rightarrow c_k = b_j \, , j \leftarrow j+1 \, , k \leftarrow k+1 \\ a_i \leq b_j \Rightarrow c_k = a_i \, , i \leftarrow i+1 \, , k \leftarrow k+1 \end{cases}$$

- After A or B is exhausted, the rest of the other list is copied to C
- Each comparison advances k so that the maximum number of comparisons is n $=$ $\alpha + \beta$, all other operations being linear, too
  Hence finally the time complexity is $\mathbf{O}(\alpha + \beta)$

Problem 4

- Using the master method analyse the complexity of merge sort.
- Using the master method analyse the complexity of binary search

Solution:

- Let's assume that $T(n)$ is the worst-case time complexity of merge sort for n integers. When $n \geq 1$ (merge sort on single element takes constant time), We can break down the time complexities as follows:
  - Divide part: the time complexity of the divide part is $\mathbf{O}(1)$, because calculating the middle index takes constant time.
  - Conquer part: We are recursively solving two sub-problems. each of size $\frac{n}{2}$. So the time complexity of each sub-problem is $T(n/2)$ and overall time complexity of conquer part is $2T(n/2)$.
  - Combine part: The worst case time complexity of the merging process is $\mathbf{O}(n)$:

  To calculate $T(n)$ we neod to sum the time complexities of the divide ,conquer and combine part , thus getting the recursion : $T(n) = \mathbf{O}(1) + 2T(n/2) + \mathbf{O}(n)$ Hence :
  $T(n) = 2T(n/2) + c.n$
  Now recall the 3 cases of complexity analysis using the master theorem :

- If $f(n) = \mathbf{O}(n^k)$ where $k \leq \log_b(a)$ then $T(n) = \mathbf{O}(n^{\log_b(a)})$
- If $f(n) = \mathbf{O}(n^k)$ where $k = \log_b(a)$ then $T(n) = \mathbf{O}(n^k. \log n))$
- If $f(n) = \mathbf{O}(n^k)$ where $k \geq \log_b(a)$ then $T(n) = \mathbf{O}(n^k)$

Comparing this with the merge sort algorithm lends us :
$T(n) = 2T(n/2) + c.n$
$T(n) = aT(n/b) + \mathbf{O}(n^k)$
where : a= 2 and b=2
$\mathbf{O}(n^k) = c.n = \mathbf{O}(n) \Rightarrow k = 1$
Also : $\log_b(a) = \log_2(2) = 1 = k$
This falls under the 2nd case of the Master theorem , hence finally the time complexity of merge sort with the master theorem is given by :
$T(n) = \mathbf{O}(n^k. \log n)) = \mathbf{O}(n^1. \log n)) = \mathbf{O}(n. \log n))$

- Since the binary search time complexity satisfies the recurrence : $T(n) = T(n/2) + \mathbf{O}(1)$
For a=1 and b=2 we have :$f(n) = \mathbf{O}(1)$
Applying the master theorem we deduce that the time complexity for binary search is given by : $\mathbf{O}(\log n))$

Problem 5

- Write a function called merge sort (using a language of your choice) that takes two arrays as parameters and sort those two arrays using the merge sort algorithm.
- Analyse the complexity of your algorithm without using the master theorem.
- Prove the 3 cases of the master theorem.
- Choose an algorithm of your choice and analyse it's complexity using the Big-O notation.

Solution:

- ```
def merge_sort (arr1 , arr2):
  if len(arr1) <= 1 and len(arr2) <= 1:
       return sorted (arr1 + arr2)
 mid1 = len (arr1) // 2
 mid2 = len (arr2) // 2
left1 = arr1 [: mid1]
```
```

```
  right1 = arr1 [ mid1 :]
  left2 = arr2 [: mid2 ]
  right2 = arr2 [ mid2 :]
  sorted_left = merge_sort ( left1 , left2 )
  sorted_right = merge_sort ( right1 , right2 )
  return merge ( sorted_left , sorted_right )
def merge ( left , right ) :
  result = []
  while left and right :

  if left [0] <= right [0]:
  result . append ( left . pop (0) )
  else :
  result . append ( right . pop (0) )
  if left :
  result . extend ( left )
  if right :
  result . extend ( right )
  return result

  A = [4 , 2 , 1 , 6 , 8]
  B = [3 , 7 , 5 , 9]
  sorted_array = merge_sort (A , B )
  print ( sorted_array )
  Output : [1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]
```

- let's analyse the time complexity of this algorithm without using the master theorem : we distinguis several levels to the tree our algorithm is based on :
  - At the first level, there is one node with value n.
  - At the second level, there are two nodes with value $\frac{n}{2}$.
  - At the third level, there are four nodes with value $\frac{n}{4}$.
  - At the k-th level, there are 2^k nodes with value $\frac{n}{2^k}$.
  - The last level will have nodes with value 1 , and there will be n of them.

  We can compute the total number of nodes in the tree as follows:

  $$1 + 2 + 4 + \cdots + 2^{k-1} + n = 2^k - 1 + n$$

  where k is the number of levels in the tree. Since the height of the tree is k, we have: $2^k = n + 1 \implies k = \log_2(n+1)$ Therefore, the tree has $\log_2(n+1)$ levels.

To compute the time complexity of merge sort based on this tree, we can start at the bottom of the tree and work our way up. At each level k, the total work done is $O(n)$ (since there are n nodes at that level, each of which takes constant time to sort).
Therefore, the total work done at all levels is: $O\left(n\log_2(n+1)\right)$
We deduce that the algorithm has time complexity : $\mathbf{O}(n.\log(n))$

Problem 6

- Write a function using python3 that multiply two matrices A,B (without the use of numpy or any external library).
- What's the complexity of your algorithm (using big-O notation)?
- Write the same function in C. (bonus)
- Optimize this multiplication and describe each step of your optimisation.

Solution:

-
```
def matrix_multiply (A , B ) :
 # check the dim of each matrix
 if len( A [0]) != len( B ) :
 raise ValueError (" Matrix dimensions do not match ")

 # define the result matrix
 result = [[0 for j in range ( len( B [0]) ) ] for i in range ( len( A
    ) ) ]

 # multiplication
 for i in range (len ( A ) ) :
 for j in range (len( B [0]) ) :
 for k in range (len ( B ) ) :
 result [ i ][ j ] += A [ i ][ k ] * B [ k ][ j ]

 return result
```

- The complexity of this algorithm is $\mathbf{O}(n^3)$. This is due to the fact that each element of the result matrix requires n multiplications and n additions, and there are n^2 elements in total, leading to a total of n^3 operations.
- Let's write the same function in C:

```c
#include <stdio.h>
#include <stdlib.h>

int** matrix_multiplication(int** A, int** B, int m, int n, int nB) {
    int i, j, k;
    int** res = (int**)malloc(m * sizeof(int*));
    for (i = 0; i < m; i++) {
        res[i] = (int*)malloc(nB * sizeof(int));
        for (j = 0; j < nB; j++) {
            res[i][j] = 0;
            for (k = 0; k < n; k++) {
                res[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return res;
}
```

- To optimize the multiplication we propose the following vectorization approach :

```python
def py_matmul3(a, b):
    ra, ca = a.shape
    rb, cb = b.shape
    assert ca == rb, f"{ca} != {rb}"

    output = np.zeros(shape=(ra, cb))
    for i in range(ra):
        output[i] = np.dot(a[i], b)


    return output
```

Using broadcasting, we can essentially remove the loop and using just a line output[i] = np.dot(a[i], b) we can compute entire value for the ith row of the output matrix. What numpy does is broadcasting the vector a[i] so that it matches the shape of matrix b. Then it calculates the dot product for each pair of vector. Broadcasting rules are pretty much same across major libraries like numpy, tensorflow, pytorch etc.

Problem 7

Quiz

Solution

a) A

b) D

c) C