Assessed Coursework 1 for Algorithms and Machine Learning (MATH20017)

This notebook contains the questions for Part 1 of the assessed coursework for the unit Algorithms and Machine Learning (MATH20017), Autumn 2024.

Marks for this coursework will count 10% towards your final grade.

Important Submission Requirements:

- You must submit:
  i. A PDF file with all answers visible.
  ii. The notebook used to create the file (either `.ipynb` or `.rmd`).
  iii. Any other files required (e.g., images).

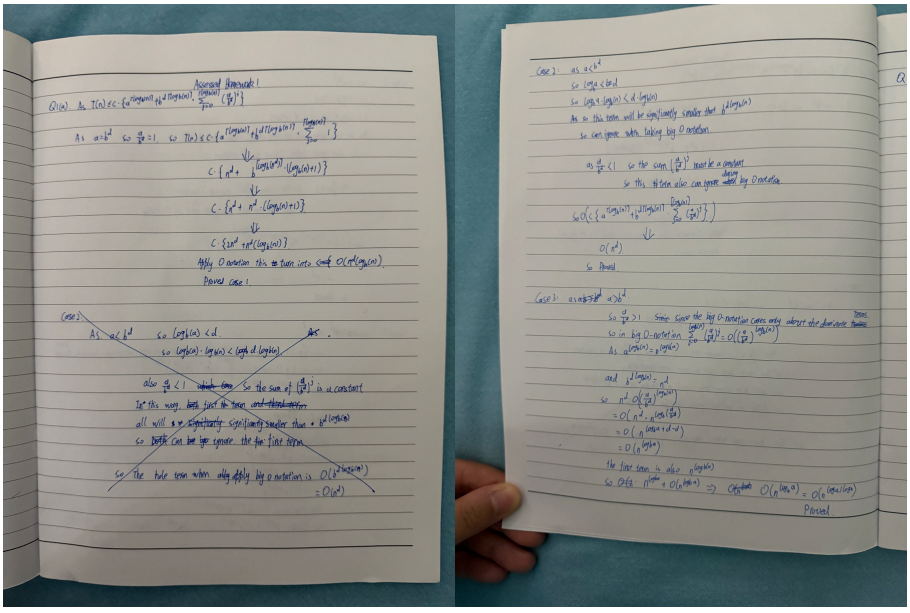
Ensure that all answers are well presented and clearly readable.

Contact: henry.reeve@bristol.ac.uk

Question 1: Divide-and-Conquer Algorithms (16 marks)

(a) (8 marks)

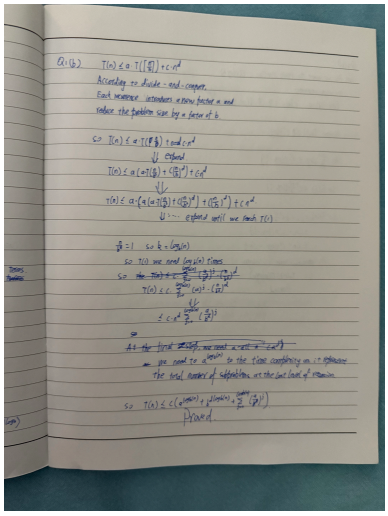
Prove that the key lemma for master method bound implies the master method bound theorem.



(b) (8 marks)

Prove the key lemma for master method bound.

Remark: You may follow the proof strategy from the lectures. The goal is to understand and clearly explain each step of the proof in your own words.



Question 2: Monte Carlo Method (10 marks)

(a) (10 marks)

Edit the code below (updates 1, 2, 3) to create and apply a function that generates a Monte Carlo estimate of π, accurate to 2 decimal places. You can check this by comparing with `np.pi`.

```
#python

import numpy as np # load the numpy library
np.random.seed(2024) # set random seed for reproducibility
n = 1000
U = np.random.rand(n)
```

```
def monte_carlo_pi(num_observations=1000, random_seed=2024):
    # set random seed for reproducibility
    np.random.seed(random_seed)

    # initialise sequence of i.i.d uniform
    U = np.random.rand(num_observations)
    V = np.random.rand(num_observations)

    num_in_circle = 0

    for i in range(num_observations):
        if (((U[i] - 1/2) ** 2 + (V[i] - 1/2) ** 2) <= 1/4):
            num_in_circle += 1

    pi_estimate = 4 * num_in_circle  / num_observations # Update (3)

    return pi_estimate
```

```
# Test your function
pi_estimate_mc = monte_carlo_pi(num_observations=10000000, random_seed=2024)
print("NumPy estimate: ", round(np.pi, 2))
print("Monte Carlo estimate: ", round(pi_estimate_mc, 2))
```

```
NumPy estimate:  3.14
Monte Carlo estimate:  3.14
```

Question 3: Quick Sort Algorithm (32 marks)

(a) (16 marks)

Implement the quick sort algorithm in Python.

```python
# Python

import random

def pivot_partition (X, pivot_index):
    pivot_value = X[pivot_index]
    lower = []
    upper = []
    n = len(X)
    for i in range(n):
        if(X[i] < pivot_value):
            lower.append(X[i])
        elif(X[i] > pivot_value):
            upper.append(X[i])
    return pivot_value, lower, upper



def quick_sort (X):

    n = len(X)

    if (n < 2) :
        return X.copy()

    #// Choose random pivot
    pivot_index = random.randint(0, n - 1)

    pivot_value, lower, upper = pivot_partition(X, pivot_index)
    #// partition around the corresponding pivot value.

    lower = quick_sort(lower)
    upper = quick_sort(upper)
    #// make two recursive function calls to quick_sort


    Z = [] # // initialise empty array

    for i in range( len(lower) ):
        Z.append(lower[i])

    for i in range( len(lower), n - len(upper) ):
        Z.append(pivot_value)

    for i in range( n - len(upper), n):
        Z.append(upper[i - n + len(upper)])
    #// concatenate sorted sub-arrays

    return Z
```

```
import numpy as np # load the numpy library

def sorting_function_test(sorting_function, random_seed=2024,
                          array_size=30, max_int=100, num_tests=50):
  np.random.seed(random_seed) # set the random seed
  output = []
  num_tests_failed = 0

  for i in range(num_tests):
    X=list(set(np.random.randint(max_int, size=(array_size))))
    np.random.shuffle(X)
    if sorted(X) != sorting_function(X):  # Compare with built-in sorted
            num_tests_failed += 1

  if num_tests_failed==0:
    print("Success! All tests passed.")
  else:
    print(num_tests_failed,"/",num_tests," failed.")
```

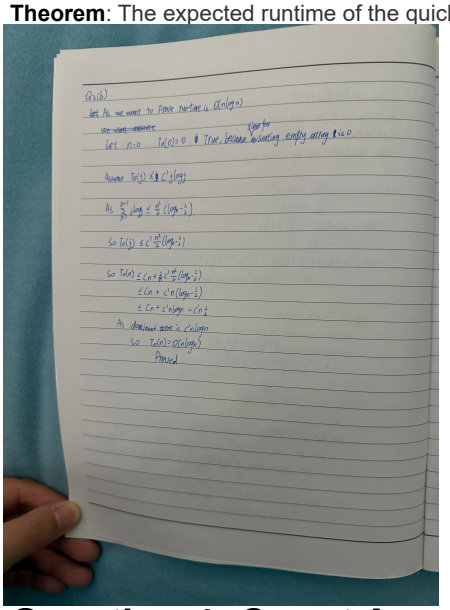
```
sorting_function_test(quick_sort)
```

```
Success! All tests passed.
```

(b) (16 marks)

Prove the theorem on the expected runtime of the quick sort algorithm.

Theorem: The expected runtime of the quick_sort algorithm applied to an array of length n is $O(n \log n)$.



Question 4: Count Array Inversions (32 marks)

(a) (16 marks)

Implement a function called `count_inv_and_sort` which takes as input a numerical array `X` and outputs a tuple with:

1. The number of array inversions in `X`.
2. A sorted array containing the same elements as `X`.

```
import numpy as np


def count_split_inv_and_merge(x, y):
    nx = len(x)
    ny = len(y)
    z = np.zeros(nx + ny)
    split_inv = 0
    i = 0
    j = 0
    #// initialisation

    for k in range(nx + ny):
        if(i < nx and (j == ny or x[i] <= y[j])):
            z[k] = x[i]
            i += 1
        else:
            z[k] = y[j]
            #// Add on the length of X[i : nX]
            split_inv = split_inv + nx - i
            j += 1

    return split_inv, z
```

```
def count_inv_and_sort(x):
    n = len(x)
    if (n < 2):
        return(0, x)
    else:
        (l_inv, y) = count_inv_and_sort(x[0 : n // 2])
        (r_inv, z) = count_inv_and_sort(x[n // 2 : n])
        (split_inv, w) = count_split_inv_and_merge(y, z)
        return (l_inv + r_inv + split_inv, w)
```

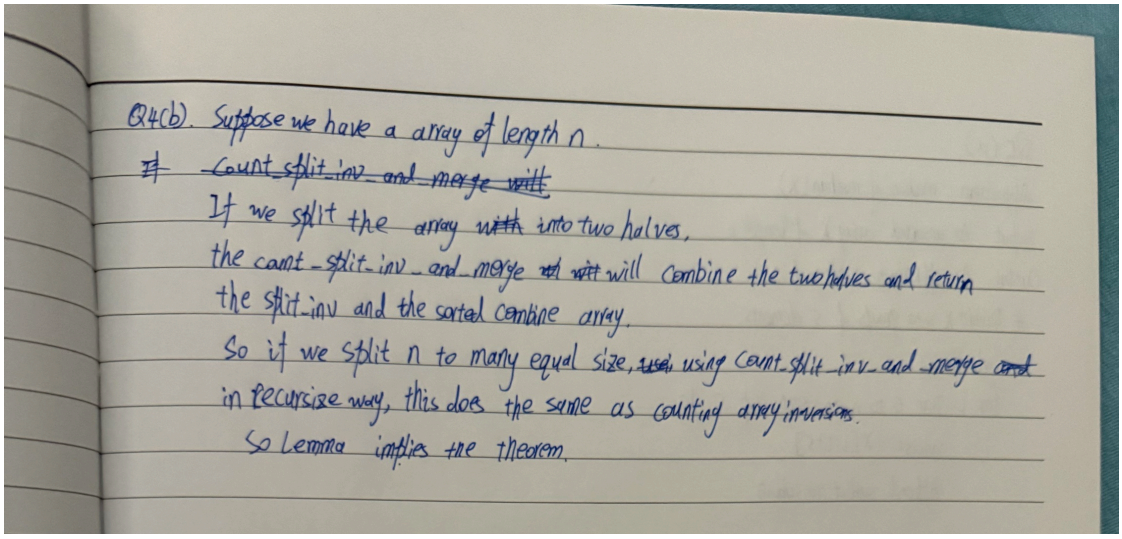
```
def naive_inversion_count(X:list):
# a Theta(n^2) approach for counting array inversions
    n = len(X)
    num_inv = 0
    for i in range(n):
        for j in range(i, n):
            if X[i]>X[j]:
                num_inv += 1

    return num_inv
```

```
import numpy as np
np.random.seed(2024) # set random seed
num_tests = 30
num_tests_failed=0 # initialise
for i in range(num_tests):
    test_list=np.random.rand(20)
    # if disagreement occurs a test has failed
    if naive_inversion_count(test_list) != count_inv_and_sort(test_list)[0]:
        num_tests_failed+=1
if num_tests_failed==0:
    print("All tests passed.")
else:
    print(f"There were {num_tests-num_tests_failed}\
    tests passed out of a total of {num_tests}")
```

```
All tests passed.
```

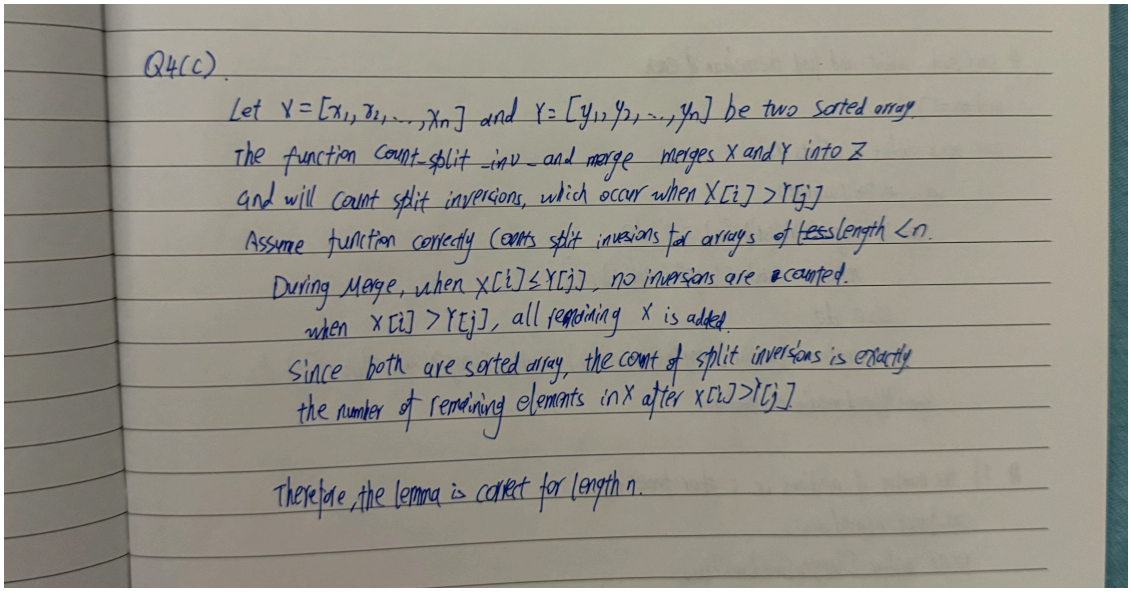
(b) (8 marks)

Prove that the Lemma on counting split inversions implies the Theorem on counting array inversions.



(c) (8 marks)

Prove the Lemma on counting split inversions.



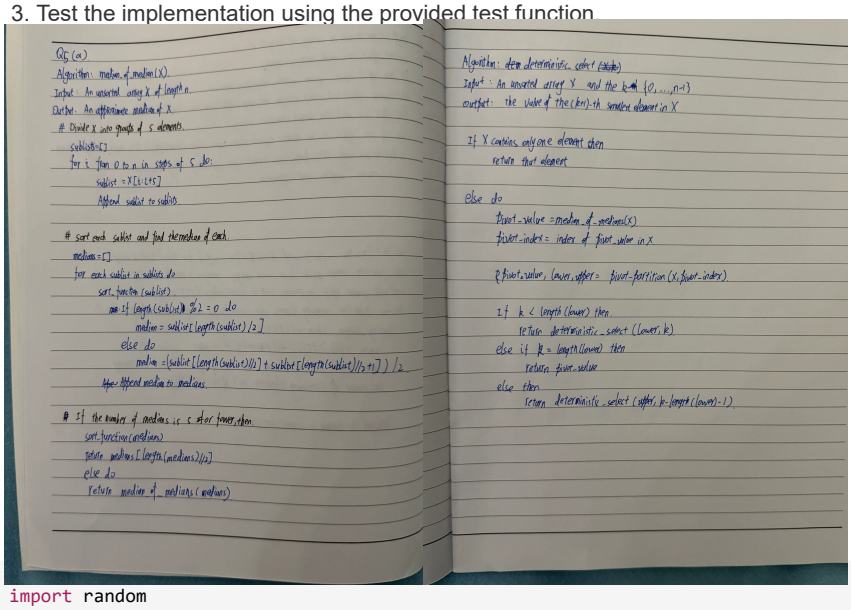
Question 5: Selection Problem (10 marks)

(a) (10 marks)

Create a deterministic algorithm to solve the selection problem in time $\Theta(n)$.

Steps:

1. Write the pseudo-code for the algorithm.
2. Implement it in Python as `deterministic_select`.
3. Test the implementation using the provided test function




```python
import random

def pivot_partition (X, pivot_index):
    pivot_value = X[pivot_index]
    lower = []
    upper = []
    n = len(X)
    for i in range(n):
        if(X[i] < pivot_value):
            lower.append(X[i])
        elif(X[i] > pivot_value):
            upper.append(X[i])
    return pivot_value, lower, upper
```

```
def sort_function (X):

    n = len(X)

    if (n < 2) :
        return X.copy()

    pivot_index = random.randint(0, n - 1)

    pivot_value, lower, upper = pivot_partition(X, pivot_index)

    lower = sort_function(lower)
    upper = sort_function(upper)

    Z = []
    for i in range( len(lower) ):
        Z.append(lower[i])

    for i in range( len(lower), n - len(upper) ):
        Z.append(pivot_value)

    for i in range( n - len(upper), n):
        Z.append(upper[i - n + len(upper)])

    return Z
```

```
def median_of_medians(X):

    # Divide X into sublists of 5 elements each
    sublists = [X[i:i+5] for i in range(0, len(X), 5)]

    # Sort each sublist and find the median
    medians = [sort_function(sublist)[len(sublist) // 2] for sublist in sublists]


    # If there are 5 or fewer medians, sort and return the median
    if len(medians) <= 5:
        return sort_function(medians)[len(medians) // 2]


    # Otherwise, recursively find the median of medians
    return median_of_medians(medians)
```

```
def deterministic_select(X, k):

    # Base case: only one element in the list
    if len(X) == 1:
        return X[0]

    # Find a good pivot using the median of medians
    pivot_value = median_of_medians(X)
    pivot_index = np.where(X == pivot_value)[0][0]


    # Partition the array around the pivot using the pivot_partition function
    pivot_value, lower, upper = pivot_partition(X, pivot_index)

    # Determine where the k-th smallest element is
    # k-th element is in the lower partition
    if k < len(lower):
        return deterministic_select(lower, k)

    # k-th element is the pivot itself
    elif k == len(lower):
        return pivot_value

    # Adjust k for the upper partition
    else:
        return deterministic_select(upper, k - len(lower) - 1)
```

```
import numpy as np

def selection_function_test(selection_function, random_seed=2024, array_size=30, alpha=0.5, num_tests=50):

    np.random.seed(random_seed) # set the random seed

    output = []
    num_tests_failed = 0

    k_alpha = int(alpha*(array_size-1)) # choose k

    for i in range(num_tests):
        X=np.random.rand(array_size)
        if sorted(X)[k_alpha] != selection_function(X,k_alpha):
            num_tests_failed+=1
    if num_tests_failed==0:
        print(f"Success! All {num_tests} tests passed.")
    else:
        print(num_tests_failed,"/",num_tests," failed.")

selection_function_test(deterministic_select, alpha=0.3, num_tests=100, array_size=1000)
```

```
Success! All 100 tests passed.
```