Assessed coursework 1 for Algorithms and Machine Learning (MATH20017), Autumn 2024

Introduction

This document contains the questions for Part 1 of your assessed coursework for the unit Algorithms and Machine Learning (MATH20017). The marks for this coursework will count 10% towards your final grade.

Please contact henry.reeve@bristol.ac.uk with any questions regarding this document. Whilst I am unable to provide solutions in advance of all work being handed in, I can provide clarification.

Handing in your coursework

How you present your coursework is important. You should complete your coursework using either Google Colab, a Jupyter notebook, or Rmarkdown. Whichever approach you take you must submit all of the following:

(1) A pdf file in which all answers within your notebook are visible.
(2) The notebook used to create the file (either a `.ipynb` file or a `.rmd` file).
(3) Any other files required to create the pdf in (1) e.g. images.

Important: Answers lacking any one of (1),(2),(3) are missing may not be awarded marks.

For answers using mathematical notation you have two options:

1. Include your answers in the Python notebook directly via Tex;
2. Write your answers via pen and paper and include a photo within the final document.

Important: Please ensure that the answers are well presented and clearly readable. Failure to do so can lead to a substantial loss of marks.

Question 1 (16 marks)

In the lectures we introduced the Master-Method-Bound for Divide-and-Conquer algorithms. We consider algorithms which follow the "standard recurrence format.''

Definition (Standard recurrence format). *Let $T(n)$ be the run-time complexity for a divide and conquer algorithm for problems of size n. We say that T follows the standard recurrence format with parameters $a, b \in \mathbb{N}$ and $d \in [0, \infty)$ if there exists $n_0 \geq b$, $c > 0$ with $T(n) \leq c$ for $n \leq n_0$, and for $n > n_0$,*

$$T(n) \leq a \cdot T(\lceil n/b \rceil) + c \cdot n^d.$$

We have the following asymptotic bound on the time-complexity of divide and conquer algorithm following a standard recurrence format.

Theorem (The master method bound). *Suppose that T is the time-complexity of a divide-and-conquer algorithm and T follows the standard recurrence format with parameters $a \in \mathbb{N}$, $b \in \mathbb{N} \setminus \{1\}$ and $d \in [0, \infty)$. Then we have*

$$T(n) \leq \begin{cases} O\big(n^d \log(n)\big) & \text{if } a = b^d & \text{(Case 1)} \\ O\big(n^d\big) & \text{if } a < b^d & \text{(Case 2)} \\ O\big(n^{\log(a)/\log(b)}\big) & \text{if } a > b^d & \text{(Case 3).} \end{cases}$$

A useful stepping stone in proving "master method bound'' theorem is to first prove the following (Key lemma for master method bound).

Lemma (Key lemma for master method bound). *Suppose that T is the time-complexity of a divide-and-conquer algorithm and T follows the standard recurrence format with parameters $a \in \mathbb{N}$, $b \in \mathbb{N} \setminus \{1\}$ and $d \in [0, \infty)$. Choose $n_0 \geq b$, $c > 0$ with $T(n) \leq c$ for $n \leq n_0$, and for $n > n_0$,*

$$T(n) \leq a \cdot T(\lceil n/b \rceil) + c \cdot n^d.$$

Then we have

$$T(n) \leq c \cdot \left\{ a^{\lceil \log_b(n) \rceil} + b^{d \lceil \log_b(n) \rceil} \cdot \sum_{j=0}^{\lceil \log_b(n) \rceil} \left(\frac{a}{b^d} \right)^j \right\}.$$

Q1(a) (8 marks) Prove that the key lemma for master method bound implies the master method bound theorem stated above.

Q1(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 (10 marks)

In this question we consider the Monte Carlo method. Suppose you wish to simulate the generation of a vector of random variables $(U_0, U_1, \ldots, U_{n-1})$ consisting of n i.i.d. (independent and identically distributed) random variables such that each $U_i \sim \mathrm{Unif}([0, 1])$. You can do this in NumPy as follows:

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

Q2(a) (10 marks) Edit the code below (updates (1),(2),(3)) to create and apply a function which creates a Monte-Carlo estimate of π.

The result of applying your function should be accurate to 2 decimal places. You can check this by comparing with `np.pi`.

Note (1): Your code may take some time to run. You may wish to begin with a relatively small value of n for debugging purposes and then increase n to `n=10000000` once your code has been written.

Note (2): You can make use of the NumPy `np.mean` function if you wish.

```
def monte_carlo_pi(num_observations: int=1000,random_seed: int=0)-> float:

    # set random seed for reproducibility
    np.random.seed(random_seed)

    # initialise sequence of i.i.d uniform
    U=np.random.rand(num_observations)
    V=None # (1) Update so that V is also random vector
    # V should contain n=num_observations i.i.d. observations V_i
    # where each V_i drawn from a uniform distribution on [0,1]

    num_in_circle=0

    for i in range(num_observations):
      pass
      # (2) Update - Important details missing

    pi_estimate = None # (3) Update the output

    return pi_estimate
```

Use the following code to confirm that your function gives the correct answer to 2 decimal places. You may wish to begin with a smaller value for `num_observations` and only increase the value when you are happy with the rest of your solutions.

```
pi_estimate_mc = monte_carlo_pi(num_observations=10000000,random_seed=2024)

print("NumPy estimate: "+str(round(np.pi,2)))
print("Monte Carlo estimate: "+str(round(pi_estimate_mc,2)))
```

Question 3 (32 marks)

In the lectures we also discussed how we can adapt ideas from the quick-select algorithm to provide randomised approach to sorting an array: quick-sort. The psuedo-code is given below:

Algorithm 1: quick_sort

Input: A length n array X

if $n < 2$ **then**
  return $X.\text{copy}()$

Choose $\text{pivot\_index} \sim \text{Unif}(\{0, \ldots, n-1\})$ `// Choose random pivot`

$(\text{pivot\_value}, \text{lower}, \text{upper}) = \text{pivot\_partition}(X, \text{pivot\_index})$

`// partition around the corresponding pivot value.`

$\text{lower} = \text{quick\_sort}(\text{lower})$

$\text{upper} = \text{quick\_sort}(\text{upper})$

`// make two recursive function calls to quick_sort`

$Z = []$ `// initialise empty array`

for $i \in \{0, \ldots, \text{len}(\text{lower}) - 1\}$ **do**
  $Z.\text{append}(\text{lower}[i])$

for $i \in \{\text{len}(\text{lower}), \ldots, n - \text{len}(\text{upper}) - 1\}$ **do**
  $Z.\text{append}(\text{pivot\_value})$

for $i \in \{n - \text{len}(\text{upper}), \ldots, n - 1\}$ **do**
  $Z.\text{append}(\text{upper}[i - n + \text{len}(\text{upper})])$

`// concatenate sorted sub-arrays`

return Z

Note that the quick sort algorithm calls a pivot_partition subroutine given by the following pseudo code.

Algorithm 2: pivot_partition

Input: A length n array X and an index $\text{pivot\_index} \in \{0, \ldots, n-1\}$

$\text{pivot\_value} = X[\text{pivot\_index}]$

$\text{lower} = []$, $\text{upper} = []$ `// initialise empty lower and upper arrays`

for $i \in \{0, \ldots, n-1\}$ **do**
  `// compare with pivot_value to choose partition element`
  if $X[i] < \text{pivot\_value}$ **then**
    $\text{lower.append}(X[i])$
  else if $X[i] > \text{pivot\_value}$ **then**
    $\text{upper.append}(X[i])$

return $(\text{pivot\_value}, \text{lower}, \text{upper})$.

Q3(a) (16 marks) First implement the quick-sort algorithm in Python. Once you have implemented the quick-sort function you should test your implementation using the `sorting_function_test` from the second computer lab. Include both your implementation and the test within your answer.

Q3(b) (16 marks) Prove the following theorem on the expected run time of the quick-sort algorithm. You are encouraged to try to adapt the proof on the expected run time of the quick-select algorithm from the lectures.

Theorem (quick sort): Suppose the quick_sort algorithm is applied to an array of length n. The expected run-time is $O(n \log n)$.

You may wish to begin by establishing the following lemma.

Given a numerical array $Z = [Z_0, \ldots, Z_{n-1}]$ is a numerical array we write $\mathcal{S}_\circ(Z)$ for the random run-time of the quick_sort algorithm applied to Z. Next, for $n \in \mathbb{N} \cup \{0\}$ let $T_\circ(n) := \max_Z \mathbb{E}\left[\mathcal{S}_\circ(Z)\right]$ denote the maximum expected run time, where the maximum is over all arrays Z of length at most n. Also let $T_\circ(0) := 0$.

Lemma: There exists a constant $C > 0$ such that $T_\circ(0) \leq C$ and for all $n \in \mathbb{N}$ we have

$$T_\circ(n) \leq Cn + \frac{2}{n} \sum_{j=0}^{n-1} T_\circ(j). \tag{1}$$

You should clearly explain your reasoning.

You can use the following inequality without proof: $\sum_{j=1}^{n-1} j \log j \leq \frac{n^2}{2}\left(\log n - \frac{1}{2}\right)$ for all $n \in \mathbb{N} \setminus \{1\}$.

Question 4 (32 marks)

Given a numerical array X of length n, an *array inversion* within X is an ordered pair $i < j$ with $i,j \in \{0, \ldots, n-1\}$ and $X[j] < X[i]$. In the lectures we discussed the problem of counting the number of array inversions within an array. In the lectures, we introduced the following count_inv_and_sort algorithm for counting the number of array inversions through a divide-and-conquer strategy.

Algorithm 3: count_inv_and_sort

Input: An array X

$n = \mathrm{len}(X)$
if $n < 2$ **then**
  return $(0, X)$
else
  $(\mathrm{l\_inv}, Y) = \mathrm{count\_inv\_and\_sort}(X[0 : \lfloor n/2 \rfloor])$
  $(\mathrm{r\_inv}, Z) = \mathrm{count\_inv\_and\_sort}(X[\lfloor n/2 \rfloor : n])$
  $(\mathrm{split\_inv}, W) = \mathrm{count\_split\_inv\_and\_merge}(Y, Z)$
  return $(\mathrm{l\_inv} + \mathrm{r\_inv} + \mathrm{split\_inv}, W)$

Notice that the count_inv_and_sort algorithm calls the following count_split_inv_and_merge function as a sub-routine.

Algorithm 4: count_split_inv_and_merge

Input: Sorted arrays X and Y

$n_X = \mathrm{len}(X)$, $n_Y = \mathrm{len}(Y)$
$Z = \mathbf{0}_n$, $\mathrm{split\_inv} = 0$, $i = j = 0$ `// initialisation`
for $k = 0$ *to* $n_X + n_Y - 1$ **do**
  if $i < n_X$ *and* ($j = n_Y$ *or* $X[i] \leq Y[j]$) **then**
    $Z[k] = X[i]$
    $i = i + 1$
  else
    $Z[k] = Y[j]$
    $\mathrm{split\_inv} = \mathrm{split\_inv} + n_X - i$ `// Add on the length of` $X[i : n_X]$
    $j = j + 1$
return $(\mathrm{split\_inv}, Z)$.

Q4(a) (16 marks) In Python, implement a function called `count_inv_and_sort` which takes as input a numerical array X (a Python list) with distinct elements and outputs a tuple containing both:

1. The number of array inversions in X.
2. A sorted array Y containing the same elements as X but occurring in ascending order.

Your function should have a worst-case run-time complexity of $O(n \log(n))$.

Once you have implemented your algorithm, test your `count_inv_and_sort` function's ability to count array inversions as follows: First implement the `naive_inversion_count` algorithm as follows (over page):

```python
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
```

Once you have implement the `naive_inversion_count` algorithm use test code below which compares the answers from the two functions.

```python
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}")
```

Now let's turn to the task of proving that $\mathrm{count\_inv\_and\_sort}$ algorithm solves the inversion counting problem. Let's consider the following results.

Theorem (Counting array inversions): Suppose X is an array consisting of n distinct numbers. Then the $\mathrm{count\_inv\_and\_sort}$ algorithm returns a tuple $(\mathrm{n\_inv}, \tilde{X})$ where $\mathrm{n\_inv}$ is the number of array inversions within the array X and $\tilde{X}$ is a sorted copy of the array X.

A useful step in proving this theorem is the following lemma.

Lemma (Counting split inversion): Suppose X is a sorted array containing n_X distinct numbers and Y is a sorted array containing n_Y distinct numbers. Then the $\mathrm{count\_split\_inv\_and\_merge}$ function will return a tuple $(\mathrm{split\_inv}, Z)$ where Z is a sorted array containing the $n_x + n_Y$ distinct elements within the arrays X and Y, and $\mathrm{split\_inv}$ is the number of split inversions across X and Y. More precisely, $\mathrm{split\_inv}$ consists of the number of ordered pairs $(i, j) \in \{0, \ldots, n_X\} \times \{0, \ldots, n_Y\}$ such that $X[i] > Y[j]$.

Q4(b) (8 marks) Prove the that the Lemma (Counting split inversion) implies the Theorem (Counting array inversions).

Q4(c) (8 marks) Prove the Lemma (Counting split inversion).

Question 5 (10 marks)

In the lectures we introduced the following selection problem.

Problem: The selection problem

Input: An (unsorted) numerical array X of length n and $k \in \{0, \ldots, n-1\}$.

Output: The value of the $(k+1)$-th smallest element in X.

We discussed an elegant randomised approach to solving this problem with a worst-case expected complexity of $\Theta(n)$ for arrays of size at most n.

Q5(a) (10 marks) Can you create a deterministic algorithm which solves the selection problem in time $\Theta(n)$ for arrays of size at most n?

1. First write the pseudo code for such an algorithm entitled `deterministic_select`.
2. Second implement your algorithm in Python and call it `deterministic_select`. The first argument should be the input array X and the second the number $k \in \mathbb{N}$.
3. Test your `deterministic_select` function as described below.

In part 1. you may assume the existence of a deterministic sorting function called sort_function with a worst case time complexity of $\Theta(m \log(m))$ when called with an array of length $m \in \mathbb{N}$. You may also assume the existence of a function $\{$pivot_partition$\}$ which solves the partition problem in $\Theta(n)$ time.

In part 2. you can call Python's `sorted` function as a sub-routine, as well as the `pivot_partition` function from Question 3. Note that Python's `sorted` function is $\Theta(m \log(m))$ when called with an array of length $m \in \mathbb{N}$ (not $Theta(m)!$).

In part 3. you should test your function `deterministic_select` with the following test function (`selection_function_test`).

```python
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.")
```

Next apply the `selection_function_test` to your `deterministic_select` function as follows.

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

End of coursework.