COM2031 - Advanced Algorithms Deadline Weds 10th November 2021 – 16:00

COM2031 Coursework, Autumn Semester 2021

Professor Steve Schneider
Department of Computer Science
University of Surrey

Note

- **The latest version of this handout is available on SurreyLearn. Check that you are working with the latest available version before submission.**
- **The coursework is individualised. You need to run a python script to get your individual sets of values.**

Deadline.

Your submission has to be received in SurreyLearn by

Weds 10th November 2021 – 16:00.

Instructions

- Answer both questions.
- Marks are awarded on an all-or-nothing basis per sub-question/text field, marks are only given for fully correct answers per sub-question/text field.
- A Python script `individual_values.py` is provided for you to generate your individual values for the two questions.
- The script can be run on the desktop machines in the Computer Science lab as follows:

```
python3 individual_values.py <URN>
```

  where `<URN>` is your individual URN.
- In order to run this on your own computer, you will have to have Python 3 installed plus additional python packages such as `numpy` — Consult the web on how to install these.

- Marking will be automated, ie a python script extracts your answers, and checks them against template answers or tests some of their properties as appropriate.
- Feedback will consist in the following:
  - My marking script produces a report on which answers are not correct.
  - A detailed sample solution will be provided when marks are returned.
- Submit your complete PDF file on SurreyLearn. Filename: `coursework_<username>.pdf` where `<username>` is your username, eg `xy0123`.
- You need to use a current version of the *original* Acrobat reader to fill this form in. Other PDF viewers are likely not to allow you to save the document with any contents in the active text fields. No current Acrobat reader version is supported on Linux.
- **Do not flatten the PDF document** (eg by printing it to a file) as this will make it impossible to extract your answers automatically which will result in a zero mark.
- Answers that require floating point numbers need to have three digits after the decimal dot with rounding of the last digit.
- For the sake of fairness and so that all of you have exactly the same information, I can't answer individual question on the coursework by email — instead please submit them to the SurreyLearn forum.
- Lists of values will be entered as comma-separated list as follows:

  Empty List: Leave the corresponding field empty.

  List with one item: Just enter the item without any commas

  List with more then one item: separate by comma like `a,b,c,d`
- Exercises must be solved *individually*, without consultation with other students. Students are advised to read the exercise questions carefully.

Marks Marks can be gained as follows:

- Question 1: up to 50 marks
- Question 2: up to 45 marks
- Compliance with submission instructions: up to 5 marks.

Individualisation

This coursework is individualised based on your URN, that is you will all do the same coursework, but everyone gets different numbers to start from. In order to get your individual numbers for Q1 and Q2 you will need to run the script `individual_values.py` under Python. It will give you a set of 8 2D-points for Q1 and a set of 8 intervals for Q2.

Failure to use your individual numbers as output by the script or manipulating the script will lead to loss of all marks.

Student Information

Enter the following information:

First Name:	Joshua
Last Name:	Everett
URN:	6621182
Username:	je00452

Declaration of Originality

I confirm that the submitted work is my own work. No element has been previously submitted for assessment, or where it has, it has been correctly referenced. I have also clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook. I agree that the University may submit my work to means of checking this, such as the plagiarism detection service Turnitin(R) UK. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

Note that "plagiarism" also includes collusion between students.

I confirm that I abide by the Declaration of Originality: ✔

1 2D Minimal Distance

This exercise is about applying one of the algorithms from the lecture on a small scale by hand *to experience how a divide-and-conquer algorithm works. In your solutions, refer to the points given by their index i (as given by the script) rather than just by their coordinates. It is essential that you follow the algorithm minutely.*

Consider the following algorithm:

Algorithm 1 Algorithm to calculate the minimal distance between 2D points

procedure MINDIST(list of points, sorted by x-coordinate.)
  if one point in the list **then**
    return infinity
  else if two points in the list **then**
    return Euclidean distance of the two points
  end if
  Split list of points into left and right halves by x-coordinate.
  Calculate x-coordinate of separation line L between left and right halves as the average of the x-coordinates of right-most point in the left half and the left most point on the right half.
  δ_1 =MINDIST(left half)
  δ_2 =MINDIST(right half)
  $\delta = \min(\delta_1, \delta_2)$
  Take all points less than δ distance from separation line L.
  Sort these points by y-coordinate.
  Scan these points in y-order and calculate Euclidean distance between each point and next 11 points (if any) by index.
  if any of these distances is less than δ **then**
    update δ
  end if
  return δ
end procedure

Using this algorithm, find the *minimal distance* of points within a set given S of 2D-points. Obtain your personal set of points as follows (not using this set will lead to loss of all marks):

1. Download the Python3 script `individual_values.py` to one of the Linux machines in the Computer Science lab.

2. Run the script as described earlier.

3. The script will print a list of 8 points to the screen – this is the set S of points you need to calculate with. Multiple runs of the script will produce exactly the same number with a given URN.

4. **Note:** Assistance running the scripts will be given in the labs in Week 5 and 6.

5. Whenever we refer to the index i of a point, we always mean the **original** index of the point, that is the one in the list that this script is printing out.

Calculate the minimal distance within your individual set of points following the above algorithm and enter the following selected intermediate results into the form fields below:[1]

1. Comma-separated indices of points when sorted by x-coordinate: 6,4,2,1,7,0,5,3 **Marks: 3** sorted

2. x-coordinate of dividing line L in the first call to `MinDist` (first level of recursion): $L =$ -0.5 **Marks: 4** Line

3. Comma-separated indices of points to the left of L: $P_L =$ 6,4,2,1 PL **Marks: 1**

4. Comma-separated indices of points to the right of L: $P_R =$ 7,0,5,3 PR **Marks: 1**

5. x-coordinates of the dividing lines L_L and L_R for the two calls to `MinDist` in the second level of recursion:
   $L_L =$ -3.5 **Marks: 4** LineL
   $L_R =$ 2.5 **Marks: 4** LineR

6. Comma-separated indices of points to the left of L_L: $P_{LL} =$ 6,4 PLL **Marks: 1**

7. Comma-separated indices of points to the right of L_L but left of L: $P_{LR} =$ 2,1 **Marks: 1** PLR

8. Comma-separated indices of points to the left of L_R but right of L: $P_{RL} =$ 7,0 **Marks: 1** PRL

9. Comma-separated indices of points to the right of L_R: $P_{RR} =$ 5,3 PRR **Marks: 1**

10. Comma-separated indices i of the points S_L that are in the δ-strip around L_L (leave empty if none): 4,2 **Marks: 8** SL

11. Minimal distance δ_L between all points to the left of L: **Marks: 2**
    $\delta_L =$ 5 dL

[1] We will need to calculate more intermediate values (such as the δs resulting for the different splits etc) – but I am asking only for the ones that are crucial for inferring whether you followed the right algorithm.

12. Comma-separated list of indices i of the points S_R that are in the δ-strip around L_R (leave empty if none): 2,1,7,0,5,3 **Marks: 8** SR

13. Minimal distance δ_R between all points to the right of L: **Marks: 1**
$\delta_R =$ 1.414 dR

14. Comma-separated list of indices i of the points that are in the δ-strip around L (leave empty if none): 1,7 **Marks: 8** s

15. Overall minimal distance $\delta =$ 1.414 **Marks: 2** d

Solution

- This assignment will be assessed using automated marking.
- Each student gets their own starting values
- Answers are marked automatically, using a Python script `points2d.py` on my side that compares the submitted student answers with the correct answers calculated. This script will be released to the students as part of the feedback.
- As it is too easy to copy an end result for example using a brute force approach without understanding the given algorithm, most marks are given for intermediate results that only the given algorithm but not a brute force algorithms calculates which primes the student to engage with the given algorithm.
- For the solution it is important to realise that the division of the delta-strip into $\delta/2$ boxes is only important for the derivation and proof of the algorithm, but not for executing it (we only need to know that we need to check the nearest 11 points in y coordinate). – A computer calculating which points falls into which box explicitly would spent much more time with it than doing the max 11 distances!
- It is not so important to have the right numeric results (however also this is marked), but that you execute the algorithm, ie follow all the steps of it slavishly – like a computer would. This has been checked by collecting intermediate results.
- Also note that in the delta-strip you must calculate *all* distances of points whose indices in y-order are less than 11 apart as the points are ordered by y index.
- For calculating the position of the dividing line you need the *median* of the indices, not the *mean*. (In fact you do not even need to know what the median is – just split the sorted list into left and right half.)

- The algorithm does not remember whether it has checked/calculated a distance before. It would need a look-up table for that. And having this is not worth the computational time to entertain it. So the algorithm might calculate some of the distance twice , and so do you if you follow the algorithm.
- For your understanding, you need to distinguish conceptually between executing the algorithm, the visualisation of the algorithm, and the proof of the algorithm. Here we are only concerned with executing it (ie following a sequence of steps, a recipe without any short cuts). Visualisation helps for finding a proof or understanding the different computational steps, but is not part of executing it.

2 Weighted Interval Scheduling

Apply the weighted interval scheduling algorithms from the lecture. Again use the provided script `individual_values.py` to generate your individual set of intervals based on your URN.

Algorithm 2 Weighted Interval Scheduling Algorithm

Calculate the maximum weight set of compatible intervals:

 # Core of the algorithm:
procedure CALCOPT($n, p(1), \cdots p(n), v_1, \cdots, v_n$)
  $M[0] = 0$
  for $i = 1 \cdots n$ **do**
    $M[i] = \max(M[i-1], v_i + M[p(i)])$
  end for
  return M
end procedure
 # Postprocessing
procedure POSTPROCESS(n,M)
  $j = n$
  $S = \emptyset$
  while $j > 0$ **do**
    if $M[j] > M[j-1]$ **then**
      $S = S \cup \{j\}$
      $j = p(j)$
    else
      $j = j - 1$
    end if
  end while
  return S
end procedure
 # run the algorithm
Sort the n intervals given by (s_i, f_i, v_i) by finish times f_i so that $f_1 < f_2 < \cdots f_n$
Compute $p(1), p(2), \cdots, p(n)$
M = CALCOPT($n, p(1) \cdots p(n), v_1 \cdots v_n$)
S = POSTPROCESS(n,M)
 # S now contains the indices i of the selected intervals

As a reminder, $p(i)$ is the greatest index such that $f_{p(i)} \leq s_i$ or 0 if no such index exists. In other words, it is the index of the latest finishing interval whose finish time is still compatible with interval i's start time. Obtain your personal set of intervals by running `individual_value.py`. Then by following the algorithm, calculate the set of intervals of maximum value.

Enter below intermediate results in the course of executing the algorithm:

$p(1) =$ 0 **Marks: 2** p1

$p(2) =$ 0 **Marks: 2** p2

$p(3) =$ 0 **Marks: 2** p3

$p(4) =$ 0 **Marks: 2** p4

$p(5) =$ 1 **Marks: 2** p5

$p(6) =$ 1 **Marks: 2** p6

$p(7) =$ 1 **Marks: 2** p7

$p(8) =$ 6 **Marks: 2** p8

$M[1] =$ 8 **Marks: 3** M1

$M[2] =$ 8 **Marks: 3** M2

$M[3] =$ 8 **Marks: 3** M3

$M[4] =$ 8 **Marks: 3** M4

$M[5] =$ 15 **Marks: 3** M5

$M[6] =$ 15 **Marks: 3** M6

$M[7] =$ 15 **Marks: 4** M7

$M[8] =$ 19 **Marks: 4** M8

Comma-separated list of names (single capital letter as printed by script `individual_values.py` of selected intervals: C,A,F **Marks: 3** selected

Solution

- Each student gets their own starting values.
- Answers are marked automatically, using a Python script `WeightedIntervalScheduling.py` on my side that compares the submitted student answers with the correct answers calculated. This script will be released to the students as part of the feedback.
- Most marks are given for intermediate results that only the given algorithm but not a brute force algorithm calculates, so you are required to engage with the given algorithm.

For the solution it is important to

- follow the algorithms step by step
- calculate the $p(i)$ values.

- calculate the $M[i]$ or `OPT` values
- apply the postprocessing to extra the correct intervals with the help of M.

It is not so important to have the right numeric results (however also this is marked), but that you execute the algorithm, ie follow all the steps of it slavishly – like a computer would. This has been checked by collecting intermediate results.

For your understanding, you need to distinguish conceptually between executing the algorithm, the visualisation of the algorithm, and the proof of the algorithm. Here we are only concerned with executing it (ie following a sequence of steps, a recipe without any short cuts). Visualisation helps for finding a proof or understanding the different computational steps, but is not part of executing it.