COMS21103: Coursework 1

This is an individual coursework

2015/2016

Due: 26th November 2015 (by 23:59) via the online submission system (SAFE)

This coursework focuses on solving problems using dynamic programming and consists of four parts. Each part considers a different problem and is divided into a number of questions. Each question is either marked as theoretical [T], or implementation [I]. Parts 2, 3 and 4 contain both theoretical and implementation questions. Part 1 contains only implementation questions. The implementations are in Java.

Marks *The parts are not equally weighted.* The four parts have total marks of $45, 20, 15$ and 20 in that order. This gives a total mark out of 100. You shouldn't necessarily expect to do all the questions; part of the assessment is how far you manage to get in the available time. Remember that you may be able to do some of the later stages without completing the earlier ones. A submission could get over 60% on this coursework by focusing on giving good answers to the first two parts. There are intentionally no mark breakdowns for the parts. *This coursework is worth* 20% *of the unit.*

Plagiarism *This is an individual coursework.* You must work on this coursework individually for both the theoretical and implementation components. You must not discuss your submission with others. A major component of parts 2, 3 and 4 is to develop a recursive formula for the corresponding problem. This must be your own work. In accordance with departmental policy, any suspected cases of plagiarism will be referred to the plagiarism committee.

Implementation For each implementation question (marked with an [I]) we have provided detailed implementation instructions. You must follow the instructions *exactly.* Your implementations will be **automarked**. You are responsible for checking that your submission compiles cleanly on a standard lab machine and conforms precisely to the input/output specifications before you submit. Otherwise you are at risk of getting *zero* for that question. We are not doing this to be cruel, we simply do not have time to manually compile and run every submission individually. We will not be testing how your code handles malformed inputs. If you think the instructions are unclear, please ask for clarification via the Blackboard forum.

Examples For each part we have provided a number of examples in the correct format for testing purposes. For each example, we have provided the correct answer. These examples can be downloaded from,

`http://www.cs.bris.ac.uk/Teaching/Resources/COMS21103/DSAexamples.zip`

Memoization For some of the implementation questions you are asked to memoize a recursive algorithm. You must not use any memoization libraries or automated memoization methods. You must

memoize your recursive algorithm manually by modifying your code to explicitly store previously computed answers to subproblems. We believe that this is beneficial to your understanding of why and when memoization improves the time complexity of recursive algorithms.

Report Your answers to the theoretical components (marked with an [T]) must be submitted in a single PDF file with the (precise) name `DSAreport.pdf`. Your reports will **autoprinted**. If your report has the wrong filename or is not a PDF, you are at risk of getting *zero*. You must include your name and user id on the top of *every* page you submit. This is so that we can identify your report after it is printed. There is no written component for any question marked with an [I]. We recommend strongly against submitting handwritten scans. If you do, make sure that your scans and handwriting are legible.

Submission You must submit your coursework using the online submission system (SAFE). A submission that attempts every part will contain exactly the following five files:

`DSAP1.java` `DSAP2.java` `DSAP3.java` `DSAP4.java` `DSAreport.pdf`

Any additional files you submit may be deleted. If you have not attempted a particular part, simply omit the corresponding file(s). If you submit these files as a zip, you must unzip them yourself on SAFE (there is a button for this). Do not put the files in a subdirectory. *Remember that in Java if you rename a file, you also have to rename the class (i.e. don't rename the files at 23:58 on the 26th).*

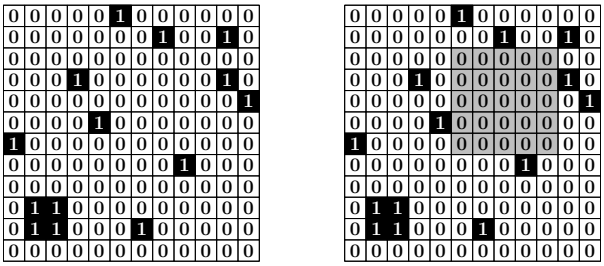
Exercises There are more dynamic programming problems on problem sheet 4. I encourage you to attempt them as practice. Solving dynamic programming problems is a skill and like any skill you can get better by practising. I strongly believe that this is a skill worth acquiring.

Reading You may also find Chapter 15 of CLRS or Chapter 5 of JE (available online and linked from the course webpage/Blackboard) helpful. These chapters contain explanations of solutions to several dynamic programming problems that we haven't discussed. You may find inspiration for the questions below by reading these.

Part 1 - The `Largest-Empty-Square` problem [45 marks]

This part concerns the LARGEST-EMPTY-SQUARE problem seen in the first lecture on Dynamic Programming. We begin by giving a formal definition of the problem for completeness (though you may prefer to look at the lecture instead). The input to the problem is an $n \times n$ array A. Each entry in the array $A[x, y]$ is either 0 or 1. In the lecture we visualised the array as a 2D monocrome image where $A[x, y]$ represents the pixel with coordinates (x, y). Pixel (x, y) is considered to be empty if and only if $A[x, y] = 0$. A square with side length ℓ is a 2D subarray $A[x' \ldots x, y' \ldots y]$ with $\ell = y - y' + 1 = x - x' + 1$. A square is empty iff every pixel in it is empty. The problem is to find the side length of the largest empty square in A.

Example We begin by giving a human-readable example. The formal input/output description (for the implementation) is at the end. Suppose that $n = 12$, in pictoral form, an example input A is given on the left below.



The correct output for this example is 5. In particular an empty square with side length 5 is highlighted (in grey) on the right above. You should verify for yourself that there is no square with side length 6 in the input.

Clarifications In the lecture, the array and coordinate system are one indexed. You can decide whether to zero-index or one-index. It's fine either way (as long as you are consistent) because it does not affect the output. There may be a tie for which empty square is largest in A. This does not affect the output. The correct output is the side length of the square and *not* its area, nor its location.

Submission There is no theoretical component to this first part. Your submission for this part must contain exactly one file named `DSAP1.java`. We will compile your submission using the following (single) command:

```
javac DSAP1.java
```

The input file format that we will use for testing is given after the questions.

Questions:

a) [I] Implement a (non-memoized) recursive algorithm for the LARGEST-EMPTY-SQUARE problem. Your implementation should use the recursive formula for the problem seen in the lecture.

Your implementation for this question must be run by the following command:

```
java DSAP1 -r <filename>
```

Here `<filename>` is the name of an input text file. The flag `-r` indicates that your submission is to run in (non-memoized) recursive mode. Your program must output one single integer on standard output (`stdout`). This integer should be the side length of the largest empty square in the input. You must not output anything else on standard output.

b) [I] Implement a (seperate) memoized version of your recursive algorithm. Your memoized algorithm should run in $O(n^2)$ time (i.e. linear in the number of pixels in A). As part of your testing you should compare your implementations for efficiency. In particular you should be able to find (relatively small) inputs for which the memoized algorithm completes in a couple of seconds while the non-memoized algorithm takes even longer to complete than it takes to log on to a university maintained windows machine (i.e. several minutes).

Your implementation for this question must be run by the following command:

```
java DSAP1 -m <filename>
```

Here `<filename>` is the name of an input text file. The flag `-m` indicates that your submission is to run in memoized recursive mode. Your program must output one single integer on standard output. This integer should be the side length of the largest empty square in the input. You must not output anything else on standard output.

c) [I] Implement an iterative algorithm for the LARGEST-EMPTY-SQUARE problem. Your iterative algorithm should also run in $O(n^2)$ time (i.e. linear in the number of pixels in A). As part of your testing you should also compare iterative implementation to both previous implementations.

Your implementation for this question must be run by the following command:

```
java DSAP1 -i <filename>
```

Here `<filename>` is the name of an input text file. The flag `-i` indicates that your submission is to run in iterative mode. Your program must output one single integer on standard output. This integer should be the side length of the largest empty square in the input. You must not output anything else on standard output.

Evaluation Your submission will be tested on a variety of inputs of different sizes (including some very large inputs). We will be testing for correctness for all questions and scalability for questions b) and c).

File format The file format for the LARGEST-EMPTY-SQUARE problem is as follows. The first line contains a single positive integer which gives the value of n. The subsequent lines give A a row at a time (without any spaces). In particular after the first line, there will be n lines, each containing n characters. Each character will be either `0` or `1` indicating whether that pixel is empty. The following gives the example input above in the correct file format. This file and several other examples can be downloaded from the course website (see the paragraph on **Examples** at the start of the document).

`P1eg1.txt`

```
12
000001000000
000000010010
000000000000
000100000010
000000000001
000010000000
100000000000
000000001000
000000000000
011000000000
011000100000
000000000000
```

To test this input on your recursive implementation, (assuming that `DSAP1` and `P1eg1.txt` are in the same directory) the command must be:

```
java DSAP1 -r P1eg1.txt
```

The program should output `5` to standard output (and nothing else).

Part 2 - The `Corner-Shop` problem [20 marks]

You are about to take over running a very small corner shop by the sea in Weston-super-Mare. You will run the shop for n days. As the shop is so small, you can only stock either umberellas or suncream (but not both) at any one time. The profit you make on the i-th day is determined by whether you have umberellas or suncream in stock. If you have umberellas in stock on the i-th day, you will make $U[i]$ pounds of profit. Alternatively, if you stock suncream on the i-th day you will make $S[i]$ pounds of profit.[1] However, switching from between stocking umberellas and suncream costs m pounds. You can choose to change stock at the end of any day (and the new stock is ready in time for the next day). You can choose which item is in stock at the start of day one (without any cost). Your goal is to determine the maximum amount of profit that you can obtain during the n days (after paying any restocking costs).

Example We begin by giving a human-readable example. The formal input/output description (for the implementation) is at the end. Suppose that $n = 5$ and $m = 10$, in table form, an example input is given by:

Day	1	2	3	4	5
$U[i]$	5	13	15	5	5
$S[i]$	30	20	2	14	14

One plan for the example given is to stock suncream for days 1 and 2, umbrellas for day 3 and then suncream again for days 4 and 5. This gives a total profit of $30+20+15+14+14-(10\cdot 2) = 73$. The final $-(10 \cdot 2)$ is the cost of switching stock twice. Another plan is to stock suncream for days 1 and 2 and umbrellas for days $3, 4$ and 5. This gives a total profit of $30+20+15+5+5-(10\cdot 1) = 65$. Again, the final $-(10 \cdot 1)$ is the cost of switching stock once.

Neither of these is the best possible. For this example, the best plan is to stock suncream every day. This gives a total profit of $30 + 20 + 2 + 14 + 14 = 80$. You should check that you agree that this is an optimal plan.

It is tempting to think that this problem can be solved by a greedy approach. One greedy approach would be to always stock the item on each day which gives the largest profit. Another greedy approach would be to switch stock overnight whenever it will increase your profit on the next day. Neither of these approaches correctly solves the problem in general. In particular these approaches correspond (respectively) to the two suboptimal plans above.

Clarifications You never run out of the item you have in stock. You cannot change which item you have in stock during the day. Both arrays U and S have length n.

Submission This part contains both a theoretical and an implementation component. Your answers to the theoretical questions (marked [`T`]) should be included in your report. Your implementation for this part must contain exactly one file named `DSAP2.java`. We will compile your implementation using the following (single) command:

```
javac DSAP2.java
```

[1] You may be surprised that your future profits can be determined so far in advance. Fortunately, weather forecasting became extremely accurate in 2015. Too bad the post office isn't as efficient.

The input file format that we will use for testing is given after the questions.

Questions:

a) [T] In this question you will develop a recursive formula for the CORNER-SHOP problem.

We define $P[i, x]$ to be the maximum profit that can be obtained if you only run the shop for the first i day(s) *and* you start the i-th day with item x in stock. Here x can take two values `u` (umbrellas) or `s` (suncream). Observe that $P[1, \mathtt{u}] = U[1]$ and $P[1, \mathtt{s}] = S[1]$ and the answer to the CORNER-SHOP problem is $\max\{P[n, \mathtt{u}], P[n, \mathtt{s}]\}$.

Give a recursive formula for $P[i, x]$ in terms of $P[i-1, \mathtt{u}], P[i-1, \mathtt{s}]$ and m.

b) [T] Explain why the recursive formula you gave in a) is correct. You do not have to give a formal proof.

c) [I] Implement a (non-memoized) recursive algorithm for the CORNER-SHOP problem. Your implementation should use the recursive formula you developed in a).

Your implementation for this question must be run by the following command:

```
java DSAP2 -r <filename>
```

Here `<filename>` is the name of an input text file. The flag `-r` indicates that your submission is to run in (non-memoized) recursive mode. Your program must output one single integer on standard output. This integer should be maximum amount of profit that can be obtained for the given input. You must not output anything else on standard output.

d) [I] Implement a (seperate) memoized version of your recursive algorithm. Your memoized algorithm should run in polynomial time. As part of your testing you should compare your implementations for efficiency. Your implementation for this question must be run by the following command:

```
java DSAP2 -m <filename>
```

Here `<filename>` is the name of an input text file. The flag `-m` indicates that your submission is to run in memoized recursive mode. The output should be the same as in the previous question.

e) [I] Implement an iterative algorithm for the CORNER-SHOP problem. Your iterative algorithm should also run in polynomial time. As part of your testing you should compare your iterative implementation to both previous implementations.

Your implementation for this question must be run by the following command:

```
java DSAP2 -i <filename>
```

Here `<filename>` is the name of an input text file. The flag `-i` indicates that your submission is to run in iterative mode. The output should be the same as in the previous two questions.

f) [T] What is the time complexity of your iterative algorithm? With reference to your code or otherwise, argue that the time complexity that you have given is correct.

Evaluation Your implementations will be tested on a variety of inputs of different sizes (including some very large inputs). We will be testing for correctness for all three implementation questions and scalability for questions c) and e). For the theoretical questions we will be partly marking based on how well you justify your answers. You will also be marked based on the time complexities given. Algorithms with better time complexities will achieve better marks.

File format The file format for the CORNER-SHOP problem is as follows. The input file contains three lines. The first line contains two positive integers: The first number gives the value of n and the second gives the value of m. Both the second and third lines contain exactly n integers. The second line gives the array U. The entries of U are given in day order. The first integer is $U[1]$. The third line gives the array S. The entries of S are also given in day order. The first integer is $S[1]$. Numbers are separated by white space. To make the examples human readable the number of spaces between each pair of numbers may vary in the input. You should allow for this in your implementation. The following gives the example input above in the correct file format. This file and several other examples can be downloaded from the course website (see the paragraph on **Examples** at the start of the document).

P2eg1.txt

```
1 5 10
2 5  13 15 5  5
3 30 20 2  14 14
```

Part 3 - The `Magic-Tournament` problem [15 marks]

The magicians guild of Weston-super-Mare has decided to enter a team into the next magic tournament! The head magician has asked you to help select the team under the following rules.

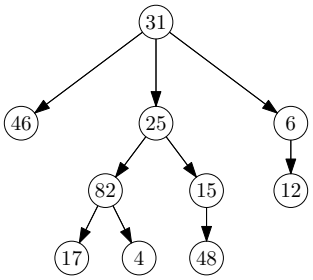
In preparation for the tournament, each magician has been tested and assigned a magical ability score. The magical ability of the team is equal to the sum of the magical abilities of the team members.

The guild contains n magicians and is organised into a strict hierarchy. That is it forms a tree with the head magician at the root. Each magician can have any number of apprentices (including zero) and those apprentices may in-turn have apprentices (and so on). There is no connection between the magical ability of a magician and their position in the guild hierarchy (typical!).

To avoid potential arguments, the head magician has also declared that the team cannot contain both a magician and one (or more) of their apprentices. Aside from this there are no other limits on the team. In particular, there is no limit on the number of team members.

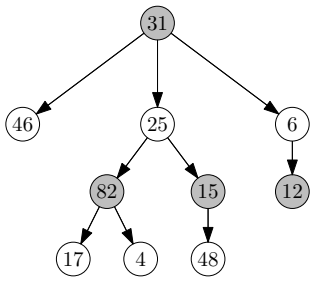
The goal is to determine the valid team with the highest magical ability. The output should be the ability of that team.

Example We begin by giving a human-readable example. The formal input/output description (for the implementation) is at the end. Suppose that $n = 10$, in tree form an input is given by:

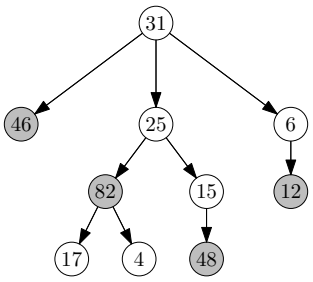


Each magician is represented by a vertex which contains their magical ability score. There is an arrow from each magician to each of their apprentices. The following gives three possible teams (including an invalid team). The team members are the grey vertices.

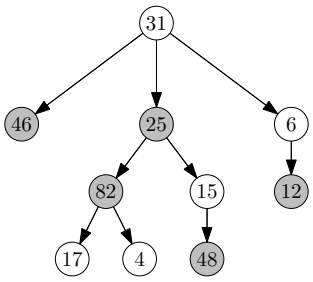
valid team 1:



valid team 2:



an invalid team



In the above, the leftmost two teams are valid but the rightmost team is invalid. This is because it contains both the magician with ability 25 and one of his apprentices (the one with ability 82). Valid team 1 has total ability $31+82+15+12=140$ and valid team 2 has ability $46+82+48+12=188$. In fact, valid team 2 has the highest possible ability (of any valid team). The output should be 188.

Clarifications The magical ability of a magician will be a positive integer (but may be arbitrarily large). The number of apprentices that a magician may have is not bounded. The depth of the hierarchy is not bounded.

Submission This part contains both a theoretical and an implementation component to this part. Your answers to the theoretical questions (marked [T]) should be included in your report. Your implementation for this part must contain exactly one file named `DSAP3.java`. We will compile your implementation using the following (single) command:

```
javac DSAP3.java
```

The input file format that we will use for testing is given after the questions.

Questions:

a) [T] Give a recursive formula for the MAGIC-TOURNAMENT problem.

Hint: Consider a team containing magician (vertex) v and only other magicians in the subtree rooted at vertex v.

b) [T] Explain why the recursive formula you gave in a) is correct. You do not have to give a formal proof.

c) [I] Implement an algorithm for the MAGIC-TOURNAMENT problem. Your implementation should use the recursive formula you developed in the last question.

Your implementation for this question must be run by the following command:

```
java DSAP3 <filename>
```

Here `<filename>` is the name of an input text file. Your program must output one single integer on standard output. This integer should be the highest achievable team ability for the given input. You must not output anything else on standard output.

d) [T] What is the time complexity of your algorithm? Is memoization useful for this problem? Discuss your answer.

Evaluation Your implementations will be tested on a variety of inputs of different sizes (including some very large inputs). We will be testing for correctness and scalability. For the theoretical questions we will be partly marking based on how well you justify your answers. You will also be marked based on the time complexities given. Algorithms with better time complexities will achieve better marks.

File format The file format for the MAGIC-TOURNAMENT problem is as follows. The first line contains a single positive integer which gives the value of n. Each subsequent line encodes a magician. Each magician has a unique id number between 1 and n. The magicians will always appear in increasing id order in the input. The head magician always has id 1. The first number on a line gives the id number of the magician. This is spurious but it makes it easier for humans to read the file. The second number is the magical ability of the magician which is then followed by a colon. After this colon is a list of all of this magician's apprentices. Each apprentice is given by their id number and they are listed in ascending order. A magician with no apprentices will have no numbers after the colon. Numbers are separated by a variable amount of white space. The following gives the example input above in the correct file format. This file and several other examples can be downloaded from the course website (see the paragraph on **Examples** at the start of the document).

P3eg1.txt

```
10
1  31 : 2  3  4
2  46 :
3  25 : 5  6
4  6  : 7
5  82 : 8  9
6  15 : 10
7  12 :
8  17 :
9  4  :
10 48 :
```

Part 4 - The `Day-Off` problem [20 marks]

As you did such an excellent job running the corner shop, a local sandwich delivery person asks you to take over for her while she is away for an n day holiday. She warns you that delivering sandwiches is extremely tiring and advises you to take a day off to recover once in a while. You can take off whichever days you like.

Before she leaves she gives you her order book which gives the number of sandwiches $B[i]$ that she has orders for on the i-th day. She also gives you an extremely long and boring book of EU regulations regarding delivery work. After much reading, you discover that the maximum number of sandwiches you are allowed to deliver depends on the number of days since your last day off. Specifically, if the last day off was j day(s) ago then the maximum number of sandwiches you are legally allowed to deliver is $M[j]$. You can assume that $M[j] \leq M[j-1]$ for all j.

If you work on the i-th day then the number of sandwiches you will deliver is $\min\{B[i], M[j]\}$. Here j is the number of days since your last day off. If you have the i-th day off then you deliver 0 sandwiches.

You goal is to maximise the total number of sandwiches that you deliver by choosing which days you take off.

Example We begin by giving a (human readable) example. The formal input/output description (for the implementation) is at the end. Suppose that $n = 4$, in table form, an example input is given by:

Day	1	2	3	4
$B[i]$	10	1	7	7
$M[i]$	8	4	2	1

The best plan is to take a day off only on day 2. This allows you to deliver 8 sandwiches on day one, 0 on day two, 7 on day three and 4 on day four. This gives a total of 19 sandwiches delivered. If you didn't take any days off, you would deliver only $8 + 1 + 2 + 1 = 12$ sandwiches.

Clarifications You should assume that you had the day off before you started running the delivery service. If your last day off was yesterday than the maximum number of sandwiches you are legally allowed to deliver is $M[1]$. For all j, $M[j] \geq 0$. Both arrays B and M have length n. You can take as many days off as you like. The rest of the EU regulations were not relevant so we haven't reproduced them here. If you want to know more about the EU regulations regarding delivery work, please feel free to ask in the drop-in session.

Submission This part contains both a theoretical and an implementation component to this part. Your answers to the theoretical questions (marked [`T`]) should be included in your report. Your implementation for this part must contain exactly one file named `DSAP4.java`. We will compile your implementation using the following (single) command:

```
javac DSAP4.java
```

The input file format that we will use for testing is given after the questions.

Questions:

a) [T] Give a recursive formula for the DAY-OFF problem.

b) [T] Explain why the recursive formula you gave in a) is correct. You do not have to give a formal proof.

c) [T] Explain how to obtain the output to the DAY-OFF problem from the recursive formula you gave in a).

d) [I] Implement a (non-memoized) recursive algorithm for the DAY-OFF problem. Your implementation should use the recursive formula you developed in the last question.

Your implementation for this question must be run by the following command:

```
java DSAP4 -r <filename>
```

Here `<filename>` is the name of an input text file. The flag `-r` indicates that your submission is to run in (non-memoized) recursive mode. Your program must output one single integer on standard output. This integer should be maximum number of sandwiches that can (legally) be delivered for the given input. You must not output anything else on standard output.

e) [I] Implement a (seperate) memoized version of your recursive algorithm. Your memoized algorithm should run in polynomial time. As part of your testing you should compare your implementations for efficiency.

Your implementation for this question must be run by the following command:

```
java DSAP4 -m <filename>
```

Here `<filename>` is the name of an input text file. The flag `-m` indicates that your submission is to run in memoized recursive mode. The output should be the same as in the previous question.

f) [T] What is the time complexity of your memoized algorithm? With reference to your code or otherwise, argue that the time complexity that you have given is correct.

g) [I] Implement an iterative algorithm for the DAY-OFF problem. Your iterative algorithm should also run in polynomial time. As part of your testing you should also compare your iterative implementation to both previous implementations.

Your implementation for this question must be run by the following command:

```
java DSAP4 -i <filename>
```

Here `<filename>` is the name of an input text file. The flag `-i` indicates that your submission is to run in iterative mode. The output should be the same as in the previous two questions.

h) [T] What is the time complexity of your iterative algorithm? With reference to your code or otherwise, argue that the time complexity that you have given is correct.

Hint: There are several ways to formulate this problem as a recursive formula. Have you considered whether you can obtain a better time complexity with a different recursive formula?

Evaluation Your implementations will be tested on a variety of inputs of different sizes (including some very large inputs). We will be testing for correctness for all three implementation questions and scalability for questions e) and g). For the theoretical questions we will be partly marking based on how well you justify your answers. You will also be marked based on the time complexities given. Algorithms with better time complexities will achieve better marks.

File format The file format for the DAY-OFF problem is as follows. The input file contains three lines. The first line contains a single positive integer which gives the value of n. Both the second and third lines contain exactly n integers. The second line gives the array B. The entries of B are given in day order. The first integer is $B[1]$. The third line gives the array M. The entries of M are given in increasing order. The first integer is $M[1]$. Numbers are separated by a variable amount of white space. The following gives the example input above in the correct file format. This file and several other examples can be downloaded from the course website (see the paragraph on **Examples** at the start of the document).

P4eg1.txt

```
4
10 1 7 7
8  4 2 1
```