Assignment 2

General

General

You must read fully and carefully the assignment specification and instructions.

- **Course:** COMP20007 Design of Algorithms @ Semester 1, 2023
- **Deadline Submission:** Friday 26th May 2023 @ 11:59 pm
- **Course Weight:** 20%
- **Assignment type:** individual
- **ILOs covered:** 1, 2, 3, 4
- **Submission method:** via ED

Purpose

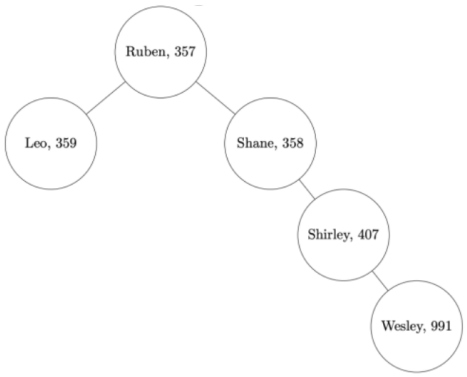
The purpose of this assignment is for you to:

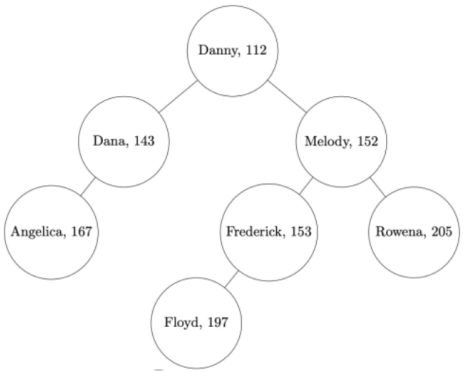
- Design efficient algorithms in pseudocode.
- Improve your proficiency in C programming and your dexterity with dynamic memory allocation.
- Demonstrate understanding of data structures and designing and implementing a set of algorithms.

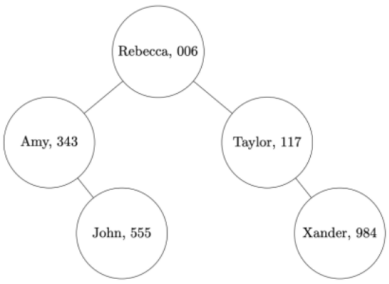
Question 1: Quasi-balanced Search Trees

Quasi-balanced Search Trees

In the following diagram, which we call a *quasi-balanced search tree (QUBSET)*, every node represents a student. Each student has a name and each student has a student number, both of which are shown in each node. Every diagram shown is structured by the same set of rules. *You can assume for the whole question that all names/IDs are unique.*







Part A

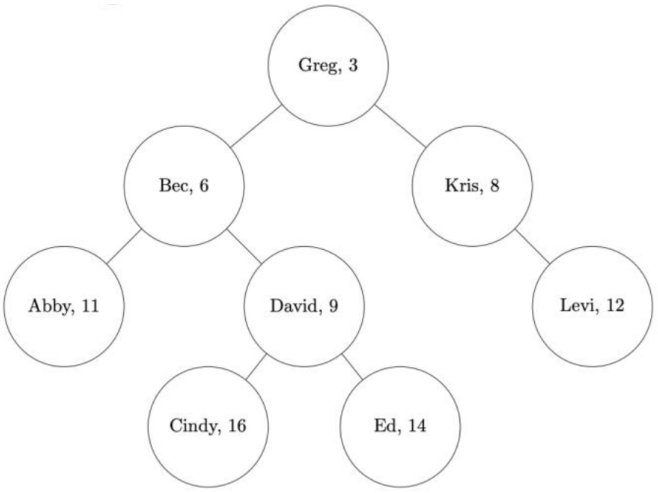
i. If you remove the student IDs and only consider the names, what do you notice about the order/structure? Explain your answer. **0.5 marks**

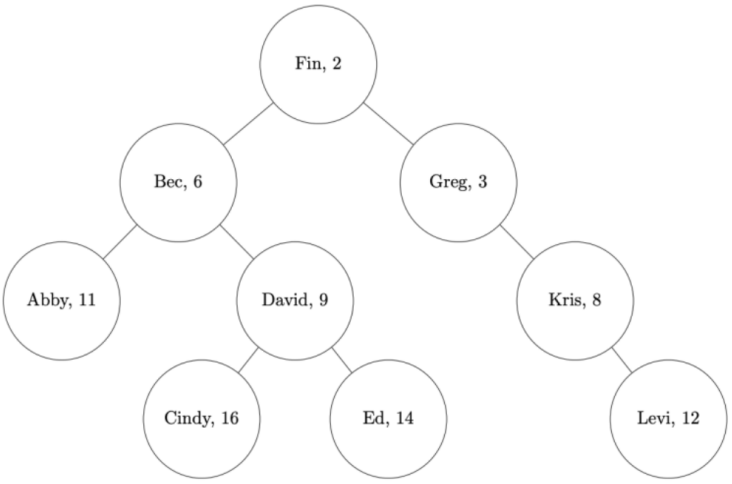
ii. If you remove the names and only consider the student IDs, what do you notice about the order/structure? Explain your answer. **0.5 marks**

At the bottom of this page, we provide two examples that show what happens when we add a new student (node) to our diagram.

Part B

If we add the student Fin who has a student ID of 2, the *QUBSET* changes from left to right.





Write down all the missing steps in this process. You should provide just as much detail as the examples shown at the bottom. **2 marks**

Part C

Given an arbitrary *QUBSET*, T, and a new student S, write a new function add_student(T, S) that adds the new student to the diagram. Assume the operation is done in place (there should be no return value). You can assume T_{name} and T_{id} give the student name/ID respectively. T and S are of the same type and you can assume S has no children. Your pseudocode should look like the pseudocode that is given in Lecture 9. **3 marks**

Part D

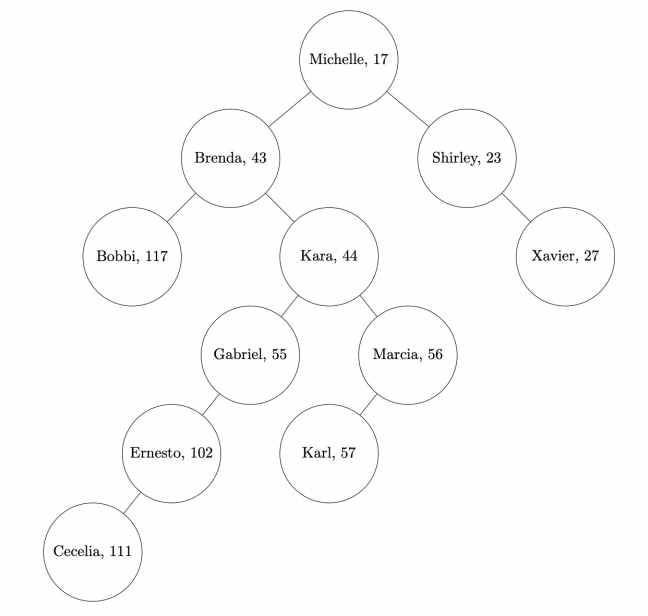
We want the height of *QUBSET* to be as small as possible. Give an example of the worst case height when we add 5 students to an empty *QUBSET*. **1 mark**

Part E

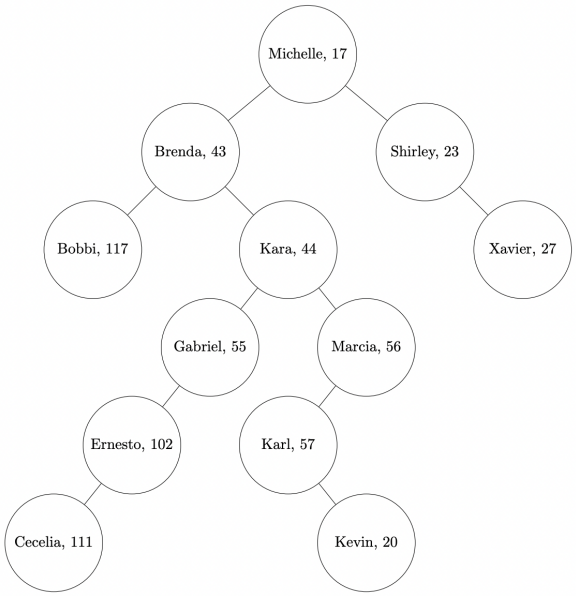
Suppose we have a group of n students that are listed in alphabetical order. If we add them to a binary search tree (sorted just by name and ignoring their student IDs), it can be shown that it degenerates to a tree of height n. Explain why the *QUBSET* we have used in this question is likely to have a height much smaller than n. **1 mark**

Example 1

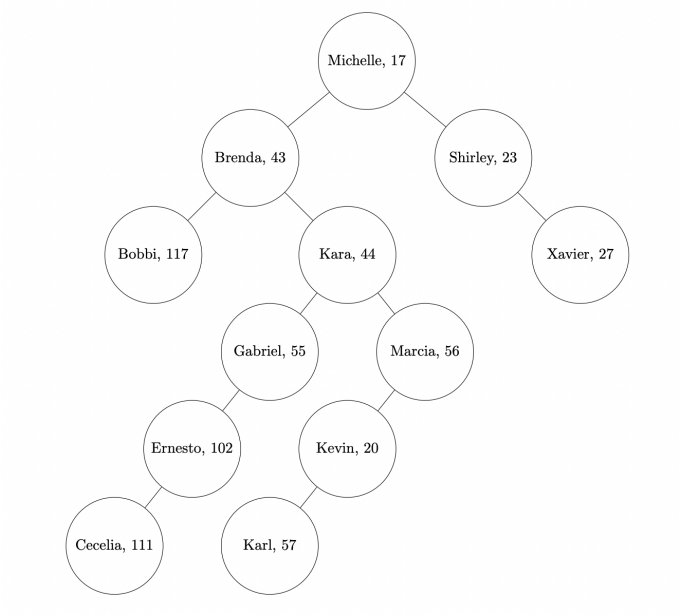
We start with the following *QUBSET*

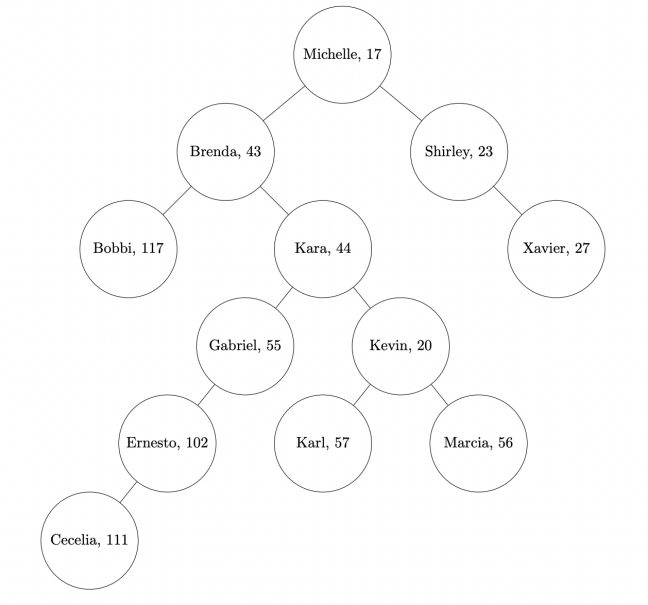


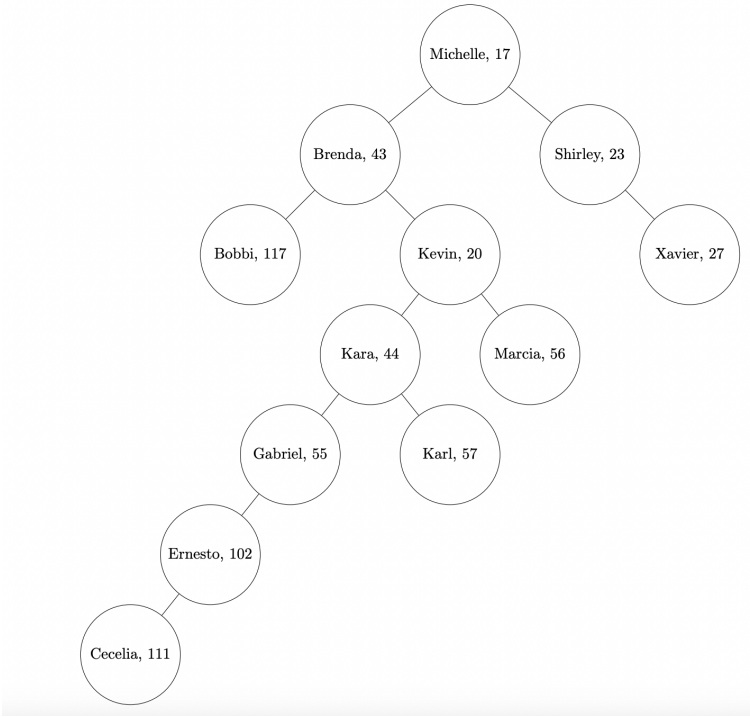
Now suppose we add Kevin who has a student ID of 20 to the *QUBSET* following the rule in part a) i.

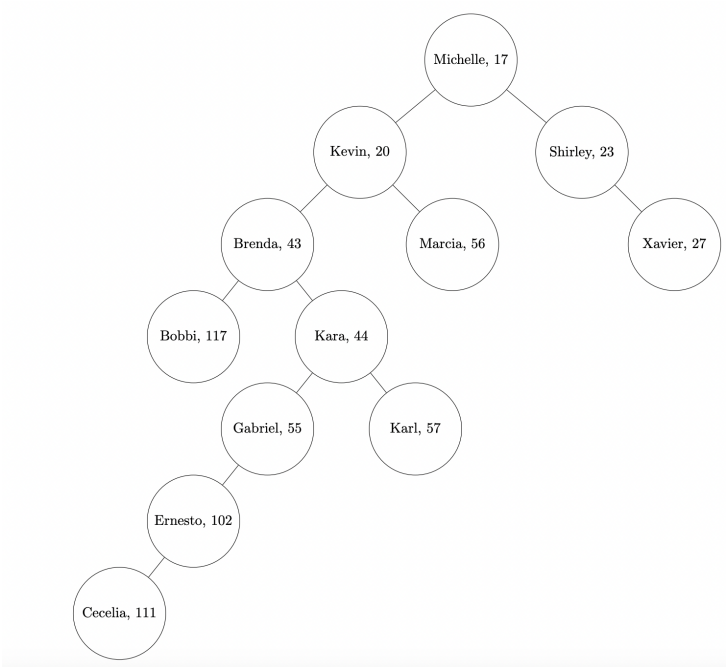


After this, the tree does not follow the rule in part a) ii. We continue to update the diagram until it follows the *QUBSET* rule.


Michelle, 17
Brenda, 43
Shirley, 23
Bobbi, 117
Kara, 44
Xavier, 27
Gabriel, 55
Marcia, 56
Ernesto, 102
Kevin, 20
Cecelia, 111
Karl, 57


Michelle, 17
Brenda, 43
Shirley, 23
Bobbi, 117
Kara, 44
Xavier, 27
Gabriel, 55
Kevin, 20
Ernesto, 102
Karl, 57
Marcia, 56
Cecelia, 111

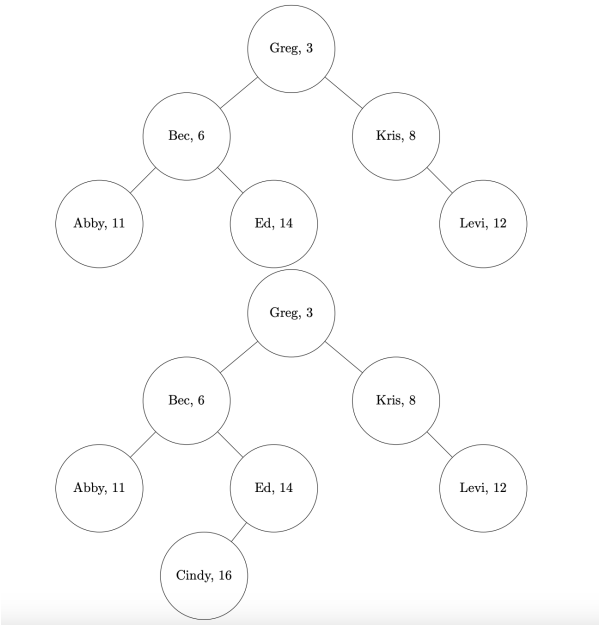

Michelle, 17
Brenda, 43
Shirley, 23
Bobbi, 117
Kevin, 20
Xavier, 27
Kara, 44
Marcia, 56
Gabriel, 55
Karl, 57
Ernesto, 102
Cecelia, 111



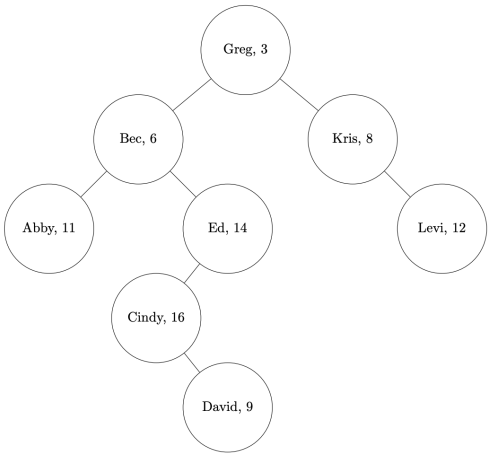
Now we satisfy the *QUBSET* rule in a) ii. and stop.

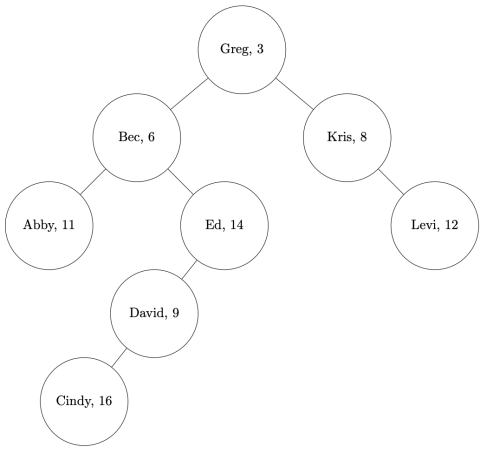
Example 2

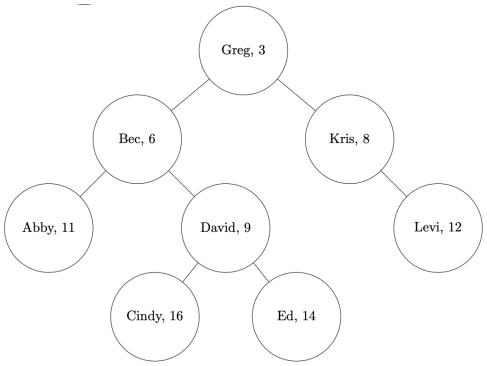
If we start with the following *QUBSET* and then add Cindy who has a student ID of 16, we do not need to change anything afterwards.



Now suppose we add David who has a student ID of 9 into our *QUBSET*. The following steps occur







And after this we are done.

Question 2: Colourful Study Notes

Tala is studying very hard for COMP20007. They read the corresponding book chapters every week to consolidate their understanding of the subject material.

To help with their studying. Tala likes to use highlighters of different colours. Here is an example, taken from the chapter on Brute Force methods:

> brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

In this example, Tala used **blue** to highlight the words "brute force", **yellow** to highlight the word "problem" and **green** to highlight "problem statement".

After doing this manually for years, Tala had the idea to write a program that automatically highlights terms. To do this, they used a program to read their past notes and check, for each word, how many times it was highlighted. This led to a collection of tables, one per word, with scores relative to how frequent that word was highlighted and which colour. For example, for the word "problem", the table looks like this:

colour	score
blue	0
yellow	10
green	8
no colour	5

In this example, **yellow** has a larger score because it is the most frequent colour the word "problem" appears in Tala's notes. Note there is also a row for **no colour**, meaning the word is not highlighted. After using the program to read their notes, Tala ends up with a set of tables like the one above, one per word.

Part A (code)

For the first automatic highlighter program, you should implement a C program that reads as input:
- A **sentence**. This is just a sequence of words split by whitespace, with no punctuation marks.
- A **set of word tables.** The set will only contain tables for words that appear in the sentence. You can assume there are only 4 colours and they are represented as **integers** from 0 to 3: 0 is "no colour", 1 is "green", 2 is "yellow" and 3 is "blue".

Then, it generates as the output:
- A **sequence of colours**, one per word. This should be a sequence of integer values, where each

integer represent a colour, as above.

The output sequence of colours should follow an **optimisation criterion:** it is the sequence with the **highest total score**. For Part A, the total score is the sum of individual scores per word. Formally speaking, assume a sentence has *n* words, with each word *w* numbered from *1* to *n*. Assume *F(n)* gives the maximum score for a sentence, which we define as

$$F(n) = \max_{C=c_1 \ldots c_n} \sum_{i=1}^{n} WC(w_i, c_i) = \sum_{i=1}^{n} \max_{c} WC(w_i, c)$$

where *WC(w,c)* corresponds to the score for colour *c* in word table *w.* The first term states that our goal is to the maximise the score given a **sequence** of words/colours The second term then simplifies this to maximise the score for each **individual** word/colour.

The equation above only gives the maximum score: your code should generate the sequence of colours that gives this maximum score.

Part B (code)

Following testing of the previous program, Tala is a bit frustrated because it would always pick the same colour for each term. But as show in the example above with the word "problem", sometimes the same word can be highlighted with different colours, depending on the sentence.

To solve this problem, Tala introduces an extra **colour transition** table. This table gives scores for colours given the **previous colour**. Here is an example:

previous colour	colour	score
blue	blue	10
yellow	blue	5
green	blue	3
no colour	blue	8
...	...	...
yellow	green	0
no colour	no colour	20

In this example table, if the previous word is highlighted in **blue** and the current word is also **blue** (as in the "brute force" example above), this **transition** gives a score of 10.

For the second automatic highlighter, you should enhance the code in Part A. The input is now:

- A **sentence**. As in Part A.
- A **set of word tables.** As in Part A.
- A **colour transition table**. Each entry in this table has two integers, corresponding to **previous colour** and **colour**, and a score. Integers represents colours, as in Part A.

The output is the same as Part A: a sequence of integers that gives the optimal highlighting. However, the criterion of highest total score now needs to sum the colour transition scores as well. Your program should follow a **greedy** approach: for every word from left to right, select the colour based on the maximum **sum** of two scores: the one from the word table and the one for the transition table.

Part C (written solution)

Tala is pleased with the result but notices the colouring could sometimes still be better. They realise the greedy algorithm in Part B is not actually generating the optimal sequence. To see this, we can write the formal equation for the maximiser with the colour transition table:

$$F(n) = \max_{C=c_1 \ldots c_n} WC(w_1, c_1) + \sum_{i=2}^{n} (WC(w_i, c_i) + CT(c_{i-1}, c_i))$$

CT(c1,c2) corresponds to the score in the colour transition where *c1* is the previous colour. The equation is similar to the one in Part A, but for the second to the last word, we take the sum of *WC* and *CT*. The main challenge here is the *CT* term inside the sum, which requires the colour for the previous word. This is fine if we are iterating over entire sequences of colours. However, the greedy algorithm picks one colour at a time from left to right, potentially missing the optimal sequence because it contains a colour that was optimal for an individual word.

Tala then decides to create a better algorithm. Their first idea is to use a brute force approach: generate all possible sequences of colours, calculate their scores and select the one with max score.

Assume you have *n* words and a total of *C* colours. State the complexity of the approach above in all cases.

Part D (written solution)

Tala quickly realises the brute force approach is too slow. They believe it is possible to use a Dynamic Programming solution instead. Here is Tala's reasoning:

- For the first word, there is no transition, so we can pick the best colour according to the word table.
- For the second word and afterwards, there are two options:
- Option 1: the best sequence includes the best colour for the previous word. In this case, we just need to pick the colour that maximises the sum of *WC(w,c)* and *CT(c1,c),* where *c1* is the colour of the previous word.

- Option 2: the best sequence does not include the best colour for the previous word. In this case, we need to check the sum of *WC(w,c)* and *CT(c1,c)* for **all other colours** that the previous word **could** have been highlighted with. Then we choose the colour that maximises this sum.

The greedy algorithm only stored the score for the best colour at each word. The above reasoning shows that we can get the best sequence if we store the scores for **all colours** at each word, that is, if we also take into account the total number of words.

With this in mind, write a recurrence relation for the score *F*.

Part E (code)

Write a C program that implements the dynamic programming approach in Part D to get the best **score** for a sequence of colours. Inputs are the same as Part B, Output should be a single number containing the best score. For this part, you can assume you only have 4 colours, as in Parts A and B.

Part F (code)

Modify the C program from Part E to get the best sequence of colours. Inputs and Outputs are the same as Part B. For this part, you can assume you only have 4 colours, as in Parts A and B.

Part G (written solution)

Assume you have *n* words and a total of *C* colours. State the complexity of the dynamic programming approach described in Part D in all cases.

Assignment Submission

Upload your solution here!

Add your answers to all non-programming parts for both tasks as a pdf called `written-tasks.pdf`. Submit your assignment using the "Mark" button. You may submit as many times as you like.

Note that Question 1 and Question 2 have written tasks which should be submitted as part of your PDF.

Academic Honesty

This is an individual assignment. The work must be your own work.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as doing this without proper attribution is considered plagiarism.

If you have borrowed ideas or taken inspiration from code and you are in doubt about whether it is plagiarism, provide a comment highlighting where you got that inspiration.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

“Borrowing” of someone else’s code without acknowledgment is plagiarism. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic integrity and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) in the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Late Policy and Protected Industrial Action

Usual Late Policy

The late penalty is 20% of the available marks for that project for each working day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that departmental servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, we are here to help! Frequently asked questions about the project will be answered on Ed.

The above is the usual late policy, read below for details on what the impact of protected industrial action will likely be on late penalties applied.

Statements about Protected Industrial Action

The enterprise agreement between staff and the university expired in 2021. As a part of negotiating for a better agreement which respects the contribution of staff to the functioning of the university and the importance of secure staff and a secure workforce to support them sustainably in delivering teaching (and other functions of the university), we began bargaining with the university. There were a litany of issues which for a number of years have been degrading staff experience and have regular flow on effects causing impacts on teaching, as well as student and staff wellbeing. From extensive staff consultation, we collectively put together a set of claims which provide a legally enforceable framework to ensure the issues which threaten to further impact teaching are corrected. Though management dragged their feet, they eventually agreed to meet with the staff bargaining team to discuss these claims - their response was to reject all key claims in almost their entirety around the end of last year and the start of this year.

As part of the stalemate or lack of urgency from university management, staff members of the union voted to give themselves the right to take protected industrial action. One of these actions which are currently in play is making statements about why members are taking protected action while working and another is a ban on applying late penalties.

You can find more information at unimelbebanow.com, but our claims include:

- Targets for Secure Work - Ensuring an enforceable target ensures that the university undertakes

meaningful work to ensure staff are moved to ongoing and secure positions.

- Improved Workloads - Ensuring staff have time to do all their work and that someone getting sick or some piece of work being misestimated doesn't mean weeks or months of delays.
- Restrictions on Restructures - Ensuring staff cannot be moved out of roles where the work they are responsible still needs to be done, again improving the quality of education that can be delivered where parts of the university aren't going to arbitrarily stop working on a whim and that most teams are able to stabilise and establish strong work patterns. This also includes some amendments such as staff being restructured at most once over the life of the agreement (2 or 3 years), which university management also did not agree to when proposed.

As well as a number of claims which would improve staff working conditions such as:

- Fair Pay - A modest pay rise in real terms at least meeting inflation is not a lot to ask for, but we would like at least that much as we have only had pay cuts in real terms since the pandemic began.
- Better Carer's/Parental Leave - Staff shouldn't have to put being a parent or carer on hold to work at the university, better leave entitlements ensure staff can take time off when it is needed.
- Better Flexible and Work From Home Arrangements - Not all staff at the university teach or stand in classes in front of you, a lot of work could be performed either at the university or at home, enshrining this requirement in the agreement ensures staff can work more flexibly where they don't have any real requirement to be physically at the university.

We are also fighting (defensively) to ensure clauses protecting against unpaid work are not replaced in the new agreement so that staff retain the right to be paid for all work performed.

Because University management have not meaningfully engaged with the bargaining team to work towards these claims being implemented, staff at the university are taking protected action.

The reason why protected action has been taken while writing this slide is that the second protected action mentioned is a ban on applying late penalties which is currently in effect.

Ban on Applying Late Penalties

The partial work ban of not applying late penalties is currently in effect, this means for union staff taking part in the partial work ban, no late penalties will be applied. What this means for you is that though the due date of 26th of May still applies, this simply means submitting past this date will incur no late penalty.

You should still endeavour to submit as close to this date as you can, as we will do the best we can to provide you feedback prior to the exam and marking involves work. But submitting past this date will not incur a penalty.

As with Assignment 1, we may release a sample solution prior to the exam to allow you to check your answers along with the official release of feedback. If you are submitting after this is released, ensure you do not check it until after you have submitted as this may lead to a deduction corresponding to the parts which were not

your own work and may constitute academic misconduct if used without attribution.

If you have reason to submit late (e.g. you are likely to need additional time due to circumstances typically compatible with special consideration), you might still like to let us know and we can ensure that your assignment is marked ahead of the exam if possible.

If you submit after feedback has been released, shoot the head tutor an email (grady.fitzpatrick@unimelb.edu.au) to ensure your assignment is marked.

When Protected Industrial Action Ends

When university management accepts our claims and we accept an offer endorsed by the union membership, protected industrial action will cease. This means the ban on late penalties will likely end - in the interest of fairness, we will notify you when this occurs and give a timeline on when late penalties will begin to apply again to avoid any potential disadvantage - work will not receive late penalties until this point from NTEU staff members.

Requirements: C Programming

The following implementation requirements must be adhered to:

- You must write your implementation in the C programming language.
- Your code should be easily extensible to multiple data structure instances. This means that the functions for interacting with your data structures should take as arguments not only the values required to perform the operation required, but also a pointer to a particular data structure, e.g. `search(dictionary, value)`.
- Your implementation must read the input file once only.
- Your program should store strings in a space-efficient manner. If you are using `malloc()` to create the space for a string, remember to allow space for the final end of string character, '`\0`' (`NULL`).
- Your approach should be reasonably *time efficient*.
- Your solution should begin from the provided scaffold.

Hints:

• If you haven't used `make` before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces.

• It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

Programming Style

Below is a style guide which assignments are evaluated against. For this subject, the 80 character limit is a guideline rather than a rule — if your code exceeds this limit, you should consider whether your code would be more readable if you instead rearranged it.

```
/** ***********************
* C Programming Style for Engineering Computation
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
* Definitions and includes
* Definitions are in UPPER_CASE
* Includes go before definitions
* Space between includes, definitions and the main function.
* Use definitions for any constants in your program, do not just write them
* in.
*
* Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
* Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
* style. Both are very standard.
*/

/**
* GOOD:
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
 ...

/**
* BAD:
*/

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
 ...

/** ****************************
* Variables
* Give them useful lower_case names or camelCase. Either is fine,
```

```
* as long as you are consistent and apply always the same style.
* Initialise them to something that makes sense.
*/

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

  int i = 0;
  int num_fifties = 0;
  int num_twenties = 0;
  int num_tens = 0;

 ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

  int i = 0;
  int numFifties = 0;
  int numTwenties = 0;
  int numTens = 0;

 ...
/**
* BAD:
*/

int main(int argc, char **argv) {

  /* Variable not initialised - causes a bug because we didn't remember to
  * set it before the loop */
  int i;
  /* Variable in all caps - we'll get confused between this and constants
  */
  int NUM_FIFTIES = 0;
  /* Overly abbreviated variable names make things hard. */
  int nt = 0

  while (i < 10) {
    ...
    i++;
  }

 ...

/** ********************
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
```

```
* One tab of indentation within either a function or a loop.
* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

 int i = 0;

 for(i = 100; i >= 0; i--) {
   if (i > 0) {
     printf("%d bottles of beer, take one down and pass it around,"
     " %d bottles of beer.\n", i, i - 1);
   } else {
     printf("%d bottles of beer, take one down and pass it around."
     " We're empty.\n", i);
   }
 }

 return 0;
}

/**
* BAD:
*/

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
 int i = 0;
 /* No space between variable declarations and the rest of the function.
 * No spaces around the boolean operators */
 for(i=100;i>=0;i--) {
 /* No indentation */
 if (i > 0) {
 /* Line too long */
 printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
 } else {
 /* Spacing for no good reason. */

 printf("%d bottles of beer, take one down and pass it around."
 " We're empty.\n", i);
```

```
 }
 }
 /* Closing brace not on its own line */
 return 0;}

/** ****************
* Braces:
* Opening braces go on the same line as the loop or function name
* Closing braces go on their own line
* Closing braces go at the same indentation level as the thing they are
* closing
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

  ...

  for(...) {
    ...
  }

  return 0;
}

/**
* BAD:
*/

int main(int argc, char **argv) {

  ...

  /* Opening brace on a different line to the for loop open */
  for(...)
  {
    ...
    /* Closing brace at a different indentation to the thing it's
    closing
    */
           }

 /* Closing brace not on its own line. */
       return 0;}

/** **************
* Commenting:
* Each program should have a comment explaining what it does and who created
* it.
* Also comment how to run the program, including optional command line
```

```
* parameters.
* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
* GOOD:
*/

/* change.c
*
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
13/03/2011
*
* Print the number of each coin that would be needed to make up some
change
* that is input by the user
*
* To run the program type:
* ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
*
* To see all the input parameters, type:
* ./coins --help
* Options::
*   --help                Show help message
*   --num_coins arg       Input number of coins
*   --shape_coins arg     Input coins shape
*   --bound arg (=1)      Max bound on xxx, default value 1
*   --output arg          Output solution file
*
*/

int main(int argc, char **argv) {

  int input_change = 0;

  printf("Please input the value of the change (0-99 cents
  inclusive):\n");
  scanf("%d", &input_change);
  printf("\n");

  // Valid change values are 0-99 inclusive.
  if(input_change < 0 || input_change > 99) {
    printf("Input not in the range 0-99.\n")
  }

  ...

/**
* BAD:
*/

/* No explanation of what the program is doing */
int main(int argc, char **argv) {
```

```
  /* Commenting obvious things */
  /* Create a int variable called input_change to store the input from
  the
  * user. */
  int input_change;

  ...

/** ****************
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
  printf("Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
  /* lots of computation here, pushing the else part off the screen.
  */
  ...
} else {
  fprintf(stderr, "Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}
```

Some automatic evaluations of your code style may be performed where they are reliable. As determining whether these style-related issues are occurring sometimes involves non-trivial (and sometimes even undecidable) calculations, a simpler and more error-prone (but highly successful) solution is used. You may need to add a comment to identify these cases, so check any failing test outputs for instructions on how to resolve incorrectly flagged issues.

Mark Breakdown

There are a total of 20 marks given for this assignment.

Your C programs for Task 2 should be accurate, readable, and observe good C programming structure, safety and style, including documentation. Safety refers to checking whether opening a file returns something, whether mallocs do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting, by choosing descriptive variable names. The remainder of the marks will be based on the correct functioning of your submission.

Note that marks related to the correctness of your code will be based on passing various tests. If your program passes these tests without addressing the learning outcomes (e.g. if you fully hard-code solutions or otherwise deliberately exploit the test cases), you may receive less marks than is suggested but your code marks will otherwise be determined by test cases.

Mark Breakdown

Task	$\mathrm{Mark\ Distribution}$
$\mathrm{Question\ 1}$	$\mathrm{1\ Mark\ (Part\ A) + 2\ Marks\ (Part\ B) + 3\ Marks\ (Part\ C)}$ $\mathrm{+ 1\ Mark\ (Part\ D) + 1\ Mark\ (Part\ E)}$
$\mathrm{Question\ 2}$	$\mathrm{2\ Mark\ (Part\ A) + 1\ Mark\ (Part\ B) + 2\ Mark\ (Part\ C)}$ $\mathrm{+ 2\ Mark\ (Part\ D) + 2\ Mark\ (Part\ E) + 2\ Mark\ (Part\ F) + 1\ Mark\ (Part}$

Note that not all marks will represent the same amount of work, you may find some marks are easier to obtain than others. Note also that the test cases are similarly not always sorted in order of difficulty, some may be easier to pass than others.

Additional Support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Ed discussion forum, using the folder tag Assignments for new posts. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the discussion forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking.

If you have questions about your code specifically which you feel would reveal too much of the assignment, feel free to post a private question on the discussion forum.