COMP 3711 – Design and Analysis of Algorithms
2021 Fall Semester – Programming Assignment # 1

Distributed: October 19, 2021

Due: November 1, 2021, 11:59 PM

Your solution should contain

(i) your name, (ii) your student ID #, and (iii) your email address

at the top of its first page.

Some Notes:

- Please write clearly and briefly. Your solution to Problem 1 should follow the guidelines given at
  https://canvas.ust.hk/courses/38226/pages/assignment-submission-guidelines

  In particular, your solutions should be written or printed on *clean* white paper with no watermarks, i.e., student society paper is not allowed.

- Please also follow the guidelines on doing your own work and avoiding plagiarism as described on the class home page.
  You must acknowledge individuals who assisted you, or sources where you found solutions. Failure to do so will be considered plagiarism.

- Please make a *copy* of your assignment before submitting it. If we can't find your submission, we will ask you to resubmit the copy.

- Submission: There will be two submissions for this assignment. **Both of them will be submitted to CASS and not Canvas.** Please see the assignment page for more details on how to submit to CASS.
  - A softcopy of your solution to Problem 1. As in the homeworks, this should be one PDF file (no word or jpegs permitted, nor multiple files). If your submission is a scan of a handwritten solution, make sure that it is of high enough resolution to be easily read. At least 300dpi and possibly denser.
  - Your code for Problem 2. This should be one file chomp.cpp

This assignment is to design a dynamic programming recurrence for the game of **Chomp** and then write code to solve the $5 \times n$ version of the game. The description of Chomp below is taken from *Anna R. Karlin and Yuval Peres. Game theory, Alive. American Mathematical Society Press , 2017.* Figure (1) illustrates the rules and concepts described below.

- **Chomp** is played on a bar of chocolate divided into $m \times n$ squares. The bottom left corner of the bar has been removed and replaced with a poisoned square.
- Two players take turns biting off a chunk of the bar. More specifically, each player, in his turn, chooses an uneaten chocolate square and removes it along with all of the squares that lie above and to the right of it.
- The person who bites the last piece of chocolate wins and the loser has to eat the poisoned square.

Note. There is a lot of information about Chomp available on the internet. Some sources, such as the Wikipedia page on the game, present a slightly different version that has the upper left square poisoned. Don't get confused by this.

Legal Positions:

We assign consecutive indices to the rows of the original $m \times n$ chocolate bar, starting with 1 for the bottom row and ending with m for the top row.

A *position* in an $m \times n$ game can be uniquely described by a m-tuple $\mathbf{x} = (x_1, \ldots, x_m)$; x_i is the number of squares of chocolate remaining on row i.

Note that the game requires that $n \geq x_1 \geq x_2 \cdots \geq x_m \geq 0$ and $x_1 \geq 1$. We call such an $\mathbf{x}$ a *legal position.* An $\mathbf{x}$ that does not satisfy this requirement is not a legal position.

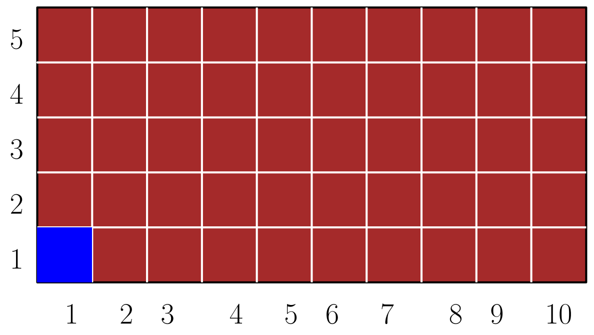
$\mathcal{X}$ denotes the set of legal positions.

Legal Moves:
We say that $\mathbf{x} \to \mathbf{x}'$ is a *legal move* if $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$, i.e., are both legal positions, and it is possible to move from $\mathbf{x}$ to $\mathbf{x}'$ in one move.

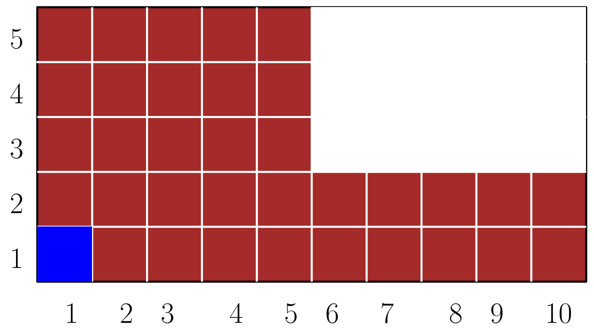
Example 1: $(3,3,2,1) \to (3,1,1,1)$ and $(3,3,2,1) \to (3,2,2,1)$ are both legal moves but $(3,3,2,1) \to (3,2,1,1)$ is not a legal move.

Example 2: Figure 1 shows that $(10,10,10,10,10) \to (10,10,5,5,5)$ and
$(10,10,5,5,5) \to (10,10,5,2,2)$.

$\texttt{Next}(\mathbf{x}) = \{\mathbf{x}' \in \mathcal{X} : \mathbf{x} \to \mathbf{x}'\}$ is the set of legal positions that can be reached by one legal move from $\mathbf{x}$.

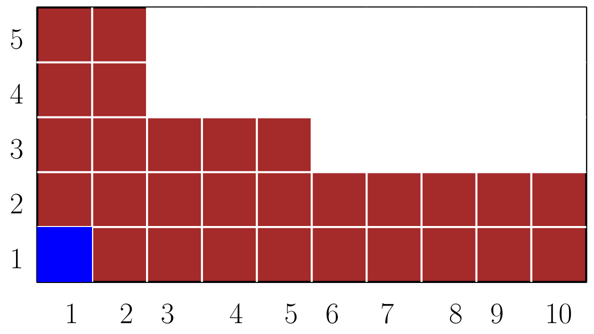


$$\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$$
$$= (10, 10, 10, 10, 10)$$



After chomping square $(3, 6)$

$\mathbf{x} = (10, 10, 5, 5, 5)$



After further chomping square $(4, 3)$

$\mathbf{x} = (10, 10, 5, 2, 2)$

After further chomping square $(3, 4)$, $\mathbf{x} = (10, 10, 3, 2, 2)$.

Figure 1: A 5×10 Chomp board. the blue square is the poisoned one.

Winning and Losing Positions: We say that a position $\mathbf{x}$ is *winning* if the first player that plays starting from position $\mathbf{x}$ has a strategy that guarantees that they win. If $\mathbf{x}$ is not a winning position, it is called a *losing position.*

Note: In what follows "square (i, j)" denotes the square at row i and column j.

Example 3: $(1, 0, 0)$ is a losing position because the current player has only one move, to eat the poisoned square.

$(r, 0, 0)$ is always a winning position for $r > 1$ because the current player can always bite the square $(1, 2)$. This puts the game in position $(1, 0, 0)$ so the next player must eat the poisoned square and lose.

Similarly $(1, 1, 1)$ and $(1, 1, 0)$ are both winning positions because the current player can bite the square $(2, 1)$, also leaving the next player in position $(1, 0, 0)$.

Example 4: $\mathbf{x} = (2, 1, 0)$ is a losing position: The first player has only two possible moves available. Either (i) bite the square at $(1, 2)$ and leave the second player in position $(1, 1, 0)$ or (ii) bite the square at $(2, 1)$ and leave the second player in position $(2, 0, 0)$. No matter which the first player bites, the second player is in a winning position and therefore wins.

Example 5: $(r, t, 0)$ with $t \neq r-1$ is always winning position and $\mathbf{x} = (r, t, 0)$ with $t = r - 1$ is always a losing position. (Can you see why? Hint, use induction.)

The general observation (can you prove why?) is that

(i) $\mathbf{x}$ is a winning position if and only if there exists a losing position $\mathbf{x}'$ such that $\mathbf{x} \to \mathbf{x}'$;

(ii) $\mathbf{x}$ is a losing position if and only if for all $\mathbf{x}'$ such that $\mathbf{x} \to \mathbf{x}'$, $\mathbf{x}'$ is a winning position.

Although CHOMP has been extensively studied, no one knows a general winning strategy.

As an example, consider the starting position $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ with $x_i = n$ for all i, for an $m \times n$ game.

It is actually known that the first player ALWAYS has a winning strategy for such a position, i.e., that the first player can always win.

But, no one knows WHAT the general winning strategy for that $\mathbf{x}$ is.

The goal of this problem assignment is to derive a dynamic program for solving fixed-size CHOMP and then implement it in code. This will involve filling in a boolean *game table.*

In what follows **T** denotes TRUE and **F** denotes FALSE.

Let m, n be given. $\mathbf{x} = (x_1, \ldots, x_m)$ denotes a legal position. $\mathbf{WIN}(\mathbf{x})$ is a *boolean* table indexed by legal game positions satisfying

$$\mathbf{WIN}(\mathbf{x}) = \begin{cases} \mathbf{T} & \text{if } \mathbf{x} \text{ is a winning position;} \\ \mathbf{F} & \text{if } \mathbf{x} \text{ is a losing position.} \end{cases}$$

Example 6: From Example 5, if (x_1, x_2) is a legal position,

$$\mathbf{WIN}((x_1, x_2)) = \begin{cases} \mathbf{T} & \text{if } x_2 \neq x_1 - 1; \\ \mathbf{F} & \text{if } x_2 = x_1 - 1. \end{cases}$$

Example 7: $\mathbf{WIN}((2,1,0)) = \mathbf{F}$; $\mathbf{WIN}((5,4)) = \mathbf{F}$; $\mathbf{WIN}((5,5)) = \mathbf{T}$;

$\mathbf{WIN}(\mathbf{x})$ is a boolean table, so you will need the following logic notation to discuss how to fill it in.

Logic Notation: In what follows $A_1, A_2, \ldots, A_n$ are boolean variables that take logical values $\mathbf{T}$ or $\mathbf{F}$:

- $A_1 \wedge A_2 = \mathbf{T}$ if and only if both $(A_1 = \mathbf{T})$ and $(A_2 = \mathbf{T})$.
- $\bigwedge_{i=1}^{n} A_i$ is equivalent to $A_1 \wedge A_2 \wedge \cdots \wedge A_n$.
- $A_1 \vee A_2 = \mathbf{T}$ if and only if at least one of $(A_1 = \mathbf{T})$ and $(A_2 = \mathbf{T})$ occurs.
- $\bigvee_{i=1}^{n} A_i$ is equivalent to $A_1 \vee A_2 \vee \cdots \vee A_n$.
- $\neg(A_1) = \mathbf{T}$ if and only if $A_1 = \mathbf{F}$.

These symbols permit you to write equations using the $\mathbf{WIN}$ table.

As examples:

If $\mathbf{x}$, $\mathbf{x}'$ are legal positions then $\Big(\mathbf{WIN}(\mathbf{x}) \wedge \mathbf{WIN}(\mathbf{x}')\Big) = \mathbf{T}$ if and only if both $\mathbf{WIN}(\mathbf{x}) = \mathbf{T}$ and $\mathbf{WIN}(\mathbf{x}') = \mathbf{T}$.

Similarly, if S is a set of legal positions, then $\Big(\bigwedge_{\mathbf{x} \in S} \mathbf{WIN}(\mathbf{x})\Big) = \mathbf{T}$ if and only if $\mathbf{WIN}(\mathbf{x}) = \mathbf{T}$ for every $\mathbf{x} \in S$.

Problem 1: Design a dynamic programming recurrence to correctly fill in the $\mathbf{WIN}(\mathbf{x})$ table.

(a) Write the DP recurrence for filling the table $\mathbf{WIN}(\mathbf{x})$ using the format below:

$$\mathbf{WIN}(\mathbf{x}) = \begin{cases} ????? & \text{if } ????? \\ ????? & \text{if } ????? \\ \cdots & \cdots \\ ????? & \text{if } ????? \end{cases}$$

The format above is generic and is not meant to imply that the recurrence has a lot of cases. It might only have two or might have as many as ten. That is up to you to discover. Also, your recurrence should include the initial condition(s).

Important. Your recurrence for (a) should be written using the logic notation that was introduced previously, e.g., $\vee$ and $\wedge$ and $\neg$ as described on the previous page.

Your recurrence should NOT use C++ semantics, e.g., ||, && and !. **Recurrences written using the C++ notation will be marked as wrong.**

(b) Prove the correctness of your recurrence from part (a)

Problem 2: Write C++ code to implement your recurrence from Problem 1 to fill in the **WIN**($\mathbf{x}$) table for an $m \times n$ game board.

Note that the purpose of this assignment is not to test whether you can code. It is to see whether you can write a DP recurrence that can be easily translated into table-filling code. As in most DPs, the real work is in getting the correct recurrence. We do not expect the actual code to be very long.

Your code must fit into our testing framework. More specifically, we are providing a file "chomp.cpp".

You need to fill in the contents of the procedure

```
void create(int n)
{ \\ Fill in details here
}
```

`create(n)` should properly fill in the **WIN**($\mathbf{x}$) table for a $5 \times n$ Chomp game. That is it should set **WIN**($\mathbf{x}$) to be either **T** or **F** appropriately for all legal positions $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$ where $x_1 \le n$.

We have already created the **WIN**($\mathbf{x}$) array.

You may NOT access the array directly. You may only access it via the following three functions:

(a) `bool setTrue(int a, int b, int c, int d, int e):`
This procedure sets **WIN**$((a, b, c, d, e)) = \mathbf{T}$.
If (a, b, c, d, e) is not a legal position, the procedure reports an error and stops the program.

(b) `bool setFalse(int a, int b, int c, int d, int e):`
This procedure sets **WIN**$((a, b, c, d, e)) = \mathbf{F}$.
If (a, b, c, d, e) is not a legal position, the procedure reports an error and stops the program.

(c) `bool report(int a, int b, int c, int d, int e)`
This procedure reports the value (**T** or **F**) stored in **WIN**$((a, b, c, d, e))$.
If (a, b, c, d, e) is not a legal position, the procedure reports an error and stops the program.
If **WIN**$((a, b, c, d, e))$ has not been set by you using one of the two preceding commands, the outcome is undefined.

Note. Formally, $\mathbf{x_0} = (0, 0, 0, 0, 0)$ is not a legal position. For technical reasons, the code will permit you to set/report the value **WIN**($\mathbf{x_0}$). This might (or might not) be useful to your design. In any case, the value of **WIN**($\mathbf{x_0}$) is technically undefined, so we will not check it.

Submission Rules

(a) We provide two files **chomp.cpp**, and **main.cpp**.

(b) **chomp.cpp** contains an empty function `create(n)` that needs to be written.

(c) Your submission will be the file **chomp.cpp** containing only the code for `create(n)` and whatever other helper functions you want to write to use with it. Nothing else.

(d) **chomp.cpp** will be called by **main.cpp**.

Our marking script will check the operation of your `create(n)` function for some value of n, $10 \leq n \leq 20$.

That is, we will check if it produces the correct values for $\textbf{WIN}(\mathbf{x})$ for all legal positions $\mathbf{x}$ with $x_1 \leq n$.

Note that the **main.cpp** provided runs `create(n)` with $n = 10$. If you change the value of n for testing purposes note that we have limited $n \leq N = 12$ when setting up the $\textbf{WIN}(\mathbf{x})$ table. This is just to be on the safe side of your personal machine memory space. To test for larger problem sizes n you may change the value of N in **main.cpp**.

(e) Our testing environment for compiling and running the program is in Linux. You can use the CSE lab 2 machines (use ssh for remote access) to access it.

(f) In that environment you can run the code using
g++ main.cpp -std=c++11.

(g) Before submitting, you should make sure that your program generates no compilation error (when compiled using the command from (f)) and no runtime error (in the lab 2 machine environment).

(h) Your code will be marked for clear documentation and properly indentation. The purpose of the documentation/indentation is to show how your code matches the DP recurrence.

By looking at the documentation, we should be able to immediately tell what part of your DP recurrence is being implemented and how.

(i) You can only access the $\textbf{WIN}(\mathbf{x})$ table via the three functions described on the previous page. They should not permit illegal positions to be accessed. Your code will fail if it tries to do that.

(j) This will be a binary check, Either your code works, or it doesn't. There will be no middle ground where it "almost works".

(k) We also provide a second **file main2.cpp** that you can run to check if your table entries with $x_3 = x_4 = x_5 = 0$ are correct. It uses a function `check2(n)`. Our real marking procedure will replace `check2(n)` with another function `check(n)`.