

CS 577 Summer 2022

Homework 1

~Submission~

- **Groups of up to six people can submit joint solutions.** Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.
 - **Submit your solutions electronically on the course Gradescope site as PDF files.** Use the Latex template provided and place your answers in the solution blocks. You may include hand-drawn figures as needed by using `\includegraphics` with the desired image.
 - **Make sure to specify on Gradescope which pages of the PDF go to which problems.**
-

~Homework Policies~

- **You may use any source at your disposal**—paper, electronic, or human—but you **must** cite **every** source that you use, and you must write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
 - The answer “**I don’t know**” (and nothing else) is worth 25% partial credit on any required problem or subproblem, on any homework or exam. We will accept synonyms like “No idea” or “WTF” or “what??”, but you must write something.
 - **Avoid the Three Deadly Sins!** Any homework or exam solution that breaks any of the following rules will be given an **automatic zero**, unless the solution is otherwise perfect. Yes, we really mean it. We’re not trying to be scary or petty (Honest!), but we do want to break a few common bad habits that seriously impede mastery of the course material.
 - Always give complete solutions, not just examples.
 - Always declare all your variables, in English. In particular, always describe the specific problem your algorithm is supposed to solve.
 - Never use weak induction.
 - **Unless otherwise specified, when asked to describe and analyze an algorithm, you must**
 - **Specify** the problem the algorithm solves if different than exactly what the question asks. For example, if the algorithm solves a more general problem.
 - **Describe** the algorithm either in English or Pseudocode depending on which most clearly and precisely conveys the ideas.
 - Give an argument for **correctness**. This sometimes requires a brief induction proof.
 - Give an asymptotic **analysis** of the algorithm’s run time.
-

Problem. *Textbook problem Ch0 problem 0*

Describe and analyze an efficient algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. [Hint: There is a trivial one-line solution!]

Solution:

An algorithm is following:

state - the current arrangement of chess board:

```
FINDWINNER(state):  
  if state == end  
    if winner_A = TRUE  
      return winner_A  
    else if winner_B = TRUE  
      return winner_B  
    else  
      return DRAW  
  state = CHOOSE_NEXT_STEP(state)  
  return FINDWINNER(state)
```

```
CHOOSE_NEXT_STEP(state):  
  if player_A = TRUE  
    state = PLAYER_A_CHOOSE_A_STEP()  
  else  
    state = PLAYER_B_CHOOSE_A_STEP()  
  return emphstate
```

analysis:

correctness: To prove the correctness of FindWinner, we use induction. When a game reaches end, there must be a winner in two player or reach draw. And if not end, the players would take a correct step, and then the arrangement of chess board will change. By the principle of the chess, it would finally reach an end or draw situation, so FindWinner is correct.

To prove the correctness of Choose_Next_Step, at the beginning, we assume that each player would pick the legal and perfect step in their turn, so Choose_Next_Step is also correct.

run time: We can use recursion tree to evaluate the run time. Each state on the chess board could be considered as a node in a recursion tree and beginning state is the root. Their have children, which are the perfect moves chosen by the players from any possible next steps. But the size of chess board is fixed and the total arrangement of the chess is finite, the run time complexity is $O(1)$ ■

Problem. Textbook problem Ch1 problem 29 parts (a) - (c)

Most graphics hardware includes support for a low-level operation called blit, or block transfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixel map 90 clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. (Why five? For the same reason the Tower of Hanoi puzzle needs a third peg.) Alternately, we could first recursively rotate the blocks and then blit them into place.

- (a) Prove that both versions of the algorithm are correct when n is a power of 2.
- (b) Exactly how many blits does the algorithm perform when n is a power of 2?
- (c) Describe how to modify the algorithm so that it works for arbitrary n , not just powers of 2. How many blits does your modified algorithm? perform?

Solution:

- (a) To prove first blit then recurse:

We use induction to prove its correctness. For the start case, $n = 1$, when there is only one pixel, which is also zero power of 2. So there is no need to do blit operation, just rotate the pixel and make it into position.

We have an induction hypothesis that this algorithm works for any $n/2 \times n/2$ pixels square (n is power of 2 and n is greater than 2), and we want to prove that it also works for $n \times n$ pixels square. An n pixels square (n is power of 2), which could also be even split into 4 small blocks, which have $n/2 \times n/2$ pixels square block each each. Since $n/2 \times n/2$ pixels square has been successfully rotated, by our assumption. We could do another blit operation, each block will move clockwise, so any pixel in $n/2 \times n/2$ blocks will move clock-wisely into the correct position in $n \times n$ pixels blocks. Thus, the algorithm that first blit then recurse works for $n \times n$ pixels block, and is correct.

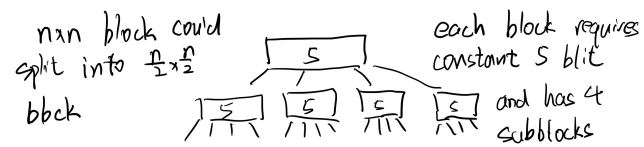
To prove first recurse then blit:

The basic case is same as previous one, just rotate and then complete, we also have the same induction hypothesis, and we could also split $n \times n$ square into four even $n/2 \times n/2$ and move clock-wisely. With the same proving as above, we could prove that first recurse then blit algorithm is also correct

- (b) when $n = 2^0$, there is just one single pixel, so it will be rotated once, we just use 0 blit; When $n = 2^1$, we just split it into 4 piece of $n/2 \times n/2$ blocks and we need 5 blits to make each block move into the correct position, so $N(\text{the number of blit}) = 5^1$; When $n = 2^2$, after the first time we split the whole pixel map in to 4 blocks and make 5 blits, we just keep splitting these 4 blocks into 16 smaller block and use 5 blits to each 4 of them, in this way, $N = 5^2$

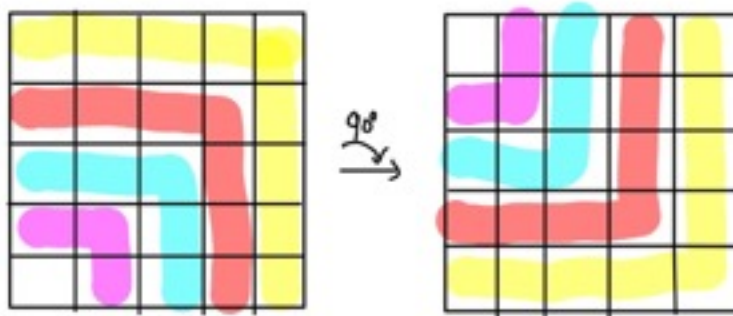
In conclusion, when $n = 2^x$ ($x = 1, 2, \dots, n$), N (the number of blit) $= 5^n$, but when $x = 0$, N will be 0.

to calculate its run time complexity, we draw a recursion tree



We get the equation $T_n = 4T_{n/2} + 5$
 we could draw the recursion tree, its height $h = \log_2 n$
 $T_n = O\left(\sum_{l=0}^{\log_2 n} 5 \cdot 4^l\right) = O(4^{\log_2 n}) = O(n^2)$

(c)



■

Problem. Textbook problem Ch1 problem 37

For this problem, a subtree of a binary tree means any connected subgraph. A binary tree is complete if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the largest complete subtree of a given binary tree. Your algorithm should return both the root and the depth of this subtree. See Figure 1.26 for an example.

Solution:

Sudo Code:

```

COMPLETETREEDEPTH(root)
  If root.left == null or root.right == null:
    Return 1
  Else
    Return Min(COMPLETETREEDEPTH(root.left), COMPLETETREEDEPTH(root.right)+1)

```

```

COMPLETETREE(root)
  If root == null
    Return null and -1
  Else
    Return MaxDepth((root and COMPLETETREEDEPTH(root)), COMPLETETREE(root.left), COMPLETETREE(root.right))
    (MaxDepth will pick the largest depth from these three and return it's root and depth)

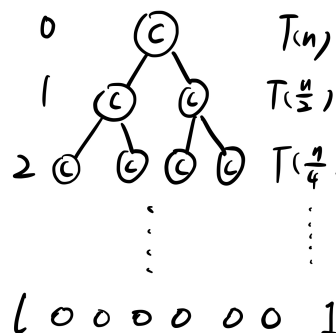
```

Correctness:

We use two parts of induction to prove the correctness

Time Complexity:

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + C \\
 &= 2T\left(\frac{n}{2}\right) + C \\
 T\left(\frac{n}{2}\right) &= 2T\left(\frac{n}{4}\right) + C
 \end{aligned}$$



$$\sum_{i=1}^n i = 2^l]$$

