Qinfei Liqli447 Yujiao Zhuyzhu427

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.

CS 577 Summer 2022 — Haoji Bian (hbian8)
Homework 1 Problem 1 — Xinjie Chen (xchen943)
Wenbo Jin (wjin44)

Problem. *Textbook problem Ch*0 *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) ■

CS 577 Summer 2022 — Haoji Bian (hbian8)
Homework 1 Problem 2 — Xinjie Chen (xchen943)
Wenbo Jin (wjin44)

Problem. *Textbook problem Ch*1 *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 x 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 x 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 on need to do blit operation, just rotate the pixel and make it into position.

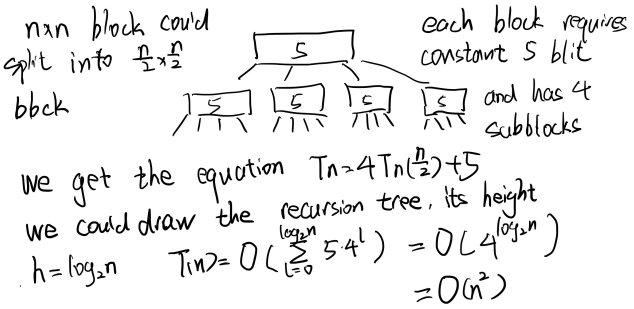
We have an induction hypothesis that this algorithm works for any n/2 x n/2 pixels square (n is power of 2 and n is greater than 2), and we want to prove that it also works for n x n pixels square. An n pixels square (n is power of 2), which could also be even split into 4 small blocks, which have n/2 x n/2 pixels square block each each. Since n/2 x 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 x n/2 blocks will move clock-wisely into the correct position in n x n pixels blocks. Thus, the algorithm that first blit then recurse works for n x n pixels block, and is correct.

To prove first recurse then blit:

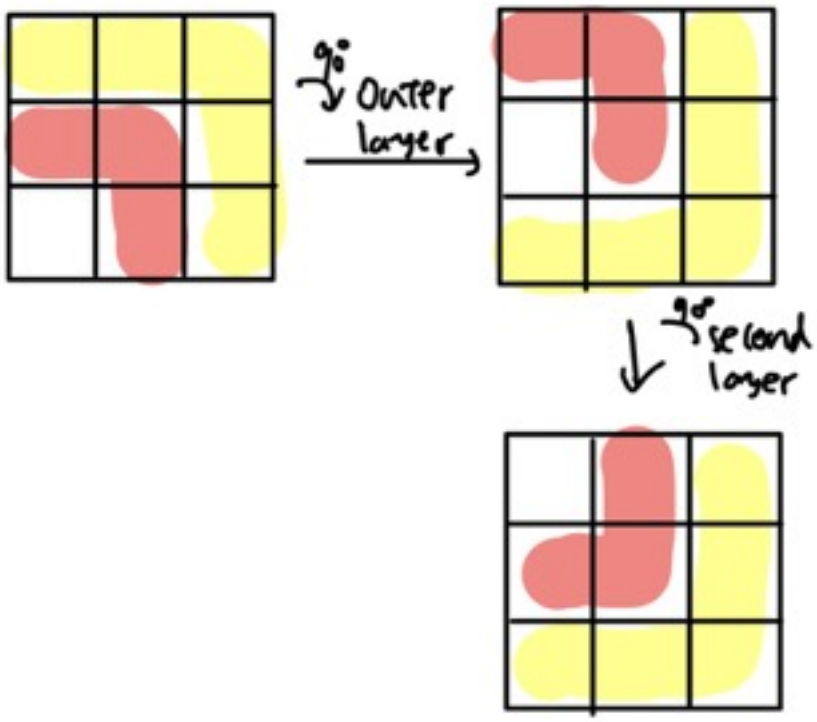
The basic case is same as previous one, just rotate and then complete, we also have the same induction hypothesis, and we could also split n x n square into four even n/2 x n/2 and move clock-wisely. With the same proving as above, we could prove that first recurse then blit algorithm is also correct

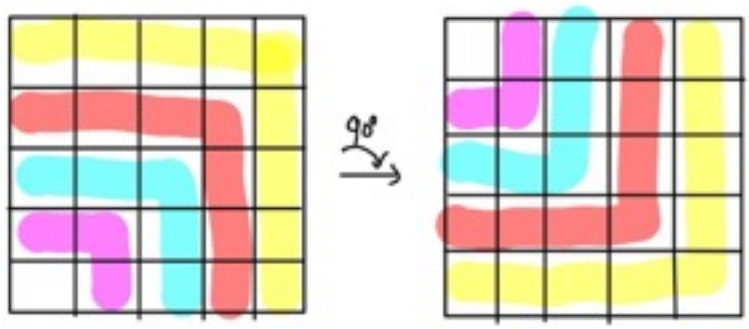
(b) when n = 2^0, there is just one single pixel, so it will be rotated once, we just use 0 blit; When n =2^1 , we just split it into 4 piece of n/2x n/2 blocks and we need 5 bilits to make each block move into the correct position, so N(the number of blit) = 5^1;When n = 2^2 , after the first time we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

In conclusion, when n $=2^x$ (x= 1,2,.....,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



(c)





First we split a whole pixel map into an L and a square. We first blit and rotate the split L. The square stays the same. Similarly, we perform a recursion operation on the remaining square of the previous partition, and then partition it into L and a smaller square center. When there is only one block in the direct base case, we blit and rotate the last small block.

The number of blits of a pixel map with edge length n is the square of n. First, we blit the whole pixel map to the right, and then we fill the blanks before the previous blit one at a time with the small blocks after the big blit. Until all blocks are moved to the corresponding positions. Except for the smallest block at the end of this algorithm, all the other blocks are blit once, and together with the previous block of blits, we can conclude that the number of blits is the square of n.

■

CS 577 Summer 2022 — Haoji Bian (hbian8)
Homework 1 Problem 3 — Xinjie Chen (xchen943)
Wenbo Jin (wjin44)

Problem. *Textbook problem Ch*1 *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:

1. To prove the correctness of CompleteTree(root): we firstly test the basic case when the input root is null, we return null, and -1. It is correct, since there is no node in tree and no depth.

We have the induction hypothesis that for any node, the function CompleteTree() returns the correct root and the depth of the largest complete tree of its left and right children. And we assume that CompleteTreeDepth(root) return the correct depth of complete tree starting from current root. We have to prove that CompleteTree(root) also return the correct root and the depth of the largest complete tree of current node.

The method MaxDepth() has three sets of two inputs: root and depth, would find which set got the largest depth and return its root and depth. If the depth of complete tree of current root is larger than its children, it would choose the current root as new root to return itself and also the depth of the completed tree. Otherwise, if smaller than its children, it will still keep the child which gets larger depth in two of the children as root to and return complete tree depth. So method CompleteTree() is correct.

2. To prove the correctness of CompleteTreeDepth(root): which returns the depth of completed tree counting from the given node(root). We have a basic case that when the given node has no left child or no right child, it returns 1, since it cannot form a completed tree with its children and only itself is a completed tree. The basic case is true.

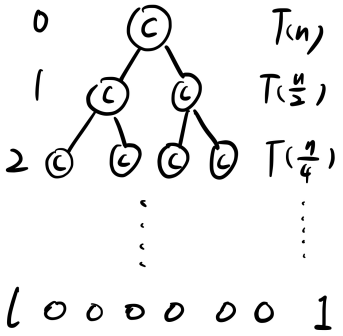
We have an induction hypothesis that for any node in a tree, the function CompleteTreeDepth() returns the correct depth of a complete tree and its child is as root. We have to prove that

CompleteTreeDepth() is also correct to count the depth of the complete tree using current given node as root. According to the definition of completed tree, since the completed tree must have full nodes and both left and right sides must be balance, we only need to consider which child has the minimum depth and only count its depth as part of the parent's completed tree depth. And also the parent does have two children, so the depth counting from parent needs to be added exactly 1. Therefore we have proved that CompleteTreeDepth is correct to return the depth of a completed tree for any node.

So by proving two algorithms, we proved that the whole algorithm is correct.

Time Complexity:

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



■