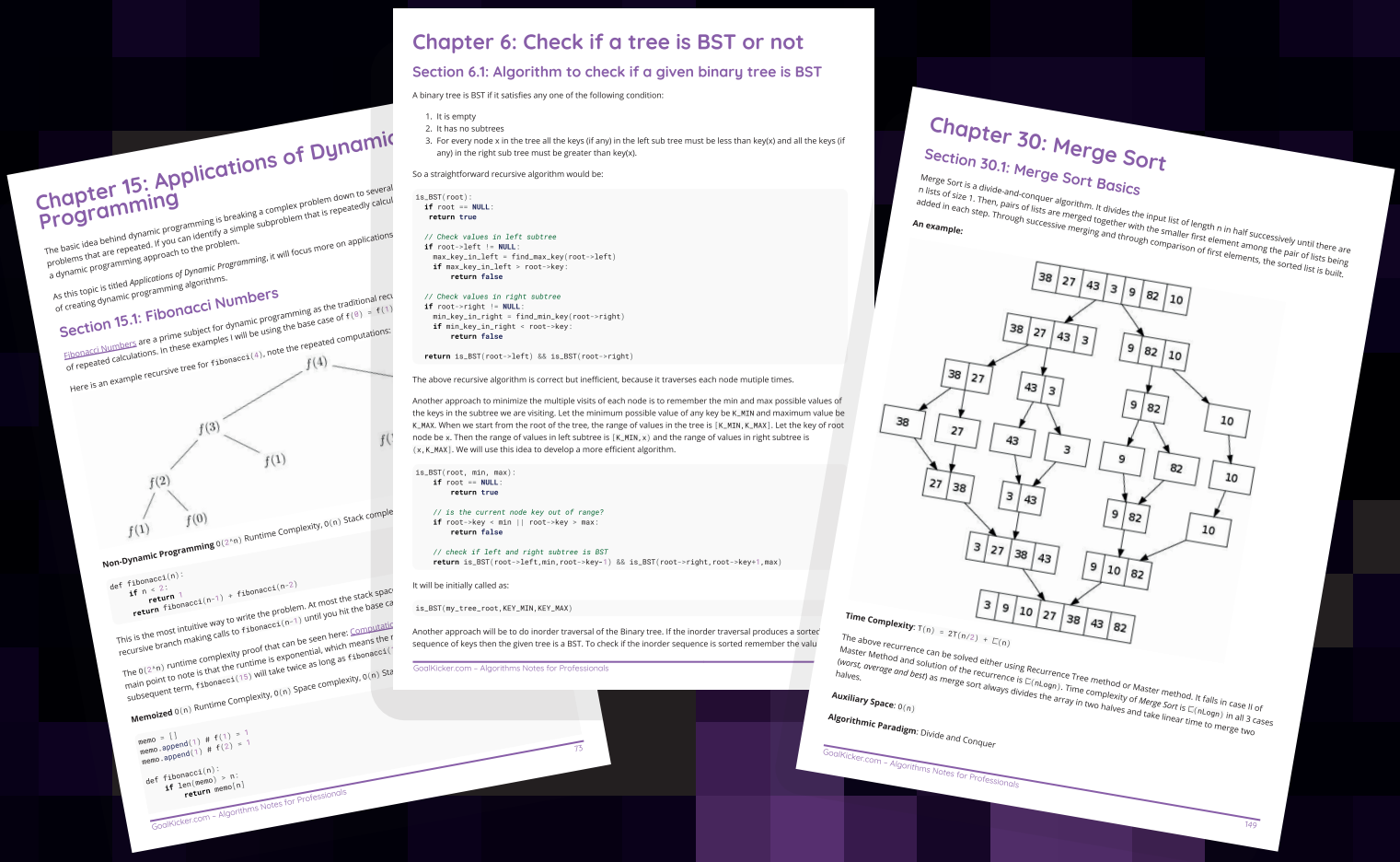
Алгоритмы

Заметки для Профессионалов



Chapter 15: Applications of Dynamic Programming

The basic idea behind dynamic programming is breaking a complex problem down to several problems that are repeated. If you can identify a simple subproblem that is repeatedly calcul a dynamic programming approach to the problem.

As this topic is titled *Applications of Dynamic Programming*, it will focus more on applications of creating dynamic programming algorithms.

Section 15.1: Fibonacci Numbers

Fibonacci Numbers are a prime subject for dynamic programming as the traditional recu of repeated calculations. In these examples I will be using the base case of f(0) = f(1)

Here is an example recursive tree for fibonacci(4), note the repeated computations:

Non-Dynamic Programming O(2^n) Runtime Complexity, O(n) Stack comple

```
def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

This is the most intuitive way to write the problem. At most the stack spac recursive branch making calls to fibonacci(n-1) until you hit the base ca

The O(2^n) runtime complexity proof that can be seen here: Computatio main point to note is that the runtime is exponential, which means the r subsequent term, fibonacci(15) will take twice as long as fibonacci(

Memoized O(n) Runtime Complexity, O(n) Space complexity, O(n) Sta

```
memo = []
memo.append(1) # f(1) = 1
memo.append(1) # f(2) = 1

def fibonacci(n):
    if len(memo) > n:
        return memo[n]
```

GoalKicker.com – Algorithms Notes for Professionals 75

Chapter 6: Check if a tree is BST or not

Section 6.1: Algorithm to check if a given binary tree is BST

A binary tree is BST if it satisfies any one of the following condition:

1. It is empty
2. It has no subtrees
3. For every node x in the tree all the keys (if any) in the left sub tree must be less than key(x) and all the keys (if any) in the right sub tree must be greater than key(x).

So a straightforward recursive algorithm would be:

```
is_BST(root):
  if root == NULL:
    return true

  // Check values in left subtree
  if root->left != NULL:
    max_key_in_left = find_max_key(root->left)
    if max_key_in_left > root->key:
        return false

  // Check values in right subtree
  if root->right != NULL:
    min_key_in_right = find_min_key(root->right)
    if min_key_in_right < root->key:
        return false

  return is_BST(root->left) && is_BST(root->right)
```

The above recursive algorithm is correct but inefficient, because it traverses each node mutiple times.

Another approach to minimize the multiple visits of each node is to remember the min and max possible values of the keys in the subtree we are visiting. Let the minimum possible value of any key be `K_MIN` and maximum value be `K_MAX`. When we start from the root of the tree, the range of values in the tree is `[K_MIN,K_MAX]`. Let the key of root node be `x`. Then the range of values in left subtree is `[K_MIN,x]` and the range of values in right subtree is `(x,K_MAX]`. We will use this idea to develop a more efficient algorithm.

```
is_BST(root, min, max):
    if root == NULL:
        return true

    // is the current node key out of range?
    if root->key < min || root->key > max:
        return false

    // check if left and right subtree is BST
    return is_BST(root->left,min,root->key-1) && is_BST(root->right,root->key+1,max)
```

It will be initially called as:

```
is_BST(my_tree_root,KEY_MIN,KEY_MAX)
```

Another approach will be to do inorder traversal of the Binary tree. If the inorder traversal produces a sorted sequence of keys then the given tree is a BST. To check if the inorder sequence is sorted remember the valu

GoalKicker.com – Algorithms Notes for Professionals

Chapter 30: Merge Sort

Section 30.1: Merge Sort Basics

Merge Sort is a divide-and-conquer algorithm. It divides the input list of length n in half successively until there are n lists of size 1. Then, pairs of lists are merged together with the smaller first element among the pair of lists being added in each step. Through successive merging and through comparison of first elements, the sorted list is built.

An example:

Time Complexity: T(n) = 2T(n/2) + Θ(n)

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is Θ(nLogn). Time complexity of *Merge Sort* is Θ(nLogn) in all 3 cases (*worst, average and best*) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

GoalKicker.com – Algorithms Notes for Professionals 149

200+ страниц

профессиональных советов и уловок

GoalKicker.com
Бесплатные книги по программированию

Отказ от ответственности
Эта неофициальная бесплатная книга создана в учебных целях и она не связана с компаниями или группами, ориентированными на работу с алгоритмами. Все товарные знаки, в том числе зарегистрированные, являются собственностью соответствующих владельцев.