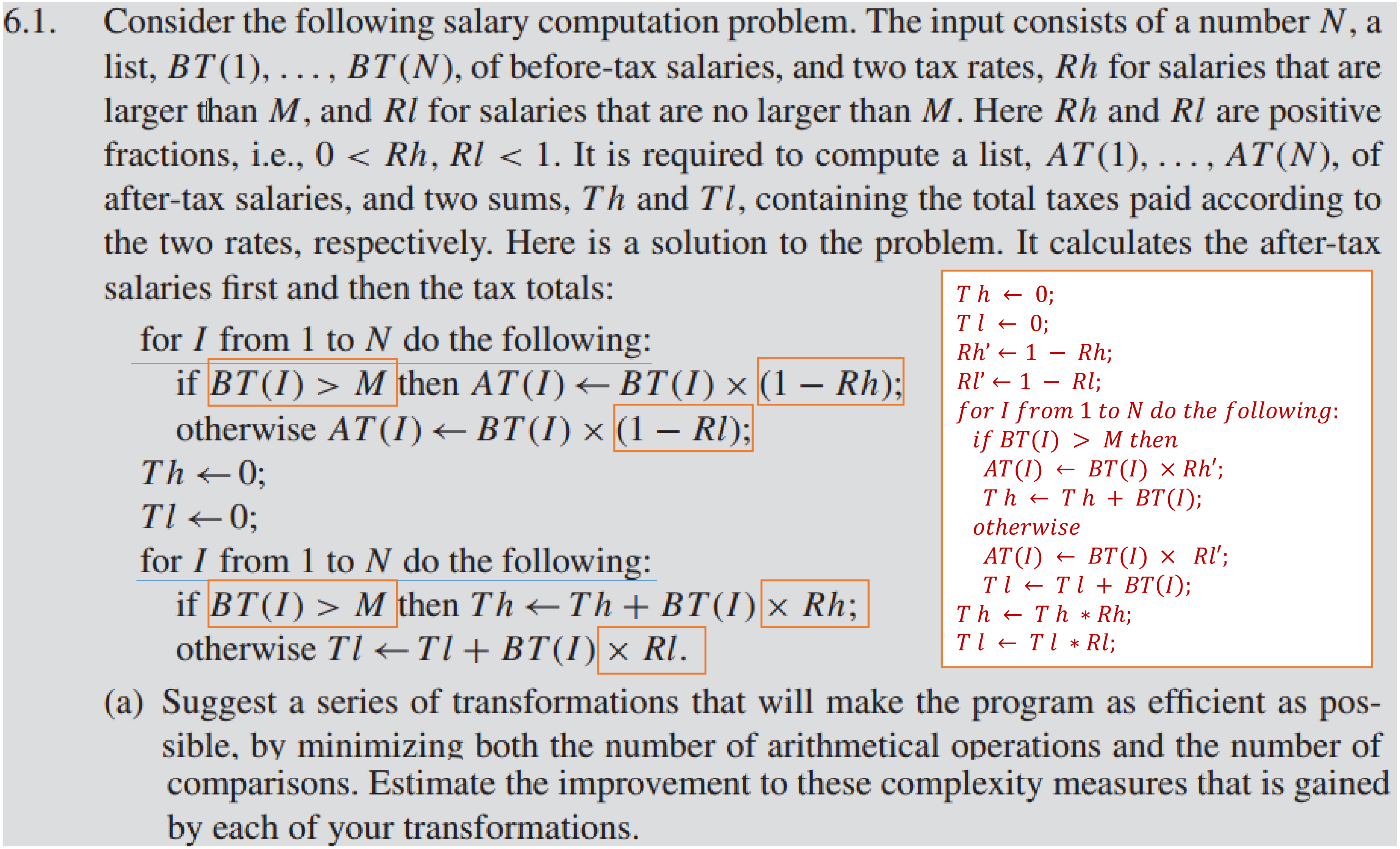
习题2-3

DH第6章练习**1**、2、**3**、**10**、**11**、**15**、16



6.1. Consider the following salary computation problem. The input consists of a number N, a list, $BT(1), \ldots, BT(N)$, of before-tax salaries, and two tax rates, Rh for salaries that are larger than M, and Rl for salaries that are no larger than M. Here Rh and Rl are positive fractions, i.e., $0 < Rh, Rl < 1$. It is required to compute a list, $AT(1), \ldots, AT(N)$, of after-tax salaries, and two sums, Th and Tl, containing the total taxes paid according to the two rates, respectively. Here is a solution to the problem. It calculates the after-tax salaries first and then the tax totals:

```
for $I$ from 1 to $N$ do the following:
    if $\boxed{BT(I) > M}$ then $AT(I) \leftarrow BT(I) \times \boxed{(1 - Rh);}$
    otherwise $AT(I) \leftarrow BT(I) \times \boxed{(1 - Rl);}$
$Th \leftarrow 0;$
$Tl \leftarrow 0;$
for $I$ from 1 to $N$ do the following:
    if $\boxed{BT(I) > M}$ then $Th \leftarrow Th + BT(I) \boxed{\times Rh;}$
    otherwise $Tl \leftarrow Tl + BT(I) \boxed{\times Rl.}$
```

```
$Th \leftarrow 0;$
$Tl \leftarrow 0;$
$Rh' \leftarrow 1 - Rh;$
$Rl' \leftarrow 1 - Rl;$
for $I$ from 1 to $N$ do the following:
  if $BT(I) > M$ then
    $AT(I) \leftarrow BT(I) \times Rh';$
    $Th \leftarrow Th + BT(I);$
  otherwise
    $AT(I) \leftarrow BT(I) \times Rl';$
    $Tl \leftarrow Tl + BT(I);$
$Th \leftarrow Th * Rh;$
$Tl \leftarrow Tl * Rl;$
```

(a) Suggest a series of transformations that will make the program as efficient as possible, by minimizing both the number of arithmetical operations and the number of comparisons. Estimate the improvement to these complexity measures that is gained by each of your transformations.

(b) How would you further improve the program, if you are guaranteed that no before-tax salary is strictly less than M (i.e., there might be before-tax salaries of exactly M, but not less)? How would you characterize the rate of improvement in this case?

```
$Th \leftarrow 0;$
$Tl \leftarrow 0;$
$Rh' \leftarrow 1 - Rh;$
$Rl' \leftarrow 1 - Rl;$
$for\ I\ from\ 1\ to\ N\ do\ the\ following:$
  $if\ BT(I) > M\ then$
    $AT(I) \leftarrow BT(I) \times Rh';$
    $Th \leftarrow Th + BT(I);$
  $otherwise$
    $AT(I) \leftarrow BT(I) \times Rl';$
    $Tl \leftarrow Tl + BT(I);$
$Th \leftarrow Th * Rh;$
$Tl \leftarrow Tl * Rl;$
```

⇒

```
$Th \leftarrow 0;$
$Tl \leftarrow 0;$
$Rh' \leftarrow 1 - Rh;$
$LT \leftarrow M * Rl;$
$for\ I\ from\ 1\ to\ N\ do\ the\ following:$
  $if\ BT(I) > M\ then$
    $AT(I) \leftarrow BT(I) \times Rh';$
    $Th \leftarrow Th + BT(I);$
  $otherwise$
    $AT(I) \leftarrow LT;$
    $Tl \leftarrow LT;$
$Th \leftarrow Th * Rh;$
```

6.3. Analyze the worst-case time complexity of each of the four algorithms given in Exer-
cise 5.15 for computing m^n. Count multiplications only.

Worst Case: $n = 2^k - 1$

iii. Algorithm *Pwr3*:

```
$PW \leftarrow 1$;
$B \leftarrow m$;
$E \leftarrow n$;
while $E \neq 0$ do the following:
    if $E$ is an even number then do the following:
2       $B \leftarrow B \times B$;
        $E \leftarrow E/2$;
    otherwise (i.e., if $E$ is an odd number) do the following:
1       $PW \leftarrow PW \times B$;
        $E \leftarrow E - 1$.
```

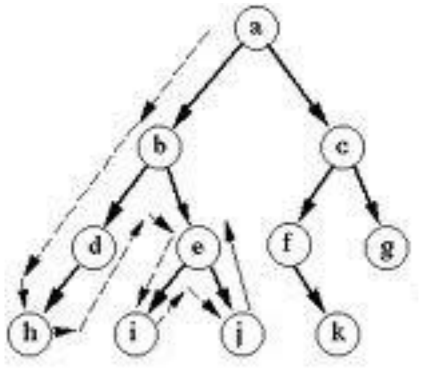
$$f(0) = f(2^0 - 1) = 0$$
$$f(1) = f(2^1 - 1) = 1$$
$$f(3) = f(2^2 - 1) = 3$$
$$f(7) = f(2^3 - 1) = 5$$
$$f(15) = f(2^4 - 1) = 7$$

$$f(n) = f\left(2^k - 1\right) = 2k - 1$$
$$= 2\log_2(n+1) - 1$$

6.10. Analyze the worst-case time complexity of the algorithms for traversing and processing information stored in trees that you were asked to design in Exercises 4.3 and 4.2.

- 4.2.
  - (a) Write an algorithm which, given **a tree *T***, calculates the sum of the depths of all the nodes of *T*
  - (b) Write an algorithm which, given **a tree *T*** and a positive integer *K*, calculates the number of nodes in *T* at depth *K*.
  - (c) Write an algorithm which, given **a tree *T***, checks whether it has any leaf at an even depth.

```
DFS-VISIT(T, u):
    Preorder processing of u;          n
    for each v child of u
        Processing of edge uv(1);      n-1
        DFS-Visit(T,v);
        Processing of edge uv(2);
    Postorder processing of u;
```



Depth-first search

O(n)

```
(a)
•   Input: T-a tree
•   Output: the sum of the depths of all
    nodes of T

sum=0;
DFS-VISIT(T, u, d):
    sum+=d;
    for each v child of u
        d'=d+1;
        DFS-Visit(T,v,d');

主函数：
DFS-Visit(T,T.root,0)
```

4.3. Write algorithms that solve the following problems by performing breadth-first traversals of the given trees. You may assume the availability of a queue Q. The operations on Q include adding an item to the rear, retrieving and removing an item from the front, and testing Q for emptiness.

(a) Given a tree T whose nodes contain integers, print a list consisting of the sum of contents of nodes at depth 0, the sum of contents of nodes at depth 1, etc.

(b) Given a tree T, find the depth K with the maximal number of nodes in T. If there are several such Ks, return their maximum.

```
L ← 0;
add(T, Q);
repeat the following:
    L ← L + 1;
    S ← 0;
    add($, Q);
    remove(V, Q);
    while V ≠ $ do the following:
        S ← S + contents of V;
        I ← 1;
        while V has an Ith offspring do the following:
            VI ← Ith offspring of V;
            add(VI, Q);
            I ← I + 1;
        remove(V, Q);
    print("Sum of contents at level ", L, " is ", S);
until is-empty(Q).
```

Each node joins and leaves the queue exactly once.

O(n)

6.11. Analyze the worst-case time and space complexities of the breadth-first algorithm for checking whether a given tree is balanced you were asked to design in Exercise 4.7.

An arithmetic expression formed by non-negative integers and the standard unary operation "$-$" and the binary operations "$+$", "$-$", "$\times$", and "$/$", can be represented by a binary tree as follows:

- An integer I is represented by a leaf containing I.
- The expression $-E$, where E is an expression, is represented by a tree whose root contains "$-$" and its single offspring is the root of a subtree representing the expression E.
- The expression $E * F$, where E and F are expressions and "$*$" is a binary operation, is represented by a tree whose root contains "$*$", its first offspring is the root of a subtree representing the expression E and its second offspring is the root of a subtree representing F.

Note that the symbol "$-$" stands for both unary and binary operations, and the nodes of the tree containing this symbol may have outdegree either 1 or 2.

We say that two arithmetic expressions E and F are *isomorphic*, if E can be obtained from F by replacing some non-negative integers by others. For example, the expressions $(2 + 3) \times 6 - (-4)$ and $(7 + 0) \times 6 - (-9)$ are isomorphic, but none of them is isomorphic to any of $(-2 + 3) \times 6 - (-4)$ and $(7 + 0) + 6 - (-9)$.
An expression E is said to be *balanced*, if every binary operation in it is applied to two isomorphic expressions. For example, the expressions -5, $(1 + 2) * (3 + 5)$ and $((-3)/(-4))/((-1)/(-100))$ are balanced, while $12 + (3 + 2)$ and $(-3) * (-3)$ are not.

4.7. Design an algorithm that checks whether an expression is balanced, given its tree representation. (Hint: perform breadth-first traversal of the tree.)

Here is an algorithm that checks whether the expression represented by the tree T is balanced. It is based on the observation that the expression is balanced precisely if at every depth of T, either all nodes contain integers or they all contain the same arithmetical operation (binary and unary "–" being considered different). We use a queue Q to perform a breadth-first traversal of the tree and the special item \$ to separate between nodes from different depths inside the queue Q. Initially Q is empty. The result is set in the variable R, which is true upon termination precisely if the expression is balanced.

```
R ← true;
add(T, Q);
while R is true and is-empty(Q) is false do the following:
    add($, Q);
    remove(V, Q);
    I ← 1;
    while V has an Ith offspring do the following;
        VI ← Ith offspring of V;
        add(VI, Q);
        I ← I + 1;
    remove(W, Q);
    while R is true and W ≠ $ do the following:
        if I > 1 and contents of V ≠ contents of W then R ← false;
        J ← 1;
        while R is true and J < I do the following:
            if W has a Jth offspring then do the following:
                WJ ← Jth offspring of W;
                add(WJ, Q);
                J ← J + 1;
            otherwise R ← false;
    remove(W, Q).
```

处理每层第一个节点

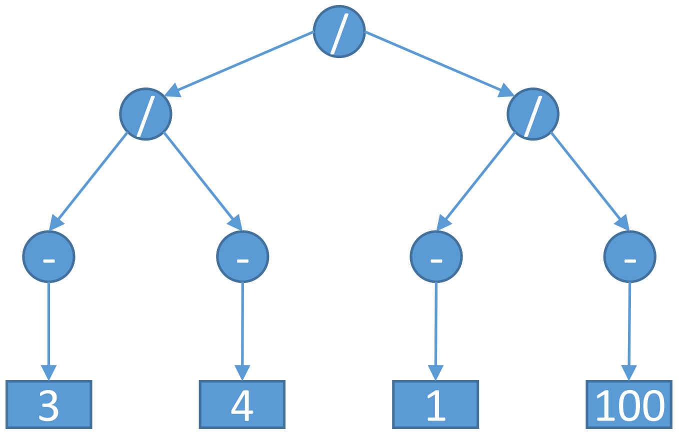
处理同层后续节点

Time: O(n)

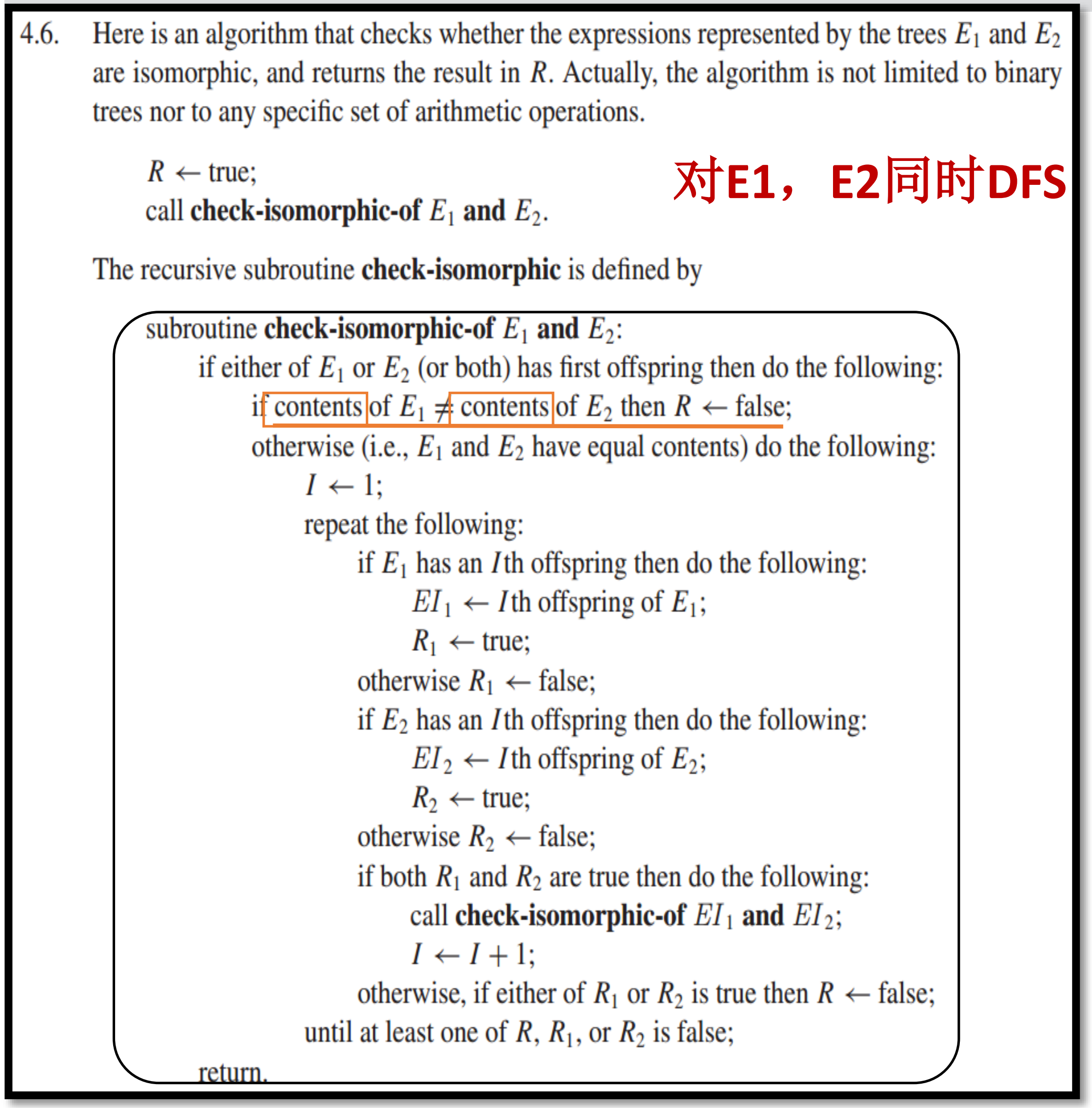
Each node joins and leaves the queue exactly once.

Space: O(n)

$((-3)/(-4))/((-1)/(-100))$



Compare them with the complexities of a straightforward depth-first algorithm for the same problem, which uses the algorithm for tree isomorphism that you were asked to design in Exercise 4.6 as a subroutine applied to the offspring of every node containing a binary operation.



4.6. Here is an algorithm that checks whether the expressions represented by the trees E_1 and E_2 are isomorphic, and returns the result in R. Actually, the algorithm is not limited to binary trees nor to any specific set of arithmetic operations.

```
R ← true;
call **check-isomorphic-of** $E_1$ **and** $E_2$.
```

对E1，E2同时DFS

The recursive subroutine **check-isomorphic** is defined by

```
subroutine check-isomorphic-of $E_1$ and $E_2$:
    if either of $E_1$ or $E_2$ (or both) has first offspring then do the following:
        if contents of $E_1 \neq$ contents of $E_2$ then $R \leftarrow$ false;
        otherwise (i.e., $E_1$ and $E_2$ have equal contents) do the following:
            $I \leftarrow 1$;
            repeat the following:
                if $E_1$ has an $I$th offspring then do the following:
                    $EI_1 \leftarrow I$th offspring of $E_1$;
                    $R_1 \leftarrow$ true;
                otherwise $R_1 \leftarrow$ false;
                if $E_2$ has an $I$th offspring then do the following:
                    $EI_2 \leftarrow I$th offspring of $E_2$;
                    $R_2 \leftarrow$ true;
                otherwise $R_2 \leftarrow$ false;
                if both $R_1$ and $R_2$ are true then do the following:
                    call check-isomorphic-of $EI_1$ and $EI_2$;
                    $I \leftarrow I + 1$;
                otherwise, if either of $R_1$ or $R_2$ is true then $R \leftarrow$ false;
            until at least one of $R$, $R_1$, or $R_2$ is false;
    return.
```

```
DFS-Check($E$)
    if ($E$ is bynary)
        if(DFS-Check(E.L)=false)return $false$;
        if(DFS-Check(E.R)=false)return $false$;
        return check-isomorphic-of E.L and E.R;
    else return true;
```

第i层节点被访问多少次？

0: 1

1: 2

2: 3

K: K+1

Special Case： E is balanced with n binary operation

$$\sum_{k=0\sim\lfloor\log_2 n\rfloor} 2^k * (k+1)$$

$$O(nlgn)$$

使用DFS能够更快么？

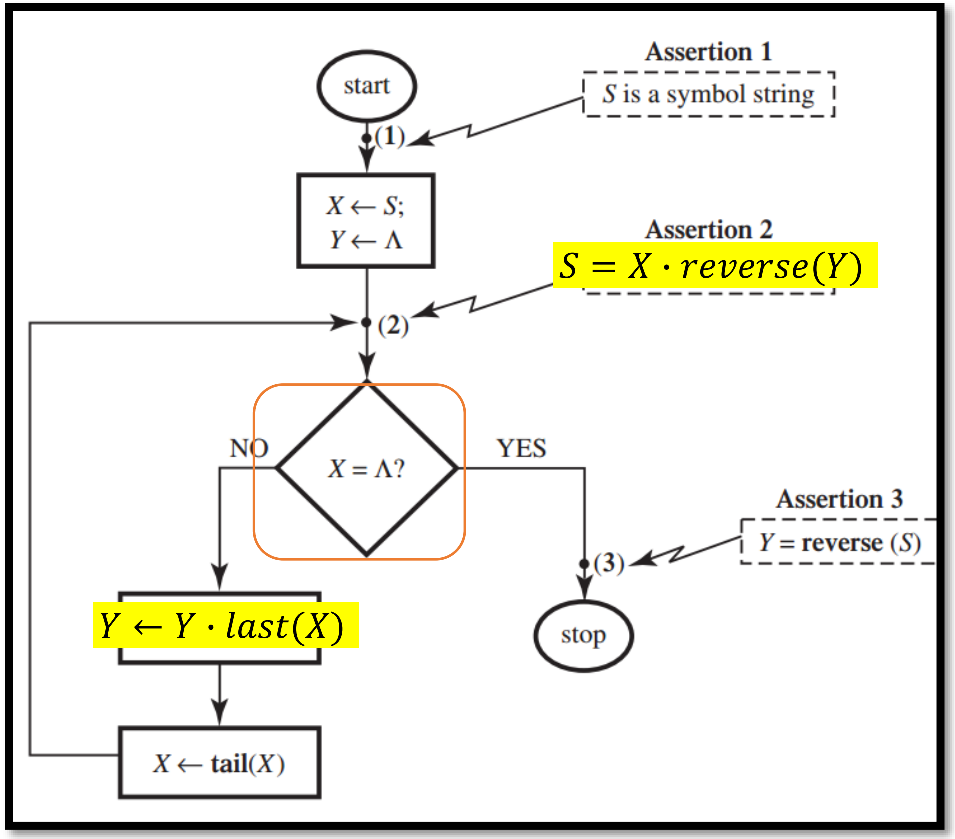
6.15. Recall the problem of detecting palindromes, described in Exercise 5.10. In the following, consider the two correct solutions to this problem: algorithm *Pal1*, presented in Exercise 5.10, and algorithm *Pal4*, which you were asked to construct in Exercise 5.14. Assume that strings are composed of the two symbols "a" and "b" only, and that the only operations we count are comparisons (that is, applications of the **eq** predicate).

(c) Assume a uniform distribution of the strings input by the algorithms. In other words, for each N, all strings of length N over the alphabet {a, b} can occur as inputs with equal probability. Perform an average-case time analysis of algorithms *Pal1* and *Pal4*.

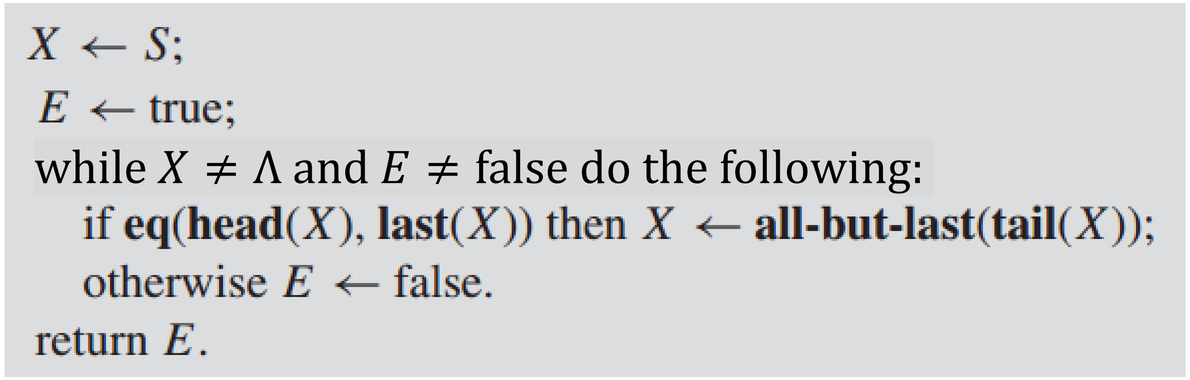
Pal1

```
$Y \leftarrow$ rev($S$);
return equal($S$, $Y$).
```

O(n)



Pal4:



```
$X \leftarrow S$;
$E \leftarrow$ true;
while $X \neq \Lambda$ and $E \neq$ false do the following:
    if **eq**(**head**($X$), **last**($X$)) then $X \leftarrow$ **all-but-last**(**tail**($X$));
    otherwise $E \leftarrow$ false.
return $E$.
```

假设S的长度为2k

S=$a_1 a_2 \dots a_k b_k \dots b_2 b_1$

$$P(a_i = b_i) = P(a_i \neq b_i) = p = \frac{1}{2}$$

1）则S是回文的概率为：$\left(\frac{1}{2}\right)^k = p^k$ K次比较

2）在第i位比较失败的概率：$\left(\frac{1}{2}\right)^{i-1} * \frac{1}{2} = \left(\frac{1}{2}\right)^i = p^i$ i次比较 $\sum_{i=1\sim k} p^i * i$

$$\begin{aligned} Avg(n) &= p^k \cdot k + \sum_{i=1\sim k} p^i \cdot i \\ &= p^k \cdot k + \left(2 - 2p^k - k \cdot p^k\right) \\ &= 2 - 2p^k < 2 \end{aligned}$$

O(1)

假设S的长度为2k+1，可类似处理得到类似结论