Sample Solution for Problem Set 5

Data Structures and Algorithms, Fall 2019

2019 年 12 月 29 日

以下答案未经审核，仅供参考。
部分算导习题未给答案，可以参考https://sites.math.rutgers.edu/ ajl213/CLRS/CLRS.html
后面有些答案只写了思路，如果有看不懂的地方欢迎发邮件咨询

9.1-1 (CLRS 3ed). See Figure 1 for the pseudocode. For simplicity, we assume the input size is a power of 2. The algorithms works as follows: We devide the numbers into pairs, and compare each pair,

```
1:  procedure MIN2($A, n$)
2:      Let $B$ be a new list.
3:      for $i = 1$ to $n/2$ do
4:          Compare $A[2i-1]$ with $A[2i]$, let $m \leftarrow \min(A[2i-1], A[2i])$
5:          Append $m$ into $B$
        $min, min2 \leftarrow$ MIN2($B, n/2$)
6:      Find the index $i$ of $min$ in $A$. That is, $A[i] = min$
7:      if $i$ is odd then
8:          $j \leftarrow i + 1$
9:      else
10:         $j \leftarrow i - 1$
11:     Compare $A[j]$ with $min2$, let $m \leftarrow \min(A[j], min2)$
12:     return $min, m$
```

图 1

and let the smaller numbers be B. The minimum number in A is surely in B. As for the second smallest number in A, it can be

- the number paired with the smallest number
- otherwise, it must win out in its pair, so it should be the second smallest number in B;

So we can recursivly find the smallest two numbers in B, and use them to calculate the smallest two numbers in A, by using an additional comparsion.

Time complexity: $T(n) = T(n/2) + n/2 + 1 \; T(2) = 1 \implies T(n) = n + \log(n) - 2$

9.2-1 (CLRS 3ed). Suppose there is such a call on line 8 or line 9:

- There is a call on line 8: in this case, $p = q$, thus $k = q - p + 1 = 1$, which means $i < k$ can never hold, so line 8 can never be called.
- There is a call on line 9: in this case, $q = r$, thus $k = q - p + 1 = r - p + 1 \geq i$, so line 9 can never be called.

9.3-1 (CLRS 3ed). The algorithm will work if they are devided into groups of 7.

If they are devided into groups of 3: In step 3, following the same argument in the book, we can find that at least $2 \cdot \frac{1}{2}\lceil \frac{n}{3} \rceil \geq \frac{n}{3}$ numbers are larger than the median, so in the worst case $\frac{2n}{3}$ numbers are smaller than the median. So the recurrence we get is $T(n) = T(n/3) + T(2n/3) + O(n)$, which implies $T(n) = \Omega(n \lg n)$, by using the substitution method.

9.3-6 (CLRS 3ed). Suppose without loss of generality that n and k are powers of 2. We first find the $n/2$ th order statistic in time $O(n)$. then reduce the problem to finding the $k/2$ quantiles of the smaller $n/2$ elements and of the larger $n/2$ elements. The time complexity is $T(n) = O(n \log k)$.

9.1-8 (CLRS 3ed). See Figure 2 for the pseudocode.

```
1: procedure MEDIAN($X, Y, n$)
2:     if n = 1 then
3:         return $\min(X[1], Y[1])$
4:     if X[n/2] ¡ Y[n/2] then
5:         return MEDIAN($X[n/2+1, \dots, n], Y[1, \dots, n/2], n/2$)
6:     else
7:         return MEDIAN($X[1, \dots, n/2], Y[n/2+1, \dots, n], n/2$)
```

图 2

Proof of correcteness: If $X[n/2] < Y[n/2]$, then the smallest $n/2$ elements in X are smaller than $X[n/2+1, \dots, n]$ and $Y[n/2+1, \dots, n]$, totally n numbers. So the smallest $n/2$ elements in X are smaller then the median. Similarly, the largest $n/2$ elements in Y are larger then the median. The median of a list of numbers remain the same if we eliminate $n/2$ numbers smaller then the median, and $n/2$ numbers larger the the median. So the median of X and Y reduces to the median of $X[n/2+1, \dots, n]$ and $Y[1, \dots, n/2]$. For the case $X[n/2] > Y[n/2]$, it is totally the same.

The time complexity is $O(\log n)$, because each recursive call reduces the numbers into a half.

9-2 (CLRS 3ed). **a.** Let m_k be the number of x_i smaller than x_k. When weights of $1/n$ are assigned to each x_i, we have $\sum_{x_i < x_k} w_i = m_k/n$ and $\sum_{x_i > x_k} w_i = (n - m_k - 1)/n$. The only value of m_k which makes these sums $< 1/2$ and $\leq 1/2$ respectively is when $\lceil n/2 \rceil - 1$, and this value x must be the median since it has equal numbers of $x_i's$ which are larger and smaller than it.
b. We first sort the n elements into increasing order.It can be done in $O(nlgn)$ worst-case time(using

mergesort or heapsort). Let S_i be the sum of the weights of the first i elements of this sorted array and note that it is $O(1)$ to update S_i. Compute $S_1, S_2, \ldots$ until you reach k such that $S_{k-1} < 1/2$ and $S_k \geq 1/2$.

c. We could modify the SELECT algorithm, after we find the median of medians(let it be x_k), we then compute the total weight of two halves. If the weights of the two halves are each strictly less than 1/2, then the weighted median is x_k . Otherwise, the weighted median should be in the half with total weight exceeding 1/2, and the search continues within the half that weighs more than 1/2. Thus the worst case running time is $T(n) = T(n/2 + 1) + \Theta(n)$.It is still $\Theta(n)$

d. Let p be the minimizer, and suppose that p is not the weighted median. Let ϵ be small enough such that $\epsilon < \min_i(|p - p_i|)$, where we don't include k if $p = p_k$. If p_m is the weighted median and $p < p_m$, choose $\epsilon > 0$. Otherwise choose $\epsilon < 0$. Thus, we have

$$\sum_{i=1}^{n} w_i d(p+\epsilon, p_i) = \sum_{i=1}^{n} w_i d(p, p_i) + \epsilon \left(\sum_{p_i<p} w_i - \sum_{p_i>p} w_i \right) < \sum_{i=1}^{n} w_i d(p, p_i),$$

the difference in sums will take the opposite sign of epsilon. It means that when $p \neq p_m$, when the point is closer to p_m, the sum would be smaller, which means p_m is the minimizer.

e. We need to find a point $p = (x, y)$ that minimizes the sum

$$f(x,y) = \sum_{i=1}^{n} w_i(|x - x_i| + |y - y_i|)$$

,We can express the cost function as the sum of two functions of one variable each:$f(x, y) = g(x)+h(y)$, where $g(x) = \sum_{i=1}^{n} w_i(|x - x_i|)$ and $h(y) = \sum_{i=1}^{n} w_i|y - y_i|)$. Because g does not depend on y and h does not depend on x, $minf(x, y) = ming(x) + minh(y)$. We simply take $p = (p_x, p_y)$ to be such that p_x is the weighted median of the x-coordinates of the p_i's and py is the weighted medain of the y-coordiantes of the p_i's.

9-3 (CLRS 3ed). a. If $i \geq n/2$, we just use SELECT. Otherwise, split the array in pairs and compare each pair. We take the smaller elements of each pair, but keep track of the other one.We recursively find the first i elements among the smaller elements. The ith order statistic is among the pairs containing the smaller elements we found in the previous step. We call SELECT on those 2i elements. That's the final answer.

b. By the Substitution method, suppose that $U_i(n) = n + cT(2i)lg(n/i)$ for smaller n, then, there are two cases based on whether or not $i < n/4$. If it is, $Ui(n) = bn/2c + U_i(\lceil n/2 \rceil) + T(2i) \leq n/2 + n/2 + cT(2i)lg(n/2i) + T(2i)$. This then satisfies the recurrence if we have that $c \geq 1$. The other case is that $n/4 \leq i < n/2$. In this case, we have that $U_i(n) = n/2+T(\lceil n/2 \rceil)+T(2i) \leq n/2+2T(2i)$, which works if we have $c \geq 2$. So, we can just pick $c = 2$, and both cases of the recurrence go through.

c. When i is a constant, $T(2i) = O(1)$ and $lg(n/i) = lgn - lgi = O(lgn)$. Thus, when i is a constant less than $n/2$, we have that

$$U_i(n) = n + O(T(2i)lg(n/i)) = n + O(lgn)$$

d. Suppose that $i = n/k$ for $k \geq 2$. Then $i \leq n/2$.
If $k > 2$, then $i < n/2$, and we have

$$U_i(n) = n + O(T(2i)lg(n/i)) = n + O(T(2n/k)lg(n/(n/k)) = n + O(T(2n/k)lgk)$$

If $k = 2$, then $n = 2i$ and $lgk = 1$. We have

$$Ui(n) = T(n) = n+(T(n)-n) \leq n+(T(2i)-n) = n+(T(2n/k)-n) = n+(T(2n/k)lgk-n) = n+O(T(2n/k)lgk)$$

10.4-4 (CLRS 3ed).

```
procedure PRINT-TREE(u)
    if u==NULL then
        return
    Print(u.key)
    PRINT-TREE(u.left-child)
    PRINT-TREE(u.right-sibling)
```

图 3

Additional Problem 1 Use mathematical indution to prove the uniqueness:

hypothesis: For a n node tree $n \geq 1$ given pre-order and in-order numbers assigned, the tree is unique.

basic: For $n = 1$, there is only one possible order, thus, it is right.

induction: Given the pre-order numbers $a_1, ..., a_n$ and in-order numbers $b_1, ..., b_n$, according to the definition of in-order, a_1 must be the root. Suppose $b_k = a_1$ for the unique k, according to the definition of in-order, $b_1, ..., b_{k-1}$ must be the in-order of the left sub-tree of the root and $b_{k+1}, .., b_n$ must be the in-order of the right sub-tree of the root. Since the size of the sub-trees is known, $a_2, ..., a_k$ is the pre-order of the left sub-tree of the root and $a_{k+1}, ..., a_n$ is the pre-order of the right sub-tree of the root. According to the hypothesis, since $k - 1 < n$, we know the left sub-tree and right sub-tree is unique, finishing the prove.

The way to construct the tree is naturally given in the proof above.

Additional Problem 2 Sorry for the mistake in problem (a), there is no O(1) algorithm to solve (a) unless the tree is a full-binary tree. So you will get full marks for (a) if you answer “no solution” or any **reasonable** solutions for the problem.

(a)Compute the size of sub-tree rooted at u:

If $u.p = NULL$, the size is $u.post\_order\_number$.

Else, if u is the left child of its father, the size is $u.p.right\_child.pre\_order\_number - u.pre\_order\_number$. Or u is the right child of its father, the size is $u.post\_order\_number - u.p.left\_child.post\_order\_number$.

(b) u is v's ancestor is and only if $u.pre\_order\_number < v.pre\_order\_number \wedge u.post\_order\_number > v.post\_order\_number$.

Sample Solution for Problem Set 6

Data Structures and Algorithms, Fall 2019

December 29, 2019

12-1 (CLRS 3ed). -
a. $\Theta(n^2)$. Only the right child will be chosen, so the tree deteriorates into a list.
b. $\Theta(n \log n)$. It's easy to prove that for all sibling subtrees, their size differ at most by 1. For each $k > 0$, we can prove by induction on k that for $n = 2^k$, the height of the tree is $k = \Theta(\log n)$. So the tree is balanced, and the bound holds.
c. $\Theta(n)$. We are actually inserting all nodes into one list.
d. Worst case: $\Theta(n^2)$. Because the tree may deteriorate into a list.
Expected: $\Theta(n \log n)$. Because informally, the tree is roughly balanced.

13.1-4 (CLRS 3ed). The possible degrees are 0 to 5, depending on how many red children a black node has, and whether it is the root.

The heights decrease at most by half, because for each path from the root to the leaf, at most half of the nodes can be red.

13.2-4 (CLRS 3ed). Figure 1 is a simple algorithm that transforms a tree into a chain:

```
RotateToChain($T$)
1: while $T$ is not a right chain do
2:     Starting from $T.root$, we go right down to the tree, until we find a node $x$ with a left child.
3:     Perform right rotate on $x$
```

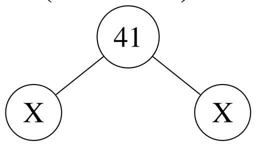
Figure 1

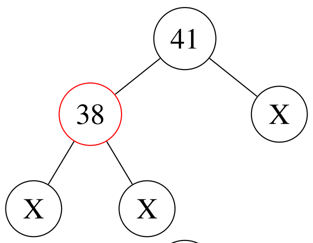
This simple algorithm is correct and uses at most $n - 1$ rotates because, considering the chain C formed by all successive right nodes starting from the root

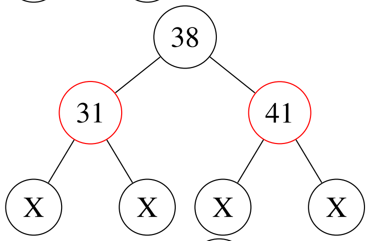
- If T is not a chain, then there exists a node x in C that has a left children;
- By performing a right rotation on x, the size of C increases by 1.

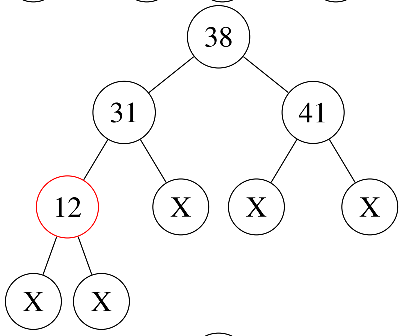
Note that left and right rotations are reversible, so given two trees S and T, if we can transform S into T, than we can also transform T back into S. Thus since any tree can be transformed into chains using $n-1$ rotations, a chain can be transformed into any tree also using $n-1$ rotations, which completes the problem.

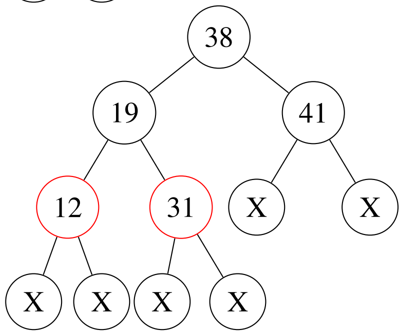
13.3-2 (CLRS 3ed). -











13.4-3 (CLRS 3ed).

13-3 (CLRS 3ed). **a.** We prove by induction that an AVL tree with height h has at least a^h nodes(where a will be determined later).

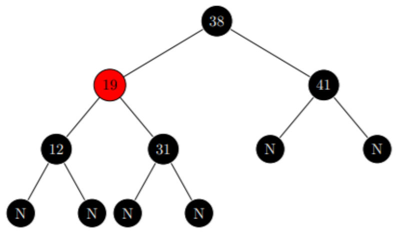
For a tree with height 1, it is true. For a tree with height h, we can remove the root and get two subtrees with height at least $h-2$ and there must be one subtree with height $h-1$ (or the height of the tree can't get h). According to the induction hypothesis, we have the lower bound for the tree

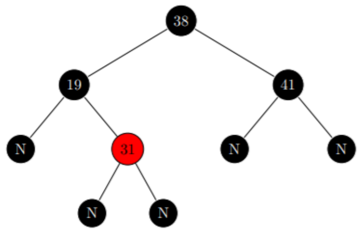
$$a^{h-1} + a^{h-2} + 1 > (a+1) * a^{h-2} > a^2 * a^{h-2} = a^h$$

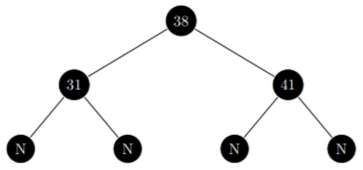
The inequality hold when $a = 1.5$. So for a tree with n node, its height is at most $\log_a n = O(\lg n)$.

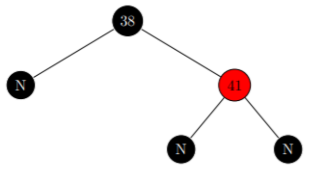
b. c. Suppose each node has attribute $.l$ and $.r$ which denote the left child and right child, $.h$ denotes the height of the subtree,$.k$ denotes the key value of the node,$.p$ denotes the father of the node. Function $right\_rotate$ and $left\_rotate$ is defined identical to it in the red-black tree. The procedure $INSERT(u, x)$ insert node x to the subtree rooted at u, where we already set $x.h = 1, x.l = NIL, x.r = NIL$. For convenience, each node with no left or right child has a left or right child NIL with $NIL.h = 0$.The pseudocode:

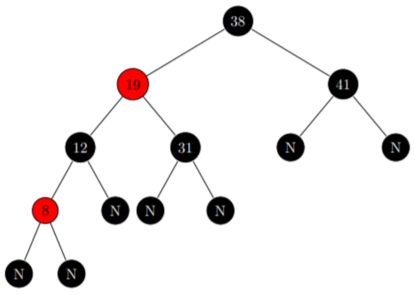
d. The procedure $BALANCE$ takes O(1) time obviously. Thus the INSERT operation has recursion $T(h) = T(h-1) + O(1)$, where h is the height. It takes $O(h)$ time and $h = O(\lg n)$. Also, only

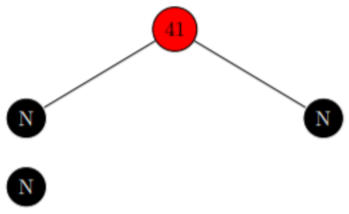

38
19
41
12
31
N
N
N
N
N
N


38
19
41
N
31
N
N
N
N


38
31
41
N
N
N
N


38
N
41
N
N


38
19
41
12
31
N
N
8
N
N
N
N
N


41
N
N
N

```
procedure BALANCE(node u)
    if $u.l.h > u.r.h + 1$ then
        v=u.l
        if $v.r.h > v.l.h$ then
            left_rotate(v)
        right_rotate(u)
    else
        v=u.r
        if $v.l.h > v.r.h$ then
            right_rotate(v)
        left_rotate(u)
procedure INSERT(u,x)
    if $u.k > x$ then
        if $u.l = NIL$ then
            u.l=x
            x.p=r
        else
            INSERT(u.l,x)
    else
        if $u.r = NIL$ then
            u.r=x
            x.p=u
        else
            INSERTu.r,x
    $u.h = max(u.l.h, u.r.h)$
    BALANCE(u)
```

Figure 2

a single rotation will occur in the second while loop because when we do it, we will be decreasing the height of the subtree rooted there, which means that it's back down to what it was before, so all of it's ancestors will have unchanged heights, so, no further balancing will be required.

Additional Problem 1 **(a)**For sample an array $P = (p_1, ..., p_n)$ from $(0, 1)$ independently and randomly as the priority of the Treap (we do not consider the case that the priority may be overlap). We now show how to construct a Treap in $O(n)$ time according to the given key value and priority. (Note that when fixing the key value and priority of each node, the structure of the Treap is unique. That proporty can be proved easily by induction).

The Treap is a Max-heap. The node in the following pseudocode follows the rules defined in problem 13-3.(The attributes and NIL, with a new attribute $.pri$ denoting the priority of the node). At the beginning the the algorithm, C is a new stack with element type Node and one initial element NIL as root node (for convinence). And we set $NIL.pri = 1$, which is the max one.

The algorithm runs in $O(n)$ time obviously since each node push and pop from C at most once. It can also be varified that the proporty of the Treap is maintained during the procedure of the for loop. At last the Treap is rooted at $NIL.r$.

(b) Supose a skip list A is constructed by the following way: the height of the skip list is $A.h$, and there are h head nodes $A[1], ..., A[h]$ which are the heads of the h layers of the skip list. Each node has attribute $.next$ point to the next node in the same layer and $.down$ to the lower layer at the same position, and $.k$ is the key value of the node, the skip list garanttee that the key value is in increasing order.

So we can see the array for a skip list is actually $A[1]$, $A[1].next$, $A[i].next.next$... until it get NIL,

```
procedure BUILD_TREAP(A,P)
    C=new Stack(Node)
    C.push(NIL)
    for $i = 1$ to $n$ do
        u=new Node()
        u.pri=P[i], u.k=A[i]
        while $C.top().pri < P[i]$ do
            v=C.pop()
        C.top().r=u, u.p=C.top()
        u.l=v, v.p=u
        C.push(u)
```

Figure 3

we set $NIL.k = -\infty$ for convenience.

Given all those convention, we give the procedure to combine two skip list A and B. For convenience, we set $A[i] = NIL$ for $A.h < i \leq B.h$ if $B.h > A.h$, and vice versa:

```
procedure MERGE(A,B)
    C.h=max(A.h,B.h)
    for $i = 1$ to $C.h$ do
        if A[i].k¡B[i].k then
            C[i]=A[i]
            C[i].next=B[i]
        else
            C[i]=B[i]
            C[i].next=A[i]
        cur=C[i].next
        cura=A[i]
        curb=B[i]
        while $cura.next \neq NIL$ or $curb.next \neq NIL$ do
            if $cura.next.k < curb.next.k$ then
                cur.next=cura.next
                cura=cura.next
            else
                cur.next=B.next
                curb=curb.next
        cur.next=NIL
    return C
```

Figure 4

Sample Solution for Problem Set 7

Data Structures and Algorithms, Fall 2019

December 29, 2019

11.2-3 (CLRS 3ed). **successful/unsuccessful searches:** Note that sorted link-list do not imporve the searching speed, since we always need $O(n)$ time to access to any element in link-list. Thus, the searching time do not change, it is $O(1+\alpha)$.
insert: Since we must find the right place to insert an element, it takes $O(1+\alpha)$ time.
delete: It doesn't change, it is $O(1)$.

11.2-6 (CLRS 3ed). We choose two number from 1 to m and 1 to L independently and uniformly, noted as a_1 and a_2. Then we try to ouput the a_2^{th} element in the a_1 link-list. If a_1 link-list's length is less then a_2, we do the procedure again. The correctness is trivial.

Time analysis: each time the probability for the algorithm to stop is $\frac{n}{L*m} = \frac{\alpha}{L}$. Thus the expected running time is

$$1 + \alpha + \frac{L}{\alpha} < L + \frac{L}{\alpha} = O(L(1+\alpha))$$

11.2-6 (CLRS 3ed). Suppose the digit for x is $x_0, ..., x_n$ from lower to higher bit. Then the hash value for x is

$$(\sum_{i=0}^{n} m^i * x_i)\%(m-1) = (\sum_{i=0}^{n}(m^i * x_i\%(m-1))\%(m-1) = (\sum_{i=0}^{n} x_i)\%(m-1)$$

It is not depend on the permutation of the digit.

The hash function can be used to distinguish different set, which don't care about the permutation of the elements in it.

11-1 (CLRS 3ed). -
a. Since $n \le m/2$, each probe has probability at most $1/2$ to fail. Thus to have strictly more than k probes, the first k probes must fail, which is of probability at most 2^{-k}.
b. $2^{-2\lg n} = O(1/n^2)$.
c. $\Pr(X > 2\lg n) = \Pr(\exists i, X_i > 2\lg n) \le \sum_{i=1}^{n} \Pr(X_i > 2\lg n) = O(1/n)$. The inequality follows from the union bound.
d. The longest probe length is n. So $\mathbb{E}(X) \le n \Pr(X > 2\lg n) + 2\lg n = n\frac{1}{n} + 2\lg n = O(\lg n)$.

Additional problem ONE -
a. Let $X_{i,j}$ be a random variable indicating that $h(i) = h(j)$, and let $X = \sum_{i<j} X_{i,j}$. Then $\mathbb{E}(X) = \sum_{i<j} \mathbb{E}(X_{i,j}) = \frac{n(n-1)}{2} \Pr[h(i) = h(j)] = \frac{n(n-1)}{2m}$.
b. The number of functions from $\{0, 1, \ldots, n-1\}$ to $\{0, 1, \ldots, m-1\}$ is m^n. The number of injections from $\{0, 1, \ldots, n-1\}$ to $\{0, 1, \ldots, m-1\}$ is $m^{\underline{n}} = m(m-1)\ldots(m-n+1)$. So the probability that a function is perfect is $P = \frac{m^{\underline{n}}}{m^n}$.
c. $\frac{1}{P} = \frac{m^n}{m^{\underline{n}}}$.
d. $(1-P)^N = \left(1 - \frac{m^{\underline{n}}}{m^n}\right)^N$
e. $(1-P)^N = \left(1 - \frac{m^{\underline{n}}}{m^n}\right)^N < 1/n \implies N = \Omega(\frac{m^n}{m^{\underline{n}}} \ln n)$.

Sample Solution for Problem Set 8

Data Structures and Algorithms, Fall 2019

2019 年 12 月 29 日

17-1 (CLRS 3ed). -

a. See Figure 1 for the pseudocode.

```
Bit-Reversal-Permutation($A, v$)
1: for $i = 0$ to $n - 1$ do
2:     $j \leftarrow \mathrm{rev}_k(i)$
3:     if $i < j$ then
4:         Swap $A[i]$ with $A[j]$
```

图 1

b. See Figure 2 for the pseudocode. The amortized time complexity is exactly the same as the

```
Bit-Reversed-Increment($a$)
1: $m \leftarrow 2^{k-1}$                          ▷ This is a constant, thus takes $O(1)$ time
2: while $m$ AND $a \neq 0$ do
3:     $a \leftarrow a$ XOR $m$
4:     $m \leftarrow m/2$                                                  ▷ Right shift
5: $a \leftarrow a$ XOR $m$
6: return $a$
```

图 2

binary counter, which is $O(n)$.

c. Yes. See problem b.

21.1-3 (CLRS 3ed). Find-Set: Each edge causes 2 calls to Find-Set, so $2|E|$ calls are made. Union: Notice each call to Union decreases the number of sets by 1. At the begining, each node is in it's own set, so there are $|V|$ sets. At the end, there are exactly k components, thus exactly k sets. So $|V| - k$ calls are required.

21.2-4 (CLRS 3ed). The running time is $\Theta(n)$, because each call to Make-Set takes time $\Theta(1)$, and for each call to Union, the smaller set has size 1, thus takes time $O(1)$.

21.3-3 (CLRS 3ed). See Figure 3 for the pseudocode.

```
LinearSearch($A, v$)
$c \leftarrow 0$
for $i = 1$ to $n$ do
    MAKE-SET($i$)
    $c \leftarrow c + 1$
Let $k$ be the largest integer such that $2^k \leq n$
for $j = 1$ to $k$ do
    $i \leftarrow 1$
    while $i \leq n$ do
        UNION($i, i + 2^{j-1}$)
        $c \leftarrow c + 1$
        $i \leftarrow i + 2^j$
while $c < m$ do
    FIND-SET(1)
```

图 3

In this problem, we assume m is large enough. The counter c is simply used to count the number of Make-Set and Union operations, so that we know how many (in fact, $m - c$) Find-Set operations we can make.

21.3-4 (CLRS 3ed). We add an attribute with two pointers(or two attributes), one pointing to the first child of a node, and the second pointing to the next sibling of a node. When we call Make-Set, we initialize these two pointers with NULL. When we call Union, suppose we are merging two sets, a and b, and that a has smaller size. Then we let a's next sibling be b's first child, and then let b's first child be a. In this way, we can print all members in a set recursively, by adopting a DFS.

21-1 (CLRS 3ed). **a.**

index	value
1	4
2	3
3	2
4	6
5	8
6	1

b. Suppose the element $u \in I_i$ is picken by E after I_j during the algorithm, we must show that it is true in the real case. u is not picken by E between I_i and I_j is because it is deleted by other elements before u. Since the array has already been sorted, those elements must be less then u and in I_i to I_{j-1}. In the real case it is true since u will not be picken out unless those elements has been picken. Thus u must be picken by E just after I_j.

c. Use the procedure $Union$ and $Makeset$ and $Find$ just discribed in the book. (With path compression and height rank) First we use $Makeset(i)$ for each $i \in [n]$ to create n nodes. Note that each node has two attribute key and id where key is the value in the node and id is the i that $key \in I_i$. Then for each I_i we use $Union(u, v)$ for all $v \in I_i$. We maintain another array $A[i]$ contain one representative element in I_i. That's the preperation.

To determine j such that $i \in K[j]$, we extract $find(i).id$. To implement $K[l] = K[j] \cup K[l]$, we just use $Union(i, A[j+1])$ and update $find(i).id$ to $j+1$.

The algorithm takes $n\alpha(n)$ amortized time according to the analysis in the book.

Additional problem 1 **a.** It is trivial to use DFS to compute the components of black blocks. We set $B[i,j] = 1$ if it is black, or 0 if it is white. We first initial each $vis[i,j]$ to 0, then do dfs:

```
1: procedure DFS(i,j)
2:     if vis[i,j]=1 or B[i,j]=0 thenreturn 0
3:         vis[i,j]=1
4:     $ans \leftarrow 0$
5:     for each $(x, y)$ adjacent to $(i, j)$ do
6:         $ans \leftarrow ans + DFS(x, y)$
7:     return ans
8: procedure COMPUTE(B)
9:     $ans \leftarrow 0$
10:    for $i \in [n], j \in [n]$ do
11:        $ans \leftarrow max(ans, DFS(i, j))$
12:    return ans
```

图 4

b. As in $21-1$, we use the union set function just as discribed in the book. For each black block, we create a union set for it use $Makeset((i, j))$. Each set for (i, j) maintain an attribute as the size of the size. Thus we sould do some change to procedure $Union((i, j), (x, y))$: just add a new line as take $find((i, j)).size + find(x, y).size$ as the new size. Then we adding a new black block, we just union the block adjacent to it. According to the amortized analysis, it takes $O(\alpha(n))$ each time.

c. We claim the anser is $\Theta(\log n)$. There are two things we need to prove: First is that it can be greater then $\Theta(\log n)$. It is trivial to see that the largest time is the longest path it can occur in the tree from one of those set. We prove the height of the tree can't be greater then $\Theta(\log n)$ if we use height rank.(let the tree with lower height be the son of another tree's root). We ust mathematical induction to prove: the tree's number of nodes for each set is at least 2^h where h is the height of the tree. That is trival in the case of $n = 1$ or $n = 2$. We focus on the induction step: if at procedure the height is increasing to be $h + 1$(it is only possible to be imcreased by 1). That can occur only when two of the previous height h tree to be union. According to mathematical induction those tree has nodes at least $2^h + 2^h = 2^{h+1}$ which maintain the hypothesis. Thus we finish the prove, and the tree's height can't be greather then $\Theta(\log n)$.

Second, we should use a example to show in some case it is truely $\Theta(\log n)$. Consider this example: each time we want to union two tree, we always pick the two node that do not have the longest path to the root to union. The path comprision then can't be effective because there are always a longest path from root to another node. Each time we union two exatly identical tree and the height is increased by 1, so when ther are $\Theta(n)$ nodes, the height is $\Theta(\log n)$.

Sample Solution for Problem Set 9

Data Structures and Algorithms, Fall 2019

2019 年 12 月 29 日

22.1-3 (CLRS 3ed). **a.** For adjacency-list representation:

```
1: Let Adj'[1...—V—] be a new adjacency list of the transposed $G^T$
2: for $u \in G.V$ do
3:     for $v \in Adj[u]$ do
4:         $INSERT(Adj'[v], u)$
```

图 1

The time complexity is $O(|V| + |E|)$ since the outer loop cost $O(|V|)$ time and the inner loop cost $O(|E|)$ totally.

b. For Adjacency-matrix representation we can just transpose the original matrix, it costs $O(|V|^2)$.

22.1-5 (CLRS 3ed). **a.** For adjacency-list representation:

```
1: Let Adj'[1...—V—] be a new adjacency list of the transposed $G^T$
2: for $u \in G.V$ do
3:     for $v \in Adj[u]$ do
4:         for $r \in Adj[v]$ do
5:             $INSERT(Adj'[r], u)$
```

图 2

Where $Adj'[1...|V|]$ is the adjacency-list for G^2. The time complexity is $O(|V||E|)$ since the outer loop cost $O(|V|)$ time and each of the inner (and inner inner)loop cost $O(|E|)$ at most.

b. For Adjacency-matrix representation we can just compute A^2 for the adjacency-matrix for G^2 where A is the adjacency-matrix for G. The time complexity can be $O(V^3)$ or better using a more clever matrix multiplication algorithm.

22.1-6 (CLRS 3ed). For suppose the graph is a simple directed graph ((u, v) and (v, u) can't both exist). Then each time we get a information about an edge (u, v) we can know v is not an universal sink.

Each access to an adjacency-matrix can help us eliminate a possible. There are at most $|V|$ possible answer, thus the complexity is $O(|V|)$.

Formally, we maintain an anser set $S = \{1, ..., n\}$ at first where $n = |V|$. Then we do n loops: each time we access to the adjacency-matrix (u, v) ramdomly where $u \in S, v \in S$. Then we can eliminate u or v from S. After $n-1$ times loops, S will only contain one element, that is the only possible universal sink, we can use $O(|V|)$ to check the answer. The total cost is $O(|V|)$.

Note that the answer in $walkccc.github$ is just a special case of the algorithm.

22.2-3 (CLRS 3ed). The textbook introduces the GRAY color for the pedagogical purpose to distinguish between the GRAY nodes (which are enqueued) and the BLACK nodes (which are dequeued). We can just treet the two color as one color. it suffices to use a single bit to store each vertex color.

22.2-6 (CLRS 3ed). Consider the graph with 5 vertices $\{a, b, c, d, e\}$ and 6 edges $\{(a, b), (a, c), (b, d), (b, e), (c, d), (c, e$
No matter what order you use to visit those vertices, the bfs tree rooted at a will contain $\{(a, b), (a, c)\}$ and $\{(b, d), (b, e)\}$ or $\{(c, d), (c, e)\}$ depending on witch vertex you visit first between b, c. So the tree $\{(a, b), (a, c), (b, d), (c, e)\}$ can't get from bfs.

$$x * (1 - \frac{1}{(1+x)^{\frac{1}{x}}}) < \ln(x+1)$$

22.4-2 (CLRS 3ed). See Figure 3 for the pseudocode.

```
CountPath($G, s, t$)
1: TOPOLOGICAL-SORT($G$)
2: give each node a new attribute $count$, with initial value 0
3: $t.count \leftarrow 1$
4: $p \leftarrow t$
5: while $p \neq s$ do
6:     $p \leftarrow p.prev$                    ▷ $prev$ is the previous node in the list
7:     for $(p, q) \in G.E$ do
8:         $p.count \leftarrow p.count + q.count$
9: return $s.count$
```

图 3

The code works by first topologically sorting the nodes in the graph G. Then we assign each node an attribute $count$, denoting the number of paths from each node to the node t. The nodes after node t in the list can't reach t, so have $count$ set to 0. Node t has exactly one way to itself, so it's $count$ is 1.

And for each node p previous to t in the list, we enumerate the first node q in the path from p to t. There are exactly $q.count$ paths from p to t that goes through q, so the number of paths from p to t is the sum of these numbers.

Finally we return $s.count$, which is the desired number of pathes we want.

```
TopoSort($G$)
1: Let $D[1..n]$ be a new array, initially all 0
2: Let $Q$ be a new empty queue
3: Let $L$ be a new list
4: for $(u, v) \in G.E$ do
5:     $D[v] \leftarrow D[v] + 1$
6: for $u \in G.V$ do
7:     if $D[u] = 0$ then
8:         $Q$.PUSH($u$)
9: while not $Q$.EMPTY() do
10:     $u \leftarrow Q$.POP()
11:     $L$.INSERT($u$)
12:     for $(u, v) \in G.E$ do
13:         $D[v] \leftarrow D[v] - 1$
14:         if $D[v] = 0$ then
15:             $Q$.PUSH($v$)
16: return $L$
```

图 4

22.4-5 (CLRS 3ed). See Figure 4 for the pseudocode.

If G has a cycle, then the algorithm will return an empty list, because the first time the algorithm reaches line 9, the queue is empty.

Sample Solution for Problem Set 10

Data Structures and Algorithms, Fall 2019

2019 年 12 月 29 日

22.5-3 (CLRS 3ed). No

22.5-7 (CLRS 3ed). Note that if $u \to v$ and $v \to w$, then $u \to w$, i.e., connectivity is transitive. For two strongly connected componenets (SCC) $A, B \subseteq V$, we say $A \to B$ if $a \to b$ for some $a \in A$ and $b \in B$.

We claim that for a graph, it is semiconnected if and only if the connectivity relation among SCC's is a total order (check the definition of total order).

So our goal is to check whether the connectivity relation is a total order for the SCC's. The first step of the algorithm is of course to calculate the SCC's, which form a DAG G', with vertices being SCC's in the original graph, and edge set also inherited from the original graph. Let $S_1, S_2, \ldots, S_k$ be the vertices in the DAG G', we can assume $S_1 \to S_2 \to \cdots \to S_k$, if it forms a total order. Note that G' has only one vertex with in-degree 0, which is S_1. And if we elminiate S_1 from the DAG, the remaining graph also hase only one vertex with in-degree 0, which in this case is S_2. And so on.

So we can use the simple algorithm in figure 1 to check whether there is a total order in the DAG. Note that the input for the algorithm is the DAG formed by the SCC's, not the original graph. The total time complexity is $O(n)$, since each step above takes time $O(n)$.

```
CheckTotalOrder($G$)
1: Calculate the in-degree for each vertices in $G$.
2: while $G$ is not an empty graph do
3:     if $G$ has more than one vertex with in-degree $0$ then
4:         return False
5:     Eliminate the only vertex in $G$ with in-degree $0$, and update the in-degree for its adjacent vertices
6: return True
```

图 1

22-4 (CLRS 3ed). Step 1: Construct the components graph;
Step 2: iterate the vertices in reversed topological order, and update the minimum index.

A1 求出最大生成树（边权取反或者边从大到小排序后贪心），然后不在最大生成树里面的边就是答案。时间复杂度$O(E \log E)$.

A2 每次取左端点最左边的所有区间中，右端点最靠右的一个加入集合，然后将剩下所有区间减掉与这个区间的交，在对子问题做递归的求解。

B 建图，每个点表示一个人，两个点有边当且仅当两个人认识。每个点计算一个度数（认识的人数目）和反向图的度数（不认识的人的数目）。每次找一个认识的人小于6或者不认识的人小于6的点删掉，并更新所有点的认识的人数目和不认识的人数目，直到无点可删。复杂度$O(n^2)$。我好像在改作业的时候把复杂度写n^2的给扣了分，现在想想好像不太有更快的做法。分数已经改回。

Sample Solution for Problem Set 11

Data Structures and Algorithms, Fall 2019

December 29, 2019

Additional Problem 2 We prove that, for an even number n, the best strategy is to first produce $n/2$, then multiply it by 2, and for an odd n, the best strategy is to first produce $n-1$, then add 1 to it, by induction.

For the base case, $n = 1$ and $n = 2$, this is trivial. Suppose all integers strictly smaller than n satisfy the induction hypothesis. Then for n:

- If n is odd, we can only produce n by adding 1 to $n-1$, so this is trivial.
- If n is even, for the sake of contradiction, suppose the best strategy is to first produce $n-1$, using T steps, then add 1 to $n-1$. So the best strategy uses $T+1$ steps to produce n. Note that since $n-1$ is odd, by the IH, we can use $T-1$ steps to produce $n-2$. And since $n-2$ is even, again by the IH, we can use $T-2$ steps to produce $(n-2)/2 = n/2-1$. So by adding 1 to $n/2-1$, we can use $T-1$ steps to produce $n/2$, and then we can produce n using T steps, a contradiction.

Additional Problem 3 See Figure 1 for the pseudocode.

```
Ad2($v$)
1: Compute SSSP for vertices $s$ and $t$
2: Let $d_s(u)$ be minimum distance from $s$ to $u$, and $d_t(u)$ be minimum distance from $t$ to $u$
3: $d_0 \leftarrow d_s(t)$
4: $d_1 \leftarrow d_s(t)$
5: for $e' \in E'$ do
6:     $e' = (u, v)$
7:     $d' \leftarrow d_s(u) + d_s(t)$
8:     $d_1 = \min(d_1, d')$
9:     $d' \leftarrow d_s(v) + d_s(u)$
10:    $d_1 = \min(d_1, d')$
11: return $d_0 - d_1$
```

Figure 1

Additional Problem 4 -
a. Construct corresponding graph with distance $-\log r_{ij}$.
b. Detect negative cycles in the graph above.

Sample Solution for Problem Set 12

Data Structures and Algorithms, Fall 2019

2019 年 12 月 29 日

A1 (1)用原图用Bellman-Ford后，求出所有还能进行松弛的点集记为S_1，对原图的反向图用Bellman-Ford后求出所有还能进行松弛的点集记为S_2，将$S_1 \cap S_2$中的点标记为S。每次对一个点进行dijstra的时候，先判断这个点v能到达S中的哪些点，将这些点的距离设置为$d = -\infty$和原点$v.d = 0$一起加入队列中，再进行dijstra相同的操作（每次从优先队列中取d最小的拿出来松弛）。

（2）（3）：求出最终矩阵D后，多加一步，用D乘以边权矩阵W（也就是对所有点再进行一次松弛），若仍然存在$d(u,k) + w_{k,v} \le d(u,v)$，则设置$d(u,v) = -\infty$。

A2 (1)举反例，比如1,6,10三种面值的货币，换12元。

（3）用$T[i]$表示利用$c[1...n]$这些货币换钱用的最少的数量。初始设$T[0] = 1$。递推方程为

$$T[i] = \min_{1 \le i \le n, i \ge c[i]} (T[i - c[i]] + 1)$$

A3 设$T[u]$表示以u为根的子树中，root上的信息broadcast到所有的节点需要的最小轮数。记u的所有孩子按$T[v]$降序排序为$v_1, v_2, ..., v_k$，则递归方程为

$$T[u] = \min_{1 \le i \le k} (T[v_i] + i)$$

复杂度$O(|V|)$。

A4 方法一：由于不能重叠，最长正反出现的子串必然有一个对称轴。枚举对称轴算以之为对称轴的最长正反出现的子串，取一个最大值。

对称轴有两种可能，一种是以字符$T[i]$为对称轴，此时我们需要找到$T[i-k] = T[i+k], T[i-k-1] = T[i+k+1], ..., T[i-k-l] = T[i+k+l]$，即长度为$l+1$。我们只需要比较每一个$T[i-j]$和$T[i+j]$是否相等，找出最长连续的一段相等的即可。

第二章对称轴在i与$i+1$之间，此时我们应当比较的是$T[i+1-j$和$T[i+j]$是否相等，找最长的一段。复杂度$O(n*n) = O(n^2)$。

方法二：此题本意大概是用动态规划，记$A[i,j]$表示在$T[i...j]$中正反都出现且不重叠的最长子串长度。下面考虑怎么递归计算$A[i,j]$：

$i == j$时为0。$i \ne j$时，首先$A[i,j]$必然不会小于$A[i+1,j]$和$A[i,j-1]$，所以先取这两个值的max。此时我们已经考虑了所有在$T[i+1,...,j]$和$T[i,...,j-1]$中正反出现且不相交的所

有子串的情况。唯一可能漏掉的情况是：正向出现的子串以i开头，反向出现的子串以j开头。记$B[i,j]$表示正向出现的子串以i开头且反向出现的子串以j开头的最长子串，递归方程为

$$A[i,j] = \max(A[i+1,j], A[i,j-1], B[i,j])$$

对$B[i,j]$进行预处理计算，同样用递归，$T[i] = T[j]$时$B[i,j] = B[i+1,j-1]+1$，否则等于0。

两个递归复杂度都是$O(n^2)$。

还有其他的方法，比如逆序字符串然后找不重叠的最长公共子串，只是需要注意考虑“不重叠”这个条件。

Sample Solution for Problem Set 13

Data Structures and Algorithms, Fall 2019

December 29, 2019

AP 1 **b.** Use $M[i,j]$ to denote the maximum value we can achieve in the range $[i..j]$.
Use $m[i,j]$ to denote the minimum value we can achieve in the range $[i..j]$.

$$M[i,j] = \begin{cases} A[i] & \text{if } i = j \\ \max(S_{ij}) & \text{o.w.} \end{cases}$$

$$m[i,j] = \begin{cases} A[i] & \text{if } i = j \\ \min(S_{ij}) & \text{o.w.} \end{cases}$$

where

$$S_{ij} = \bigcup_{i \le k < j} \{M[i,k] \operatorname{op}(k) M[k{+}1,j], m[i,k] \operatorname{op}(k) m[k{+}1,j], m[i,k] \operatorname{op}(k) M[k{+}1,j], M[i,k] \operatorname{op}(k) m[k{+}1,j]\}$$

The answer is then $M[1,n]$. Time complexity: $O(n^3)$.