

# CLRS Exercise

Tongda Xu

December 12, 2018

- 1 The Role of Algorithms in Computing
- 2 Getting Started
- 3 Growth of Functions
- 4 Divide-and-Conquer
- 5 Probabilistic Analysis and Randomized Algorithms
- 6 Heapsort
- 7 Quicksort

## 7.1 7.3

### 7.1.1 a

This is certain concerning the *Randomized* procedure, the probability of any index  $i$  is chosen from  $[0, n - 1]$  is:

$$\begin{aligned} Pr(\text{pivot} = i) &= \frac{1}{n} \\ E(X_i) &= 1 * Pr(\text{pivot} = i) + 0 * Pr(\text{pivot} \neq i) = \frac{1}{n} \end{aligned}$$

### 7.1.2 b

It is certain that if  $i$ th element is chosen as pivot, *Random-Partition* cost  $\Theta(n)$  time, and it will call *QuickSort* $[1, q - 1]$ , *QuickSort* $[q + 1, n]$  recursively.

Concerning only the first *Partition*, this would be the result:

$$\begin{aligned} E(T(n)) &= \sum_{i=1}^n Pr(\text{pivot} = i)(T(i - 1) + T(n - i) + \Theta(n)) \\ &= \sum_{i=1}^n X_i(T(i - 1) + T(n - i) + \Theta(n)) \end{aligned}$$

### 7.1.3 c

Concerning  $X_i = \frac{1}{n}$

$$\begin{aligned} E(T(n)) &= \sum_{i=1}^n \frac{1}{n} (T(i-1) + T(n-i) + \Theta(n)) \\ &= \sum_{i=1}^n \frac{1}{n} T(i-1) + \sum_{i=1}^n \frac{1}{n} T(n-i) + \sum_{i=1}^n \frac{1}{n} \Theta(n) \\ &= \frac{2}{n} \sum_{i=1}^{n-1} T(i) + \Theta(n) \end{aligned}$$

### 7.1.4 d

$$\begin{aligned} &\sum_{k=2}^{n-1} k \lg k \\ &\leq \lg \frac{n}{2} \sum_{k=2}^{\frac{n}{2}} k + \lg n \sum_{k=\frac{n}{2}}^{n-1} k \\ &= \lg n \sum_{k=2}^{n-1} k - \lg 2 \sum_{k=2}^{\frac{n}{2}} k \\ &= \lg n \frac{(n+1)(n-2)}{2} - \frac{(\frac{n}{2}+2)(\frac{n}{2}-1)}{2} \\ &\leq \lg n \frac{n^2}{2} - \frac{n^2}{8} \end{aligned}$$

by Calculus, we have:

$$(\frac{1}{2}x^2 \lg x - \frac{1}{4}x^2)|_1^{n-1} \leq E(T(n)) \leq (\frac{1}{2}x^2 \lg x - \frac{1}{4}x^2)|_2^n$$

### 7.1.5 e

Proof of  $E(T(n)) = O(n \lg n)$ :

Assume that  $\forall k \in [1, n-1], \exists c, E(T(k)) \leq c k \lg k - \Theta(k)$

For  $k = n, E(T(n)) \leq \frac{n}{2} c (\lg n \frac{n^2}{2} - \frac{n^2}{4} - \Theta(n^2)) + \Theta(n) \leq c n \lg n - \Theta(n)$

Proof of  $E(T(n)) = \Omega(n \lg n)$ :

Assume that  $\forall k \in [1, n-1], \exists c, E(T(k)) \geq c k \lg k + \Theta(k)$

For  $k = n, E(T(n)) \geq \frac{n}{2} c (\lg n \frac{(n-1)^2}{2} - \frac{(n-1)^2}{4} + \Theta(n^2)) + \Theta(n) \geq c n \lg n + \Theta(n)$

$\rightarrow E(T(n)) = \Theta(n \lg n)$

## 7.2 7.5

### 7.2.1 a

From counting Theorem, it could be noticed that:

$$p_i = \frac{(i-1)(n-i)}{C_n^3} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$$

### 7.2.2 b

$$\begin{aligned} Pr(i = \text{medium})(\text{normal}) &= \frac{1}{n} \\ Pr(i = \text{medium})(3\text{part}) &= \frac{6(\frac{1}{2}n-1)(n-\frac{1}{2}n)}{n(n-1)(n-2)} = \frac{3}{2} \frac{1}{n} \\ Pr(3\text{part}) - Pr(\text{normal}) &= \frac{1}{2} \frac{1}{n} \end{aligned}$$

### 7.2.3 c

Consider  $f_{diff} = \int_{\frac{n}{3}}^{\frac{2}{3}n} (\frac{6(i-1)(n-i)}{n(n-1)(n-2)} - \frac{1}{n}) di$

$$= \frac{(-2i^3 + 3(n+1)i^2 - 6ni - (n-1)(n-2)i)|_{i=\frac{n}{3}}^{i=\frac{2}{3}n}}{n(n-1)(n-2)}$$

$$\lim_{n \rightarrow \infty} f_{diff} = \frac{4}{27}$$

#### 7.2.4 d

Consider we are so lucky that each partition we choose the median:

In the Iteration tree, we have:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(\frac{1}{2}n) + n & n > 1 \end{cases}$$

The  $\Omega(n \lg n)$  is kept even in best case.

## 8 Sorting in Linear Time

### 8.1 8.1-1

$n-1$  times, since we need  $n$  elements to formulate

### 8.2 8.1-2

$$\Sigma_1^n \lg k < \int_1^{n+1} \lg k dk = (k \lg k - k)_1^n = (n \lg n - n) - (0 - 1) = n \lg n - n + 1$$

### 8.3 8.1-3

$\leftrightarrow$  proof at least half of branch is longer than  $h$

Consider a decision tree with  $n!/2$  elements

$\leftrightarrow$  proof at least half of branch is longer than  $h$

Consider a decision tree with  $n!/n$  elements

$\leftrightarrow$  proof at least half of branch is longer than  $h$

Consider a decision tree with  $n!/2^n$  elements, this is not significant enough and could leave only  $\Omega(\lg \frac{n!}{2^n}) = \Omega(n \lg n - n) = \Omega(n \lg n)$  elements

### 8.4 8.2-4

Consider a trim version of counting sort, build the  $C$  map up and query directly:

COUNTING-SORT-TRIM( $A, k$ )

```

1   $C[]$ 
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] ++$ 
6  for  $m = 1$  to  $k$ 
7       $C[m] += C[m - 1]$ 
8  return  $C[m]$ 
```

```

DIRECT-QUERT( $A, k, a, b$ )
1   $C = \text{COUNTING-SORT-TRIM}(A, k)$ 
2  if  $a < 1$ 
3      return  $C[b]$ 
4  else return  $C[b] - C[a - 1]$ 

```

## 8.5 8.3-2

Heapsort is not stable

The scheme would be very similar to counting sort and takes  $\Theta(n)$  time

## 8.6 8.3-4

First, with  $O(n)$  time: convert  $n$  numbers  $k_{10}$  into  $k_n$  which has 3 digits.

Second, with  $O(d(n+n))$  time (*Lemma 8.3*): Radix sort  $n$  3-digit numbers with each digit take up to  $n$  possible values.

```

DIGITS CONVERT( $X$ )
1   $result[]$ 
2  for  $i = 2$  downto 0
3       $result[i] = X/n^i$ 
4       $X = X \bmod n^i$ 
5  return  $result$ 

```

```

SORT( $A, x$ )
1   $result[]$ 
2  for each  $S$  in  $A$ 
3       $S = \text{DIGITS CONVERT}(S)$ 
4   $\text{RADIX-SORT}(A, x)$ 

```

# 9 Medians and Order Statistics

## 9.1 9.2-1

once  $p == r$ , the function return and recursion end.

## 9.2 9.2-2

It is because  $\forall k, X_k = \frac{1}{n}$ , giving information on which  $k$  would not effect observation

### 9.3 9.2-3

RANDOMIZED-SELECT-ITER( $A, p, r, i$ )

```

1  while 1
2      if  $i == k$ 
3          return  $A[i]$ 
4      else
5           $q = \text{RANDOM-PARTITION}(A, p, r)$ 
6          if  $i < k$ 
7               $r = q - 1$ 
8          else  $p = q + 1, i = i - k$ 

```

### 9.4 9.2-4

The worst case is reverse side:

$pivot = 9, 8, 7, 6, 5, 4, 3, 2, 1, 0$

### 9.5 9.1

#### 9.5.1 a

Sorting: MERGE-SORT( $A$ ) in worst case  $O(n \lg n)$

Query: CALL-BY-RANK( $A, k$ )  $i$  times in worst case  $O(i)$ , here we assume manipulating  $O(n)$  space cost  $O(n)$  time.

#### 9.5.2 b

Building: BUILD-MAP-HEAP( $A$ ) in worst case  $O(n)$

Query: calling EXTRA-MAX( $A, k$ )  $i$  times in worst case  $O(i \lg n)$

#### 9.5.3 c

Selecting: SELECT( $A, i$ ) in worst case  $O(n)$

Sorting: MERGE-SORT( $A'$ ) in worst case  $O(i \lg i)$

### 9.6 9.2

#### 9.6.1 a

$$\sum_1^{k-1} w_i = \sum_1^{k-1} \frac{1}{n} = \frac{k-1}{n} < \frac{1}{2}$$

$$\sum_{k+1}^n = \frac{n-k}{n} \leq \frac{1}{2}$$

### 9.6.2 b

WEIGHT-MEDIAN( $A$ )

```
1  w[] = SORT( $A$ ).weight
2  n = w.length
3  for  $i = 1$  to  $n$ 
4       $w[i] = w[i] + w[i - 1]$ 
5  return FIND( $w[], \frac{1}{2}$ )
```

### 9.6.3 c

SUM( $w_1, w_i, lasti, lastsum$ )

```
1  if  $i > lasti$ 
2      return  $lastsum + \text{NORMAL-SUM}(w_{lasti, i})$ 
3  else return  $lastsum - \text{NORMAL-SUM}(w_{i, lasti})$ 
```

WEIGHT-MEDIAN-LINEAR( $A$ )

```
1  while 1
2      if  $\text{sum}[w_1, w_i, lasti, lastsum] < \frac{1}{2}, \text{sum}[w_1, w_{i+1}, lasti, lastsum] > \frac{1}{2}$ 
3          return  $i$ 
4      else
5           $lastsum = \text{sum}[w_1, w_i, lasti, lastsum], lasti = i$ 
6          if  $\text{sum}[w_1, w_i] < \frac{1}{2}$ 
7               $i = \text{MEDIAN}(A, i, r)$ 
8          else  $i = \text{MEDIAN}(A, p, i)$ 
```

We will experience  $\log n$  iteration, but the load is decreasing logarithmically, so the result is linear. Notice the sum is special here, calculating the difference only.

## 9.7 9.4

### 9.7.1 a

$k \leq i$  or  $k \geq j : 0$   
 $i < k < j : \frac{2}{j-i+i}$

**9.7.2 b**

**9.7.3 c**

**9.7.4 d**

## 10 Elementary Data Structures

### 11 Hash Tables

#### 11.1 11.1-2

Consider *vector*  $\langle \text{bool} \rangle A$ ,  $A.size() = m$ , just store the bool value of  $key = m$  exist or not.

SEARCH( $A, key$ )

```
1  if  $A[key]$ 
2      return  $key$ 
3  else return  $NIL$ 
```

INSERT( $A, key$ )

```
1   $A[key] = 1$ 
```

DELETE( $A, key$ )

```
1   $A[key] = 0$ 
```

#### 11.2 11.2

##### 11.2.1 a

Consider for a ball i fall into a specific bucket  $Pr(i) = \frac{1}{n}$   
Then consider Binomial Distribution,  $Pr(k) = C_n^k Pr(i)^k (1 - Pr(i))^{n-k}$

##### 11.2.2 b

Consider random picking a slot, the probability of that slot is maximum is  $Pr_{max} = \frac{1}{n}$ , and it contains k elements  $Q_k$ . for conditional probability, we have:

$$P_k = Pr_{i=k|max} = \frac{Pr(i=k \cap max)}{Pr_{max}} \leq \frac{Pr(i=k)}{Pr_{max}} = nQ_k$$

##### 11.2.3 c

Proof:

$$\begin{aligned} Q_k &= \left(\frac{1}{n}\right)^k \left(\frac{n-1}{n}\right)^{n-k} C_n^k \\ &= \frac{(n-1)^{n-k}}{n^n} \frac{n!}{k!(n-k)!} \\ &\leq \frac{n^n}{n^n} \frac{1}{k!} \end{aligned}$$

$$\begin{aligned}
&= \frac{e^k}{k^k} \frac{1}{k^{\frac{1}{2}} (1 + \Theta(\frac{1}{n}))} \\
&\leq \frac{e^k}{k^k}
\end{aligned}$$

#### 11.2.4 d

Proof for  $Q_{k_0}$ :

$$\begin{aligned}
Q_{k_0} &= \frac{e^{(\frac{clgn}{lglg n})}}{(\frac{clgn}{lglg n})^{\frac{clgn}{lglg n}}} \\
&= \frac{n^{\frac{clg \frac{c}{c}}{lglg n}}}{\frac{n^c}{\frac{clglglgn}{n}}} = n^{\frac{clg \frac{c}{c} + clglglgn}{lglg n} - c}
\end{aligned}$$

It would not take effort to notice that since  $\lim_{n \rightarrow \infty} \frac{clg \frac{c}{c} + clglglgn}{lglg n} = 0$

$\forall c > 3 + \epsilon, Q_{k_0} = O(\frac{1}{n^3})$

And  $P_k \leq nQ_k \rightarrow P_k = O(\frac{1}{n^2})$

#### 11.2.5 e

$$E(M) = \sum_{M=1}^n MPr(M) < nPr(M > \frac{clgn}{lglg n}) + \frac{clgn}{lglg n} Pr(M \leq \frac{clgn}{lglg n})$$

A stronger conclusion to note:

$$\begin{aligned}
E(M) &= \sum_{M=1}^n MPr(M) < MPr(M > \frac{clgn}{lglg n}) + \frac{clgn}{lglg n} Pr(M \leq \frac{clgn}{lglg n}) \\
&\leq \int_{\frac{clgn}{lglg n}}^{\infty} \frac{1}{n} dn + 1 * \frac{clgn}{lglg n} \\
&= lg(\frac{clgn}{lglg n}) + \frac{clgn}{lglg n} \\
&= O(\frac{clgn}{lglg n})
\end{aligned}$$

## 12 Binary Search Trees

## 13 Red-Black Trees

## 14 Augmenting Data Structures

## 15 Dynamic Programming

### 15.1 15.1-1

$$2^n - 1 = \sum_{j=0}^{n-1} 2^j$$

### 15.2 15.1-2

Do not know how!



### 15.3 15.1-3

BOTTOM-UP-CUT-ROD( $p, n, c$ )

```
1   $r[] = c$ 
2  for  $j = 1$  to  $n$ 
3      for  $i = 1$  to  $j$ 
4           $r[i] \leftarrow \max(p[i] + r[j - i] - c)$ 
5  return  $r[n]$ 
```

### 15.4 15.1-4

MEMOIZED-CUT-ROD( $p, n, m, s$ )

```
1  if  $m[n] > -1$ 
2      return  $m[n]$ 
3  else
4      for  $i \leftarrow 1$  to  $n$ 
5           $m[n] \leftarrow \max(p[i] + r[n - i])$ 
6           $s[n] \leftarrow i$ 
7      return  $m[n]$ 
```

### 15.5 15.1-5

See Code

### 15.6 15.2-1

See Code

### 15.7 15.2-2

See Code

### 15.8 15.2-3

Assume that  $\forall k \leq n - 1, T(k) \geq c2^k$

Then  $T(n) = \sum_{k=1}^{n-1} T(k)T(n-k) = (n-1)c^22^n > c2^n$

So  $T(n) = \Omega(n), \omega(n)$

### 15.9 15.2-4

See Figure 1

### 15.10 15.2-5

For each level  $h(i) = i(n-i)$

For tree  $T(n) = 2\sum_{i=1}^{n-1} i(n-i)$

Ex 15.2.4

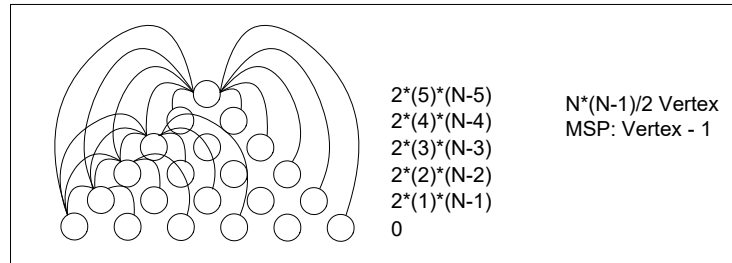


Figure 1: 15.2-4

$$\begin{aligned}
 &= \frac{3n^3 + 3n^2}{3} - \frac{2n^3 + 3n^2 + n}{3} \\
 &= \frac{n^3 - n}{3}
 \end{aligned}$$

### 15.11 15.2-6

Assume that  $\forall k \leq n-1, N(k) = k-1$

Then  $N(n) = N(n-1) + 1$

So  $N(n) = n-1$

### 15.12 15.3-1

running through:  $T(n) = n * P_n^n = n * n! > 4^n$

running recursion:  $T(n) = 2 \sum_{i=1}^{n-1} 4^i + n = \frac{8}{3} 4^{n-1} + n \leq 4^n$

running through takes longer

### 15.13 15.3-2

no overlapping subproblem call

### 15.14 15.3-3

Yes

### 15.15 15.3-4

Do not know how!

### 15.16 15.4-1

See code

### 15.17 15.4-2

See code

### 15.18 15.4-3

LCS( $X, Y$ )

```
1  $DP \leftarrow []$ 
2 return LSC-AID( $X.length, Y.length$ )
```

LSC-AID( $i, j$ )

```
1 if  $i = 0$  or  $j = 0$ 
2      $DP[i][j] \leftarrow 0$ 
3 else
4     if  $X[i] = Y[j]$ 
5         if  $DP[i-1][j-1] = NIL$ 
6              $DP[i-1][j-1] = \text{LSC-AID}(i-1, j-1)$ 
7          $DP[i][j] \leftarrow DP[i-1][j-1] + 1$ 
8     else
9         if  $DP[i-1][j] = NIL$ 
10             $DP[i-1][j] = \text{LSC-AID}(i-1, j)$ 
11        if  $DP[i][j-1] = NIL$ 
12             $DP[i][j-1] = \text{LSC-AID}(i, j-1)$ 
13         $DP[i][j] = \max\{DP[i][j-1], DP[i-1][j]\}$ 
14 return  $DP[i][j]$ 
```

### 15.19 15.4-5

This is easy to construct from bottom to top, and straightforward to see a time complexity of  $\Theta(n^2)$ :

LONGEST-MONO-INCREASE( $s$ )

```
1  $DP \leftarrow []$ 
2  $DP[1] \leftarrow s[1]$ 
3 for  $i \leftarrow 1$  to  $n$ 
4     for  $j \leftarrow i-1$  downto 1
5         if  $DP[j].end < s[i]$ 
6              $DP[i] \leftarrow DP[i].length < DP[j].length + 1 ? DP[j] + s[i] : DP[i]$ 
7         else  $DP[i] \leftarrow DP[i].length < DP[j].length ? DP[j] : DP[i]$ 
8 return  $DP[s.length]$ 
```

### 15.20 15.5-1

A Preorder Traverse of BST

```

PRE-ORDER-PRINT-AID( $i, j, root$ )
1  if  $root[i, j] - 1 - i \geq 0$ 
2       $k\ root[i, root[i, j] - 1]$  is the left child of  $k\ root[i, j]$ 
3      PRE-ORDER-PRINT-AID( $i, root - 1, root[i, root[i, j] - 1]$ )
4  else  $d\ i - 1$  is the left child of  $k\ root[i, j]$ 
5  if  $j - root[i, j] - 1 \geq 0$ 
6       $k\ root[root[i, j] + 1, j]$  is the right child of  $k\ root$ 
7      PRE-ORDER-PRINT-AID( $root + 1, j, root[root[i, j] + 1, j]$ )
8  else  $d\ i - 1$  is the right child of  $root$ 

PRE-ORDER-PRINT( $root$ )
1   $k\ root[1, n]$  is the root
2  PRE-ORDER-PRINT-AID( $1, n, root$ )

```

### 15.21 15.5-3

Asymptotically there would be no change to the running time, just the constant  $cn^3$  increase

Time spent on  $w$  would increase from  $\Theta(n^2)$  to  $\Theta(n^3)$

### 15.22 15.1

It is easy to implement a memorized recursive algorithm, but very hard to build from down to top:

```

LONGEST-SIMPLE-PATH( $s, t$ )
1   $DP[] \leftarrow -1$ 
2  return LONGEST-SIMPLE-PATH-AID( $s, t$ )

LONGEST-SIMPLE-PATH-AID( $s, t$ )
1  if  $s \neq t$ 
2      if  $DP[s] = -1$ 
3           $DP[s] \leftarrow \max_{v \in s.adjList} \{WEIGHT(s, v) + LONGEST-SIMPLE-PATH-AID(v, t)\}$ 
4      return  $DP[s]$ 
5  else return 0

```

the  $DP[s]$  is a array with length  $V$ , all overlapping subproblem is solved by memory, so  $DP[s]$  cost  $\Theta(V)$  time to construct. In each query, it cost  $s.adjList.length()$  time, and in total it cost  $O(E)$  time. So Longest-simple-path cost  $O(E + V)$  time to compute.

## 16 Greedy Algorithms

### 16.1 16.1-1

This process fill a grid of  $\frac{1}{2}n^2$  and take space and time of  $\Theta(n^2)$ . Greedy is one-pass and take only  $\Theta(n)$ .

```

AS-ADI(a)
1  DP = []
2  return AS-ADI(0, a.length)

AS-ADI(i, j)
1  for m ← j − 1 downto i + 1
2      if a[m].f ≤ a[j].s and a[m].s ≥ a[i].f
3          S[i][j].push(a[m])
4  if S[i][j] = ∅
5      DP[i][j] ← 0
6  else
7      if DP[i][j] = NIL
8          DP[i][j] ← maxa[k] ∈ S[i][j] {AS-ADI(i, k) + 1 + AS-ADI(k, j)}
9  return DP[i][j]

```

It is easy to find that as we remove an edge from adjacent list once we find it, and we traverse every edge, the time complexity would be  $\Theta(E)$

## 17 Amortized Analysis

### 17.1 17.1-1

No, the sequence could produce  $\frac{1}{2}nk$  push and take  $\Theta(nk)$  time

### 17.2 17.1-2

Consider we shift between  $2^{k-1}$  and  $2^{k-1}-1$ , which means 100000000 to 0111111111  $\frac{1}{2}n$  times, would cost  $\Theta(nk)$  time.

### 17.3 17.1-3

$$\Theta(i) = i - lgi + \frac{2^{lgi}-1}{2-1} = 2i - ogi = i$$

### 17.4 17.2-1

Assign *push* ← 2 and *pop* ← 1

**17.5 17.2-2**

**18 B-Trees**

**19 Fibonacci Heaps**

**20 van Emde Boas Trees**

**21 Data Structures for Disjoint Sets**

**22 Elementary Graph Algorithms**

**22.1 22.1-1**

for both out-degree and in-degree  $\Theta(V + E)$  time  
both take  $\Theta(V)$  memory

**22.2 22.1-2**

$1 \rightarrow 2 \rightarrow 3 \rightarrow NIL$   
 $2 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow NIL$   
 $3 \rightarrow 1 \rightarrow 6 \rightarrow 7 \rightarrow NIL$   
 $4 \rightarrow 2 \rightarrow NIL$   
 $5 \rightarrow 2 \rightarrow NIL$   
 $6 \rightarrow 3 \rightarrow NIL$   
 $7 \rightarrow 3 \rightarrow NIL$   
 $0 - 1 - 1 - 0 - 0 - 0 - 0$   
 $1 - 0 - 0 - 1 - 1 - 0 - 0$   
 $1 - 0 - 0 - 0 - 0 - 1 - 1$   
 $0 - 1 - 0 - 0 - 0 - 0 - 0$   
 $0 - 1 - 0 - 0 - 0 - 0 - 0$   
 $0 - 0 - 1 - 0 - 0 - 0 - 0$   
 $0 - 0 - 1 - 0 - 0 - 0 - 0$

**22.3 22.1-3**

TRANSPOSE(*Adjlist*)

```
1  new AdjlistPrime
2  for each node in Adjlist
3      for each subnode in Adjlist(node)
4          AdjlistPrime(subnode).insert(node)
5      Adjlist = AdjlistPrime
```

For adjacent list: just traverse every node and rebuild one  
 $\Theta(E + V)$  for time and space complexity, hard to do it inplace

```

TRANSPOSE(Adjmatrix)
1  for each pair(i, j) in upper left Adjmatrix
2      SWAP(Adjmatrix[i, j], Adjmatrix[j, i])

```

For adjacent matrix: just transpose the matrix  
 $\Theta(V^2)$  for time and  $\Theta(1)$  for space

## 22.4 22.1-4

use an adjacent matrix as aid.

## 22.5 22.1-5

For adjacent list, it is hard. We should regard it as a BREADTH-FIRST-SEARCH( $G$ )  
end at  $d = 2$ :

```

SQUARE( $G$ )
1  for each  $u$  in  $G.vertices$ 
2       $G.reset()$ 
3       $list = \emptyset$ 
4       $u.adjlist' = \text{BFS-AID}(G, u, list, 0)$ 

```

```

BFS-AID( $G, u, list, dist$ )
1  for each  $v$  in  $u.adjlist$ 
2      if  $v.color = white$  and  $dist \leq 2$ 
3           $list.insert(u)$ 
4      BFS-AID( $G, v, list, dist + 1$ ) =
5  return  $list$ 

```

This could cost  $\Theta(V^2 + VE)$  time and  $\Theta(V + E)$  space (if optimized).

For adjacent matrix, the square process would be simple. for each index  $m$  of matrix row, if matrix[ $m$ ][ $n$ ] exist, calculate bool union of matrix[ $m$ ] and matrix[ $n$ ]:

```

SQUARE( $G$ )
1  for each  $m$  in  $G.adjMatrix$ 
2      for each  $n$   $G.adjMatrix[m]$ 
3          if  $G.adjMatrix[m][n] == 1$ 
4               $G'.adjMatrix[m] = \text{AND}(G.adjMatrix[m], G.adjMatrix[n])$ 
5  return  $G'$ 

```

The SQUARE( $G$ ) cost  $\Theta(V^3)$  time and  $\Theta(V)$  space (if optimize)

## 22.6 22.2-3

line 2  $\rightarrow u.ifgrey = 0$   
line 5  $\rightarrow s.ifgrey = 1$   
line 14  $\rightarrow v.ifgrey = 1$

## 22.7 22.2-4

take  $\Theta(V^2)$  time and  $\Theta(V^2)$  space, since we need to search every column to find adjacent list.

line 12  $\rightarrow$  **for** : *each*  $v \in M[u]$   
line 13  $\rightarrow$  **if** :  $v == \mathbf{true}$  **and**  $v.color == white$

## 22.8 22.2-5

```
SQUARE(AdjList)
1  for each  $u$  in vertices
2      for each  $v$  in AdjList( $u$ )
3          AdjList( $u$ ).append(AdjList( $v$ ))
```

For adjacent list, for each vertex  $u$ , append the adjacent list of each adjacent vertex  $v$  to adjacent list of  $u$ .

*line3* would be execute  $\Theta(E)$  times in total,

```
SQUARE(Adjmatrix)
1  for each pair( $i, j$ ) in upper left Adjmatrix
2      SWAP(Adjmatrix[ $i, j$ ], Adjmatrix[ $j, i$ ])
```

## 22.9 Edge traverse of undirected graph

According to *Theorem22.10*, all edges are either tree edge or back edge. Modify the DFS-VISIT( $G, u$ ), add a PRINT-PATH( $G, u$ ) would do it. Assume a *root* =  $u$  is selected:

```
DFS-VISIT( $G, u$ )
1   $u.color = grey$ 
2  dict[(Vertex, Vertex), edgeType] =  $\emptyset$ 
3  for each  $v$  in  $u.adjList$ 
4      if  $v.color == white$ 
5           $dict(u, v) = treeEdge$ 
6          DFS-VISIT( $G, v$ )
7      else  $dict(u, v) = backEdge$ 
8  PRINT-PATH( $G, u$ )
```



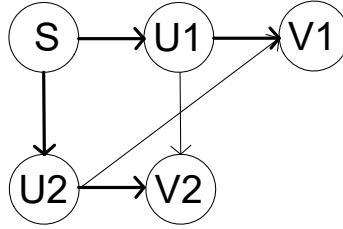


Figure 2: 22.2-6

```

PRINT-PATH( $G, u$ )
1  PRINT ("u")
2  for each  $v$  in  $u.adjList$ 
3      if  $(u, v) == treeedge$ 
4          PRINT (" → ")
5          PRINT-PATH( $G, v$ )
6      else PRINT (" → v")
  
```

line 4,6 cost same level of time as the comparison in line 3, would not change the  $\Theta(V + E)$  time complexity of  $DFS(G)$   
 the print path function as:  
 This procedure cost  $\Theta(V + E)$  as well

## 22.10 22.2-6

Consider the following condition in Figure 2:

$E_\pi = \langle s, u1 \rangle, \langle u1, v1 \rangle, \langle s, u2 \rangle, \langle u2, v2 \rangle$

In BFS Tree,  $\delta(s, v1), \delta(s, v2)$  is either  $\langle s, u1, v1 \rangle, \langle s, u1, v2 \rangle$  or  $\langle s, u2, v1 \rangle, \langle s, u2, v2 \rangle$

## 22.11 22-3.12

Tweak the  $DFS-VISIT(G, u)$  and  $DFS(G)$  would be enough:

```

DFS( $G$ )
1  for each  $u$  in  $G.V$ 
2       $u.color = white$ 
3   $c = 1$ 
4  for each  $u$  in  $G.V$ 
5      if  $u.color = white$ 
6           $DFS-VISIT(G, u, c)$ 
7           $c++$ 
  
```

```

DFS-VISIT( $G, u, c$ )
1   $u.color = grey$ 
2   $u.cc = c$ 
3  for each  $v$  in  $u.adjList$ 
4      if  $v.color == white$ 
5          DFS-VISIT( $G, v$ )

```

DFS( $G$ ) could be tweaked to do it as well

## 22.12 22.4-1

$p[27 : 28] \rightarrow n[21 : 26] \rightarrow o[22 : 25] \rightarrow s[23 : 24] \rightarrow$   
 $m[1 : 20] \rightarrow r[6 : 19] \rightarrow y[9 : 18] \rightarrow v[10 : 17] \rightarrow x[15 : 16] \rightarrow$   
 $w[11 : 14] \rightarrow z[12 : 13] \rightarrow u[7 : 8] \rightarrow q[2 : 5] \rightarrow t[3 : 4]$

## 22.13 22.4-3

A DFS( $G$ )/BFS( $G$ ) returns false when a back edge is found, easy to proof it is  $\Theta(V)$

## 22.14 22.1

### 22.14.1 a-1

Suppose  $(v, u)$  is a backedge.  $u$  is ancestor elder than parent of  $v$ . This means  $(s, u) + \text{forwardEdge}$  is shorter than  $(s, v)$  produced by BFS which is  $\delta(s, v)$  by **Theorem 22.5**. Same reason for forward edge.

### 22.14.2 a-2

By **Theorem 22.5**  $\delta(s, v) = \delta(s, v.parent) + (v.parent, v) = \delta(s, u) + (u, v) \rightarrow v.d = u.d + 1$

### 22.14.3 a-3

$v.d \leq u.d + 1$  : Same as a-1, if  $v.d > u.d + 1$ ,  $\delta(s, v) = (s, u) + \text{cross}$  instead of  $(s, v)$ .

$v.d \geq u.d$ : If  $v.d < u.d$ ,  $(v, u)$  should be find out first, since this is undirected graph.

### 22.14.4 b-1

Same as a-1, the  $(s, u) + \text{backEdge}$  would be shorter than  $(s, v)$

### 22.14.5 b-2

Same as a-2, By **Theorem 22.5**  $\delta(s, v) = \delta(s, v.parent) + (v.parent, v) = \delta(s, u) + (u, v) \rightarrow v.d = u.d + 1$

### 22.14.6 b-3

Only the first half of a-3. if  $v.d > u.d + 1$ ,  $\delta(s, v) = (s, u) + cross$  instead of  $(s, v)$ .

### 22.14.7 b-4

By **Corollary 22.4** and By **Theorem 22.5**, we know that if  $v$  is an ancestor of  $u$   $\delta(s, u) = \delta(s, v) + k \rightarrow \delta(s, u) > \delta(s, v) \rightarrow u.d > v.d$ , I did not see how  $u.d = v.d$  but the statement is correct.

## 22.15 22.3

### 22.15.1 1. proof

Euler tour exist  $\rightarrow$  in-degree == out-degree: Suppose the cycle through  $i$  vertex  $n$  times would be  $E - cycle = \{v_i, v_j, v_k, \dots, v_i\}$ . The in-degree of  $v_j$  would be the time of  $v_j$  appears with element in front, and out-degree of  $v_j$  would be the time of  $v_j$  appears with element in the back. If  $v_j$  is not head or tail, this is obvious that every time  $v_j$  appear, there is element in front and tail. If  $v_j$  is head, it must also be tail, which balance the in-degree and out-degree again.

in-degree == out-degree  $\rightarrow$  Euler tour exist

### 22.15.2 2. implement

This is very similar to SCC, we find closed cycle first then join them with other edge set. This procedure would return a cycle, which is a list of vertex. closed cycle has  $cycle.begin() = cycle.end()$ , open cycle(path, not a cycle) do not has it. But if Euler tour exist, open cycle would join close cycle into a big cycle.

CIRCLEFIND( $cycle, u, v$ )

```

1  ClosedCycleSet, OpenCycleSet =  $\emptyset$ 
2  while alladjList! =  $\emptyset$ 
3      for  $v$  in Vertex with adjList! =  $\emptyset$ 
4          if  $v.adjList!$  =  $\emptyset$ 
5              new cycle =  $\emptyset$ 
6              CIRCLEFINDAID( $cycle, u, NIL$ )
7              if  $cycle.type == closed$ 
8                  ClosedCycleSet.push( $cycle$ )
9              else OpenCycleSet.push( $cycle$ )
```

```

CIRCLEFINDAID(cycle, u, v)
1  cycle.insert(u)
2  if v! = NIL
3      v.adjList.erase(u)
4  if u == NIL
5      cycle.type = open
6      return
7  elseif u == cycle.start
8      cycle.type = close
9      return
10 else v = u
11     u = u.adjList.begin()
12     CIRCLEFIND(CYCLE, u, v)

```

## 23 Minimum Spanning Trees

## 24 Single-Source Shortest Paths

### 25 Exercise 24.1-2

Proof:

If the shortest path exist  $\rightarrow w(p) = \delta(s, v) = v.d$  after BELLMAN-FORD, then  $v.d = \delta(s, v) \neq \infty$

If BELLMAN-FORD terminates with  $v.d < \infty$ , then Lemma 24.11  $\delta(s, v) \leq v.d$ , the shortest path exist.

### 26 Exercise 24.1-3

Add a bool indicator monitoring if a real relax is conducted. If not then break the for loop. If the time that real relax happens is more than vertex number - 1, which means when entering the loop, the counter is larger than G.V, then return false, since there must be a negative weight loop:

```

RELAX-FASTER(u, v,  $\omega$ , relaxed)
1  if v.d > u.d +  $\omega(u, v)$ 
2      v.d  $\leftarrow$  u.d +  $\omega(u, v)$ 
3      v. $\pi$   $\leftarrow$  u
4      relaxed  $\leftarrow$  true

```

```

BELLMAN-FORD-FASTER( $G, \omega, root$ )
1  INITIALIZE-SINGLE-SOURCE( $G, root$ )
2   $relaxed \leftarrow true$ 
3   $counter \leftarrow 1$ 
4  while  $relaxed$ 
5       $relaxed \leftarrow false$ 
6      if  $counter > |G.V|$ 
7          return false
8      for each edge  $(u,v) \in G.E$ 
9          RELAX-FASTER( $u, v, \omega, relaxed$ )
10      $counter \leftarrow counter + 1$ 
11 return true

```

## 27 Exercise 24.1-4

This is a simple one:

```

BELLMAN-FORD-ALL( $G, \omega, root$ )
1  INITIALIZE-SINGLE-SOURCE( $G, root$ )
2  for  $i \leftarrow 1$  to  $|G.V| - 1$ 
3      for each edge  $(u,v) \in G.E$ 
4          RELAX( $u, v, \omega$ )
5  for each edge  $(u,v) \in G.E$ 
6      if  $v.d > u.d + \omega(u, v)$ 
7           $v.d \leftarrow -\infty$ 

```

## 28 Exercise 24.2-2

A simple explanation would be since the last vertex of a DAG has no edge pointing right, without any edge to relax. The algorithm would be correct with vertex  $v_{k-1}$  visited, which relax edge  $(v_{k-1}, v_k)$  already.

## 29 Exercise 24.3-2

The Dijkstra's algorithm works since we know each node  $v$  when popped up, the  $v.d$  is in non-decreasing order promised by the non-negativity of edge. Without it, we could not promise  $\omega(x, y)$  is non-negative, which means although  $x$  is on top of queue,  $y.d$  could be smaller than when  $x.d$  is popped, which ruins the proof.

Or as the proof on book, when  $\omega(y, u)$  is smaller than 1, we could not promise  $\delta(s, y) \leq \delta(s, u)$ .

Consider:  $Vertex = \{x, y, z\}, Edge = \{(x, y) = 4, (x, z) = 3, (y, z) = -10\}$

### 30 Exercise 24.3-6

Consider the modification of Dijkstra's algorithm, growing the reliable tree from one root:

INITIALIZE( $G, root$ )

```
1  for each  $v \in G.V$ 
2       $v.d = 0$ 
3       $v.\pi = nullptr$ 
4   $root.d = 1$ 
```

RELAX( $u, v, r$ )

```
1  if  $u.d < v.d * r(u, v)$ 
2       $u.d \leftarrow v.d * r(u, v)$ 
3       $v.\pi \leftarrow u$ 
```

RELIABLE-PATH-SEARCH( $G, root$ )

```
1  INITIALIZE( $G, root$ )
2   $Q \leftarrow G.V$ 
3  while  $!Q.empty$ 
4       $u = \text{EXTRACT-MAX}(Q)$ 
5      for each  $v \in G.E(u)$ 
6          RELAX( $u, v, G.r$ )
```

PRINT-PATH( $G, root, v$ )

```
1  if  $v = root$ 
2      return
3  else
4      if  $v.\pi = nullptr$ 
5          PRINT(NO SUCH PATH)
6      else
7          PRINT-PATH( $G, r, v.\pi$ )
8          PRINT( $v$ )
```

### 31 All-Pairs Shortest Paths

### 32 Question 6: CLRS Exercise 25.2-2

Consider the following dynamic programming function:

$$t_{ij}^m = \begin{cases} i = j & m = 0 \\ \bigcup_{1 \leq k \leq n} (t_{ik}^{m-1} \cap (w_{kj} \neq \infty)) & 1 \leq m \leq n-1 \end{cases}$$

TRANSITIVE-CLOSURE-BRUTAL-FORCE( $G$ )

```

1   $n = G.V$ 
2   $l[n][n][n] \leftarrow 0$ 
3  for each entry in  $l[0]$ 
4       $l[0][i][j] \leftarrow i = j$ 
5  for  $m \leftarrow 1$  to  $n - 1$ 
6      for  $i \leftarrow 1$  to  $n$ 
7          for  $j \leftarrow 1$  to  $n$ 
8               $l[m][i][j] = \bigcup_{1 \leq k \leq n} (l[m-1][i][k] \cap (G.w_{kj} \neq \infty))$ 

```