CLRS Exercise

Tongda Xu

December 16, 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:
$Pr(pivot = i) = \frac{1}{n}$
$E(X_i) = 1 * Pr(pivot = i) + 0 * Pr(pivot \neq i) = \frac{1}{n}$

7.1.2 b

It is certain that if ith element is chosen as pivot, $Random-Parition$ cost $\Theta(n)$ time, and it will call $QuickSort[1, q-1], QuickSort[q+1, n]$ recursively.
Concerning only the first $Parition$, this would be the result:
$E(T(n)) = \Sigma_{i=1}^{n} Pr(pivot = i)(T(i-1) + T(n-i) + \Theta(n))$
$= \Sigma_{i=1}^{n} X_i (T(i-1) + T(n-i) + \Theta(n))$

7.1.3 c

Concerning $X_i = \frac{1}{n}$
$E(T(n)) = \Sigma_{i=1}^{n} \frac{1}{n}(T(i-1) + T(n-i) + \Theta(n))$
$= \Sigma_{i=1}^{n} \frac{1}{n} T(i-1) + \Sigma_{i=1}^{n} \frac{1}{n} T(n-i) + \Sigma_{i=1}^{n} \frac{1}{n} \Theta(n)$
$= \frac{2}{n} \Sigma_{i=1}^{n-1} T(i) + \Theta(n)$

7.1.4 d

$\Sigma_{k=2}^{n-1} klgk$
$\leq lg\frac{n}{2}\Sigma_{k=2}^{\frac{n}{2}} k + lgn\Sigma_{k=\frac{n}{2}}^{n-1} k$
$= lgn\Sigma_{k=2}^{n-1} k - lg2\Sigma_{k=2}^{\frac{n}{2}} k$
$= lgn\frac{(n+1)(n-2)}{2} - \frac{(\frac{n}{2}+2)(\frac{n}{2}-1)}{2}$
$\leq lgn\frac{n^2}{2} - \frac{n^2}{8}$
by Calculus, we have:
$(\frac{1}{2}x^2lgx - \frac{1}{4}x^2)|_1^{n-1} \leq E(T(n)) \leq (\frac{1}{2}x^2lgx - \frac{1}{4}x^2)|_2^{n}$

7.1.5 e

Proof of $E(T(n)) = O(nlgn)$:
Assume that $\forall k \in [1, n-1], \exists c, E(T(k)) \leq cklgk - \Theta(k)$
For $k = n, E(T(n)) \leq \frac{n}{2}c(lgn\frac{n^2}{2} - \frac{n^2}{4} - \Theta(n^2)) + \Theta(n) \leq cnlgn - \Theta(n)$
Proof of $E(T(n)) = \Omega(nlgn)$:
Assume that $\forall k \in [1, n-1], \exists c, E(T(k)) \geq cklgk + \Theta(k)$
For $k = n, E(T(n)) \geq \frac{n}{2}c(lgn\frac{(n-1)^2}{2} - \frac{(n-1)^2}{4} + \Theta(n^2)) + \Theta(n) \geq cnlgn + \Theta(n)$
$\rightarrow E(T(n)) = \Theta(nlgn)$

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

$Pr(i = medium)(normal) = \frac{1}{n}$
$Pr(i = medium)(3part) = \frac{6(\frac{1}{2}n-1)(n-\frac{1}{2}n)}{n(n-1)(n-2)} = \frac{3}{2}\frac{1}{n}$
$Pr(3part) - Pr(normal) = \frac{1}{2}\frac{1}{n}$

7.2.3 c

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

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

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 $\Omega(nlgn)$ 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

$\Sigma_1^n lgk < \int_1^{n+1} lgkdk = (klgk - k)_1^n = (nlgn - n) - (0 - 1) = nlgn - n + 1$

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 $\Omega(lg\frac{n!}{2^n}) = \Omega(nlgn - n) = \Omega(nlgn)$ 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)
$C[]$
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** $A.length$
    $C[A[j]] + +$
for $m = 1$ **to** k
    $C[m] + = C[m - 1]$
return $C[m]$

Direct-Quert(A, k, a, b)

C = Counting-sort-trim(A, k)
if $a < 1$
  return $C[b]$
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 digits take up to n possible values.

digitsConvert(X)

$result[]$
for $i = 2$ **downto** 0
  $result[i] = X/n^i$
  $X = X \bmod n^i$
return $result$

sort(A, x)

$result[]$
for each S in A
  S = digitsConvert(S)
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)

```
while 1
    if i == k
        return A[i]
    else
        q = RANDOM-PARTITION(A, p, r)
        if i < k
            r = q - 1
        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 $O(nlgn)$
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 $O(ilgn)$

9.5.3 c

Selecting: SELECT(A, i) in worst case $O(n)$
Sorting: MERGE-SORT$(A^{'})$ in worst case $O(ilgi)$

9.6 9.2

9.6.1 a

$\Sigma_1^{k-1} w_i = \Sigma_1^{k-1} \frac{1}{n} = \frac{k-1}{n} < \frac{1}{2}$
$\Sigma_{k+1}^{n} = \frac{n-k}{n} \leq \frac{1}{2}$

9.6.2 b

WEIGHT-MEDIAN(A)

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

9.6.3 c

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

if $i > lasti$
    return $lastsum +$ NORMAL-SUM($w_{lasti,i}$)
else return $lastsum -$ NORMAL-SUM($w_{i,lasti}$)

WEIGHT-MEDIAN-LINEAR(A)

while 1
    if $sum[w_1, w_i, lasti, lastsum] < \frac{1}{2}, sum[w_1, w_{i+1}, lasti, lastsum] > \frac{1}{2}$
        return i
    else
        $lastsum = sum[w_1, w_i, lasti, lastsum], lasti = i$
        if $sum[w_1, w_i] < \frac{1}{2}$
            $i =$ MEDIAN(A, i, r)
        else $i =$ MEDIAN(A, p, i)

We will experience *logn* literation, 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

$k \le i$ or $k \ge j : 0$
$i < k < j : \frac{2}{j-i+i}$

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 $vector < bool > A, a.size() = m$, just store the bool value of $key = m$ exist or not.

SEARCH(A, key)
if $A(key)$
    return key
else return NIL

INSERT(A, key)
$A(key) = 1$

DELETE(A, key)
$A(key) = 0$

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:
$Q_k = (\frac{1}{n})^k (\frac{n-1}{n})^{n-k} C_n^k$
$= \frac{(n-1)^{n-k}}{n^n} \frac{\Pi_0^{k-1} n-k}{k!}$
$\leq \frac{n^n}{n^n} \frac{1}{k!}$

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

11.2.4 d

Proof for Q_{k_0}:
$Q_{k_0} = \frac{e^{(\frac{clgn}{lglgn})}}{(\frac{clgn}{lglgn})^{\frac{clgn}{lglgn}}}$
$= \frac{n^{\frac{clg\frac{e}{c}}{lglgn}}}{\frac{n^c}{n^{\frac{clglglgn}{lglgn}}}} = n^{\frac{clg\frac{e}{c}+clglglgn}{lglgn}-c}$
It would not take effort to notice that since $lim_{n\to\infty}\frac{clg\frac{e}{c}+clglglgn}{lglgn} = 0$
$\forall c > 3+\epsilon, Q_{k_0} = O(\frac{1}{n^3})$
And $P_k \leq nQ_k \to P_k = O(\frac{1}{n^2})$

11.2.5 e

$E(M) = \Sigma_{M=1}^{n} MPr(M) < nPr(M > \frac{clgn}{lglgn}) + \frac{clgn}{lglgn}Pr(M \leq \frac{clgn}{lglgn})$
A stronger conclusion to note:
$E(M) = \Sigma_{M=1}^{n} MPr(M) < MPr(M > \frac{clgn}{lglgn}) + \frac{clgn}{lglgn}Pr(M \leq \frac{clgn}{lglgn})$
$\leq \int_{\frac{clgn}{lglgn}}^{\infty} \frac{1}{n}dn + 1 * \frac{clgn}{lglgn}$
$= lg(\frac{clgn}{lglgn}) + \frac{clgn}{lglgn}$
$= O(\frac{clgn}{lglgn})$

12 Binary Search Trees

13 Red-Black Trees

14 Augmenting Data Structures

15 Dynamic Programming

15.1 15.1-1

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

15.2 15.1-2

Do not know how!

15.3 15.1-3

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

```
r[] = c
for j = 1 to n
    for i = 1 to j
        r[i] ← max(p[i] + r[j − i] − c)
return r[n]
```

15.4 15.1-4

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

```
if m[n] > −1
    return m[n]
else
    for i ← 1 to n
        m[n] ← max(p[i] + r[n − i])
        s[n] ← i
    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 $T(n) = \Sigma_{k=1}^{n-1} T(k)T(n-k) = (n-1)c^2 2^n > c2^n$
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 $T(n) = 2\Sigma_{i=1}^{n-1} i(n-i)$

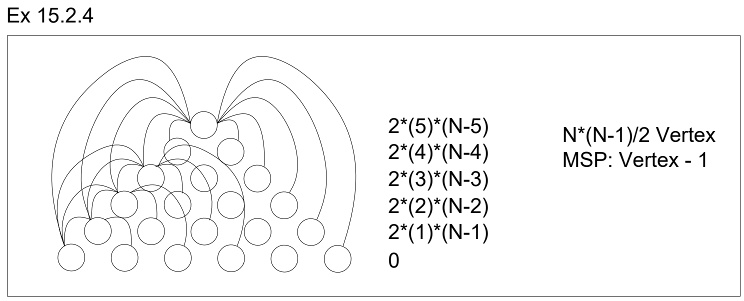

Figure 1: 15.2-4

$= \frac{3n^3+3n^2}{3} - \frac{2n^3+3n^2+n}{3}$
$= \frac{n^3-n}{3}$

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: $T(n) = 2\Sigma_{i=1}^{n-1} 4^i + n = \frac{8}{3}4^{n-1} + n \leq 4^n$
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)
DP ← [][]
return LSC-Aid(X.length, Y.length)
```

```
LSC-Aid(i, j)
if i = 0 or j = 0
    DP[i][j] ← 0
else
    if X[i] = Y[j]
        if DP[i − 1][j − 1] = NIL
            DP[i − 1][j − 1] = LSC-Aid(i − 1, j − 1)
        DP[i][j] ← DP[i − 1][j − 1] + 1
    else
        if DP[i − 1][j] = NIL
            DP[i − 1][j] = LSC-Aid(i − 1, j)
        if DP[i][j − 1] = NIL
            DP[i][j − 1] = LSC-Aid(i, j − 1)
        DP[i][j] = max{DP[i][j − 1], DP[i − 1][j]}
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)
DP ← [][]
DP[1] ← s[1]
for i ← 1 to n
    for j ← i − 1 downto 1
        if DP[j].end < s[i]
            DP[i] ← DP[i].length < DP[j].length + 1?DP[j] + s[i] : DP[i]
        else DP[i] ← DP[i].length < DP[j].length?DP[j] : DP[i]
return DP[s.length]
```

15.20 15.5-1

A Preorder Traverse of BST

PRE-ORDER-PRINT-AID$(i, j, root)$

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

PRE-ORDER-PRINT$(root)$

k $root[1,n]$ is the root
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)

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

LONGEST-SIMPLE-PATH-AID(s, t)

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

15.23 15.2

Consider the following $\Theta(n^2)$ algorithm:

Longest-palindrome-subsequence(S)

```
n ← S.size
for i ← 1 to n
    DP[i, i] ← 1
    if S[i] = S[i + 1]
        DP[i, i + 1] ← 2
    else DP[i, i + 1] ← 0
for l ← 3 to n
    for i ← 1 to n − l + 1
        if DP[i + 1, l − i] ≠ 0
            if S[i] = S[l − i + 1]
                S[i, l − i + 1] ← S[i + 1, l − 1] + 2
            else S[i, l − i + 1] ← 0
        else S[i, l − i + 1] ← 0
return DP[1, n]
```

15.24 15.3

Too hard

15.25 15.4

Printing-Neatly(l)

```
for j ← 1 to n
    DP[j] ← min_{M−j+i−Σ_i^j l_k ≥ 0, i ≤ j} {M − j + i − Σ_i^j l_k + DP[i − 1]}
return DP[n]
```

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)

```
DP = [][]
return AS-Adi(0, a.length)
```

AS-ADI(i, j)

for $m \leftarrow j-1$ **downto** $i+1$
    if $a[m].f \leq a[j].s$ and $a[m].s \geq a[i].f$
        $S[i][j].push(a[m])$
if $S[i][j] = \emptyset$
    $DP[i][j] \leftarrow 0$
else
    if $DP[i][j] = NIL$
        $DP[i][j] \leftarrow \max\limits_{a[k] \in S[i][j]} \{\text{AS-ADI}(i, k) + 1 + \text{AS-ADI}(k, j)\}$
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 $push \leftarrow 2$ and $pop \leftarrow 1$

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

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

22.3 22.1-3

```
TRANSPOSE(Adjlist)
new AdjlistPrime
for each node in Adjlist
    for each subnode in Adjlist(node)
        AdjlistPrime(subnode).insert(node)
        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)
for each pair(i, j) in upper left Adjmatrix
    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)
for each u in G.vertices
    G.reset()
    list = ∅
    u.adjlist' = BFS-AID(G, u, list, 0)
```

```
BFS-AID(G, u, list, dist)
for each v in u.adjlist
    if v.color = white and dist ≤ 2
        list.insert(u)
        BFS-AID(G, v, list, dist + 1) =
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)
for each m in G.adjMatrix
    for each n G.adjMatrix[m]
        if G.adjMatrix[m][n] == 1
            G'.adjMatrix[m] = AND(G.adjMatrix[m], G.adjMatrix[n])
return G'
```

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

22.6 22.2-3

use $u.d = \infty$ as color

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** : $each \quad v \in M[u]$
line 13 $\rightarrow$ **if** : $v ==$ **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.
*line*3 would be execute $\Theta(E)$ times in total,

SQUARE$(Adjmatrix)$

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

22.9 Edge traverse of undirected graph

According to $Theorem 22.10$, 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)

$u.color = grey$
$dict[(Vertex, Vertex), edgeType] = \emptyset$
for each v in $u.adjList$
    if $v.color == white$
        $dict(u, v) = treeEdge$
        DFS-VISIT(G, v)
    else $dict(u, v) = backEgde$
PRINT-PATH(G, u)

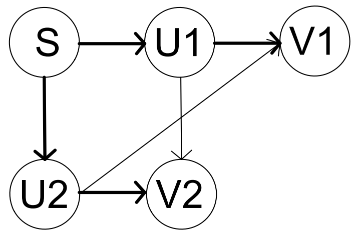

Figure 2: 22.2-6

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

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:
$E_\pi =< s, u1 >, < u1, v1 >, < s, u2 >, < u2, v2 >$
In BFS Tree, $\delta(s, v1), \delta(s, v2)$ is either $< s, u1, v1 >, < s, u1, v2 >$ or $< s, u2, v1 >$ $, < s, u2, v2 >$

22.11 22.2-7

BFS and see if there is a cycle composed of odd number of node

BFS-CHECK(G, r)

```
for each v ∈ G.V
    v.d ← ∞
    v.π = nullptr
    v.color = white
r.d ← 0
Q ← ∅
ENQUEUE(Q, r)
while Q ≠ ∅
    u ← DEQUEUE(Q)
    for each v ∈ u.adjlist
        if v.color = white
            v.color ← gray
            v.d ← u.d + 1
            v.π ← u
            ENQUEUE(Q, v)
        else return false
    v.color ← black
return true
```

22.12 22-3.5

22.12.1 a

$u.d < v.d < v.f < u.f \leftrightarrow$
(u, v) is discovered either $[u.d, v.d]$ or $[v.f, u.f]$

22.12.2 b

$v.d \leq u.d < u.f \leq v.f \leftrightarrow$
(u, v) is discovered when $v.color = grey \leftrightarrow$
(u, v) is a back edge

22.12.3 c

$v.d < v.f < u.d < u.f \leftrightarrow$
u, v has no parental relationship

22.13 22-3.7

Rewrite DFS with stack:

```
PUSH-VERTEX(S, u, u.π)
time ← time + 1
u.d ← time
u.color ← GREY
u.π ← u.π
PUSH(S, u)
```

```
POP-VERTEX(S)
time ← time + 1
u ← POP(S)
u.f ← time
u.color ← BLACK
```

```
DFS-VISIT(G, r)
S ← ∅
PUSH-VERTEX(S, r)
while S ≠ ∅
    u ← TOP(S)
    finish ← true
    for each v ∈ u.adjlist
        if v.color = white
            PUSH-VERTEX(S, v, u)
            finish ← false
    if finish
        POP-VERTEX(S)
```

22.14 22-3.12

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

```
DFS(G)
for each u in G.V
    u.color = white
c = 1
for each u in G.V
    if u.color = white
        DFS-VISIT(G, u, c)
        c++
```

```
DFS-VISIT(G, u, c)
u.color = grey
u.cc = c
for each v in u.adjList
    if v.color == white
        DFS-VISIT(G, v)
```

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

22.15 22.4-1

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

22.16 22.4-3

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

22.17 22.1

22.17.1 a-1

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

22.17.2 a-2

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

22.17.3 a-3

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

22.17.4 b-1

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

22.17.5 b-2

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

22.17.6 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).

22.17.7 b-4

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

22.18 22.3

22.18.1 1. proof

Euler tour exist $\to$ in-degree == out-degree: Suppose the cycle through i vertex n times would be $E - cycle = \{v_i, v_j, v_k, ..., v_i\}$. 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 $\to$ Euler tour exist

22.18.2 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)
ClosedCycleSet, OpenCycleSet = ∅
while alladjList! = ∅
    for v in Vertex with adjList! = ∅
        if v.adjList! = ∅
            new cycle = ∅
            CircleFindAid(cycle, u, NIL)
            if cycle.type == closed
                ClosedCycleSet.push(cycle)
            else OpenCycleSet.push(cycle)
```

```
CircleFindAid(cycle, u, v)
cycle.insert(u)
if v! = NIL
    v.adjList.erase(u)
if u == NIL
    cycle.type = open
    return
elseif u == cycle.start
    cycle.type = close
    return
else v = u
    u = u.adjList.begin()
    CircleFind(cycle, u, v)
```

23 Minimum Spanning Trees

23.1 Exercise 23.1-1

Consider the $Edge(u, v)$ connects u and v, we can find a cut that separate u from other. Initialize from u as MST, then $Edge(u, v)$ is the light edge for $A = \{u\}$.

23.2 Exercise 23.1-2

Consider $\{(a, b), (a, c), (a, d)\}$, and the cut separate $\{a\}$ and $\{b, c, d\}$.

24 Single-Source Shortest Paths

24.1 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.

24.2 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, ω, relaxed)
if v.d > u.d + ω(u, v)
    v.d ← u.d + ω(u, v)
    v.π ← u
    relaxed ← true
```

```
BELLMAN-FORD-FASTER(G, ω, root)
INITIALIZE-SINGLE-SOURCE(G, root)
relaxed ← true
counter ← 1
while relaxed
    relaxed ← false
    if counter > |G.V|
        return false
    for each edge (u,v) ∈ G.E
        RELAX-FASTER(u, v, ω, relaxed)
    counter ← counter + 1
return true
```

24.3 Exercise 24.1-4

This is a simple one:

```
BELLMAN-FORD-ALL(G, ω, root)
INITIALIZE-SINGLE-SOURCE(G, root)
for i ← 1 to |G.V| − 1
    for each edge(u,v) ∈ G.E
        RELAX(u, v, ω)
for each each edge(u,v) ∈ G.E
    if v.d > u.d + ω(u, v)
        v.d ← −∞
```

24.4 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.

24.5 Exercise 24.3-2

The Dijkstra's algorithm works since we know each node v when poped 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 poped, 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: $Vertex = \{x, y, z\}, Edge = \{(x, y) = 4, (x, z) = 3, (y, z) = -10\}$

24.6 Exercise 24.3-3

This is supposed to work since the last vertex in queue has no vertex to point to and no edge to relax.

24.7 Exercise 24.3-3

24.8 Exercise 24.3-6

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

INITIALIZE$(G, root)$

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

RELAX(u, v, r)

if $u.d < v.d * r(u, v)$
    $u.d \leftarrow v.d * r(u, v)$
    $v.\pi \leftarrow u$

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

INITIALIZE$(G, root)$
$Q \leftarrow G.V$
while $!Q.empty$
    $u =$ EXTRACT-MAX(Q)
    for each $v \in G.E(u)$
        RELAX$(u, v, G.r)$

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

if $v = root$
    return
else
    if $v.\pi = nullptr$
        PRINT(NO SUCH PATH)
    else
        PRINT-PATH$(G, r, v.\pi)$
        PRINT(V)

24.9 Exercise 24.4-2

Consider the following buttom-top DP solution: 1. Topo sort and list middle vertex $\{s, v_1, v_2, ...v_k\}$ between s and t, $\Theta(V+E)$
2. $DP[k] = u(v_k, v)$ 3. $DP[k-1] = u(v_{k-1}, v) + u(v_{k-1}, v_k)DP[k]$

This procedure takes at most $\Theta(V+E)$ time since all the add would happen within $O(E)$

24.10 Exercise 24.4-3

Modify DFS to detect backedge, using and return true when reach $|V|$ number of edges.

24.11 Exercise 24.4-5

Count the indegree of each vertex $\Theta(V+E)$
Iteratively Remove 0-in-degree vertex, put it on tail of linked list and subtract the in-degree of other vertex in its adjacent list $\Theta(E)$

25 All-Pairs Shortest Paths

25.1 Question 6: CLRS Exercise 25.2-2

Consider the following dynamic programming function:

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

TRANSITIVE-CLOSURE-BRUTAL-FORCE(G)

$n = G.V$
$l[n][n][n] \leftarrow 0$
for each entry in $l[0]$
    $l[0][i][j] \leftarrow i = j$
for $m \leftarrow 1$ **to** $n-1$
    for $i \leftarrow 1$ **to** n
        for $j \leftarrow 1$ **to** n
            $l[m][i][j] = \bigcup\limits_{1 \le k \le n} (l[m-1][i][k] \cap (G.w_{kj} \neq \infty))$