14.1-3

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

```
Rod_Cutting(p[], n, c) // p[] is the price list
                       // n is the length of the rod
// let rev[1:n] be the   // c is the cutting cost
   new array
   for i = 1 to n
      rev[i] = p[i]          // init
      for j = 1 to i-1        // run all possible cut
         rev[i] = max(rev[i], p[j] + rev[i-j] - c)
   return rev[n]
```

14.1-5
Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution.

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0:n] be a new array          // will remember solution values in r
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0                        // already have a solution for length n?
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n                 // i is the position of the first cut
7          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
8  r[n] = q                           // remember the solution value for length n
9  return q

BOTTOM-UP-CUT-ROD(p, n)
1  let r[0:n] be a new array          // will remember solution values in r
2  r[0] = 0
3  for j = 1 to n                     // for increasing rod length j
4      q = -∞
5      for i = 1 to j                 // i is the position of the first cut
6          q = max{q, p[i] + r[j - i]}
7      r[j] = q                       // remember the solution value for length j
8  return r[n]
```

Let S[0:n] be a global array

```
MEMOIZED-CUT-ROD(p, n)
  for i=0 to n
      S[i] = 0   // S init
      r[i] = -∞
return [MEMOIZED-CUT-ROD-AUX(p,n,r)
        , S[n]]
// S[n] is the optimized place
   to be cut
```

```
MEMOIZED-CUT-ROD-AUX(p,n,r)
  if r[n] ≥ 0
     return r[n]
  if n==0
     q=0
  else q=-∞
     for i=1 to n
         tmp = p[i] + MEMOIZED-CUT-ROD-AUX(p, n-i, r)
         if q < tmp
             q=tmp
             S[n] = i // save the optimized solution
     r[n] = q
  return q
```

14.2-1
Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j : i \le k < j\} & \text{if } i < j. \end{cases}$$

A: 5×10 B: 10×3 C: 3×12 D: 12×5 E: 5×50 F: 50×6

	A	B	C	D	E	F
A	0	150 (AB)	330 (ABC)	405 (ABCD)	1655 (ABCDE)	2010 (ABCDEF) ((AB)(CDEF))
B		0	360 (BC)	330 (BCD)	2430 (BCDE)	1950 (BCDEF)
C			0	180 (CD)	930 (CDE)	1770 (CDEF) (CD)(EF)
D				0	3000 (DE)	1860 (DEF)
E					0	1500 (EF)
F						

(每格表所需最小運算)

⇒ ((AB)((CD)(EF))) #

14.2-6
Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

the multiplication will be $A_1 A_2 \cdots A_n$

↑ ↑ ⋯ ↑ → n-1 arrows

To select a place to break $A_1 A_2 \cdots A_n$ into 2 pieces, there are n-1 ways i.e. n-1 pairs of parentheses. #

14.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

To enumerate all the ways,

the formula is: $P(n) = \begin{cases} 1, & \text{if } n = 1 \\ \sum_{i=1}^{n-1} (P(n-i) \cdot P(i)), & \text{if } n \geq 2 \end{cases}$

the time complexity is equal to counting catalan number

if there are n numbers, the number of counting is

$P(n) = P(1) + P(n-1) + P(2) + P(n-2) \cdots + P(n-1) + P(1) + \underline{2(n-1)}$

$= 2(P(1) + P(2) \cdots P(n-1)) + 2(n-1)$ — ①

counting $P(1) \cdot P(2) \cdots P(n-1)$

Similarly, if there are n+1 numbers

$P(n+1) = 2(P(1) + P(2) \cdots P(n)) + 2n$ — ②

we could get ② − ①: $P(n+1) - P(n) = 2P(n) + 2$

$\Rightarrow P(n+1) = 2 + 3P(n)$

i.e. $P(n) = 2 + 3P(n-1)$

$\Rightarrow P(n) + 1 = 3 + 3P(n-1)$

$= 3(P(n-1) + 1)$

$= 9(P(n-2) + 1)$

$= 3^{n-2}(P(2) + 1)$

$= 3^{n-1}$

$P(n) = 3^{n-1} - 1 = \Omega(3^{n-1})$

The recursive time complexity : $T(n) \le \begin{cases} c & \text{if } n=1 \\ c + \sum_{i=1}^{n-1} T(i) + T(n-i) + c & \text{if } n \ge 2 \end{cases}$

$\Rightarrow T(n) \le cn + 2\sum_{i=1}^{n-1} T(i)$

assume $T(n) \le cn3^{n-1}$

$T(n) \le cn + 2\sum_{i=1}^{n-1} ci\,3^{i-1}$

$= c\left(n + 2\sum_{i=1}^{n-1} i\,3^{i-1}\right)$

$= c\left(n + n3^{n-1} + \frac{1-3^n}{2}\right)$

$= cn3^{n-1} + c\left(n + \frac{1-3^n}{2}\right)$

$\le cn3^{n-1}$

since $2n+1-3^n \le 0$ for $n \ge 1$

$\Rightarrow T(n) = O(n3^{n-1})$

$\sum_{i=1}^{n-1} x^i = \frac{x^n-1}{x-1} - 1$

微分 $\Rightarrow \sum_{i=1}^{n-1} i x^{i-1} = \frac{nx^{n-1}(x-1) - (x^n-1)}{(x-1)^2}$

$= \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2}$

$x=3 \Rightarrow \frac{n3^{n-1}}{2} + \frac{1-3^n}{4}$

In summary, running recursively runs $O(n3^{n-1})$ and enumeration runs $\Omega(n3^{n-1})$. Consequently, running recursively is faster. #

14.3-4

As stated, in dynamic programming, you first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that she does not always need to solve all the subproblems in order to find an optimal solution. She suggests that she can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

Assuming there are A_1 A_2 A_3

To find $p_0 p_1 p_3$ $A_1(A_2A_3)$ $<$ $p_0 p_2 p_3$ $(A_1A_2)A_3$ and $p_0 p_1 p_2 + p_0 p_2 p_3 < p_1 p_2 p_3 + p_0 p_1 p_3$

Let $p_0, p_1, p_2, p_3 = 1, 2, 3, 3$

in this case, $p_0 p_1 p_3 < p_0 p_2 p_3$, choosing $A_1(A_2A_3)$

$A_1(A_2A_3)$: $2 \cdot 3 \cdot 3 + 1 \cdot 2 \cdot 3$ and $(A_1A_2)A_3$: $1 \cdot 2 \cdot 3 + 1 \cdot 3 \cdot 3$

$(A_1A_2)A_3$ is optimal solution and Caplet Choose $A_1(A_2A_3)$ which is not optimal solution #

(沒有鬲頸，又有老柏)

14.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

```
Find_LCS (X, Y, C, i, j)
  if i = 0 or j = 0 return
  if X[i] = Y[j]
      Find_LCS (X, Y, C, i-1, j-1)
      print (X[i])
  else if C[i-1, j] == C[i, j]
      Find_LCS (X, Y, C, i-1, j)
  else
      Find_LCS (X, Y, C, i, j-1)
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Note:
call Find_LCS (X, Y, C, m, n)
to print LCS

The time complexity is $O(m+n)$ #

14.4-5

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Let the Array be A

First of all, make a duplicate of A called A'. and sort the A'. Then, find LCS of A and A'.

Time complexity :

sorting : could me $O(n \lg n)$

LCS : $O(n^2)$

⇒ Totally : $O(n^2)$ #

Exercises

14.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST($root, n$) which, given the table $root[1:n, 1:n]$, outputs the structure of an optimal binary search tree. For the example in Figure 14.10, your procedure should print out the structure

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 14.9(b).

```
CONSTRUCT-OBST-SUBTREE
    (root[], i, j, r, dir)
if j<i  || j=i-1
    print("d"j is the dir
        of "k"r)
else
    rt = root[i, j]
    print("k"rt is the dir
        of "k"r)
    CONSTRUCT-OBST-SUBTREE
        (root, i, rt-1, rt, "left")
    CONSTRUCT-OBST-SUBTREE
        (root, rt+1, j, rt, "right")

CONSTRUCT-OPTIMAL-BST (root[], n)
    r = root[1, n]
    print("K"r is the root)
    CONSTRUCT-OBST-SUBTREE (root, 1, r-1, r, "left")
    CONSTRUCT-OBST-SUBTREE (root, rt1, n, r, "right")
```

14.5-2

Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

execution of the OBST function

e structure

0.06	0.28	0.62	1.02	1.34	1.83	2.44	3.12
INF	0.06	0.30	0.68	0.93	1.41	1.96	2.61
INF	0.00	0.06	0.32	0.57	1.04	1.48	2.13
0.00	0.00	INF	0.06	0.24	0.57	1.01	1.55
INF	0.00	INF	0.00	0.05	0.30	0.72	1.20
INF	0.00	INF	0.00	INF	0.05	0.32	0.78
0.06	0.16	2.88	0.00	0.00	0.00	0.05	0.34
INF	0.00	INF	0.00	INF	0.00	INF	0.05

root structure

i \ j	1	2	3	4	5	6	7
1	1	2	2	2	3	3	5
2	0	2	3	3	3	5	5
3	1	5	3	3	4	5	5
4	0	0	0	4	5	5	6
5	1	0	0	0	5	6	6
6	0	0	0	0	0	6	7
7	0	0	0	0	0	0	7

By root[1,7] we could get the root is 5

To construct the tree, use the method in last problem.

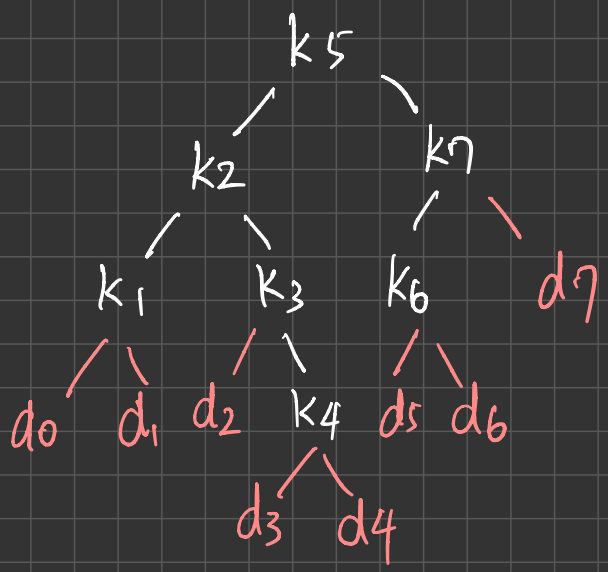
L-tree

root[1:4] = 2

root[3:4] = 3

R-tree

root[6,7] = 7



14-2 *Longest palindrome subsequence*

A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

Let dp[1:n][1:n] and p[1:n][1:n] be new array
and the length of the word be n
for i=1 to n // init
    for j=1 to n
        dp[i][j] = -1

Find_LPS(i, j, w[])
    if i=j return 1
    if i>j return 0
    if dp[i][j] ≠ -1 return dp[i][j]
    if w[i]=w[j]
        dp[i][j] = Find_LPS(i+1, j-1, w) + 2
        p[i][j] = "↙" // i+1, j-1
    else
        temp1 = Find_LPS(i, j-1, w)
        temp2 = Find_LPS(i+1, j, w)
        if temp1 < temp2
            dp[i][j] = temp1
            p[i][j] = "←" // j-1
        else
            dp[i][j] = temp2 p[i][j] = "↓"

Return_LPS(i, j, p[][], w[])
    if(i>j) return "" // empty string
    if(i=j) return w[i] // the middle word
    if p[i][j] = "←" return Return_LPS(i, j-1, p, w)
    if p[i][j] = "↓" return Return_LPS(i+1, j, p, w)
    if p[i][j] = "↙" return w[i] + Return_LPS(i+1, j-1, p, w) + w[j]

Note: To get LPS, call Find_LPS(1, n, w) and Return_LPS(1, n, w)

Time Complexity: $O(n^2)$

15.1-2
Suppose that instead of always selecting the first activity to finish, you instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

Given $S = \{a_1, a_2 \cdots a_n\}$

$St = \{s_1, s_2 \cdots s_n\}$ is a optimal set of starting time

$Fi = \{f_1, f_2 \cdots f_n\}$ is a optimal set of finish time

$a_i = [s_i, f_i)$ (a_i 要在 s_i 开始 f_i 结束)

Create a $S' = \{a_1', a_2' \cdots a_n'\}$

$a_i' = [f_i, s_i)$ i.e. a_i' is a_i in reverse

$\Rightarrow$ a subset $\{a_{i1}, a_{i2} \cdots a_{ik}\} \subseteq S$

$\Leftrightarrow \{a_{i1}', a_{i2}' \cdots a_{ik}'\} \subseteq S'$

i.e. selecting the first activity to finish is compatible with selecting the last activity #

To prove that it yields an optimal solution

Let S_{ij} be the set of activities that start after a_i and finish before a_j. To find the maximum set of S_{ij}, let the maximum set is A_{ij}, including some activity a_k. Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$

$\Rightarrow A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

S_{ij} contains $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

A_{ij} includes optimal solutions to S_{ik} and S_{kj}.

If I could find a set A'_{kj} in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then I could use A'_{kj} rather than A_{kj}. A_{ik} is similarly.

$\Rightarrow |A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$

$\Rightarrow A_{ij}$ is an optimal solution #

15.1-4

You are given a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. You wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the ***interval-graph coloring problem***. It is modeled by an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

f contains $[f_1, t_2 \dots f_n]$

note: S contains $[s_1, s_2 \dots s_n]$

s_i means the starting time of activity i

f_i means the end time of activity i

```
Hull_Assign(S[], f[], n)
Let STACK be the DSA of stack and create n Hulls variable
create t[1:2n] and a[1:n]
for i = n downto 1
    STACK.push(Hull_i)
for i = 1 to n
    t[i] = {s[i], activity = i, type = "start"}
    t[n+i] = {f[i], activity = i, type = "finish"}
sort(t)  // could be merge sort, sorted by the value of time

for i = 1 to 2n-1
    if t[i].type == "finish"
        idx = t[i].activity
        STACK.push(a[idx])
    else
        idx = t[i].activity
        a[idx] = STACK.pop()
```

a[i] means the hull that activity i use

Note: the hull that activity use is stored in a[i]

Time Complexity : $O(n \lg n)$

15.2-1
Prove that the fractional knapsack problem has the greedy-choice property.

let there are item a, b where $v_a/w_a > v_b/w_b$
ie. the average value of a is larger than b
Let $n = \min(w_a, w_b)$ If we take n weight of b we get $n \times v_b/w_b$. However, if we take n weight of a we get $n \times v_a/w_a$. The total value increase since $n \times v_a/w_a - n \times v_b/w_b > 0 \Rightarrow$ It has the greedy-choice property

15.2-2
Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(n\,W)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in the knapsack.

assuming $w = [w_1, w_2 \cdots w_n]$
w_i means the weight of item i

```
Solve Knapsack (w[], n, W)
    create dp[0:w] and p[0:w]
     max_val=0
    for i=1 to w
        dp[i] =0
    dp[0] =1
    for i=1 to n
        for j= W to W-wi
            if dp[j-wi] =1
                dp[j] =1
                p[j]= i
                max_val = max(j, max_val)
    Find_Item (p[], w[], max_val )
Find_Item (p[], w[], n)
    if n=0 return
    get one item w[n]
    Find_Item (p[], d[], n-d[p[n]])
```

15.3-1
Explain why, in the proof of Lemma 15.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

Lemma 15.2 (Optimal prefix-free codes have the greedy-choice property)
Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. In such a tree, the codewords for x and y have the same length and differ only in the last bit.

Let a and b be any two characters that are sibling leaves of maximum depth in T. Without loss of generality, assume that $a.freq \le b.freq$ and $x.freq \le y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \le a.freq$ and $y.freq \le b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$, but $x.freq = b.freq$ implies that $a.freq = b.freq = x.freq = y.freq$ (see Exercise 15.3-1), and the lemma would be trivially true. Therefore, assume that $x.freq \ne b.freq$, which means that $x \ne b$.

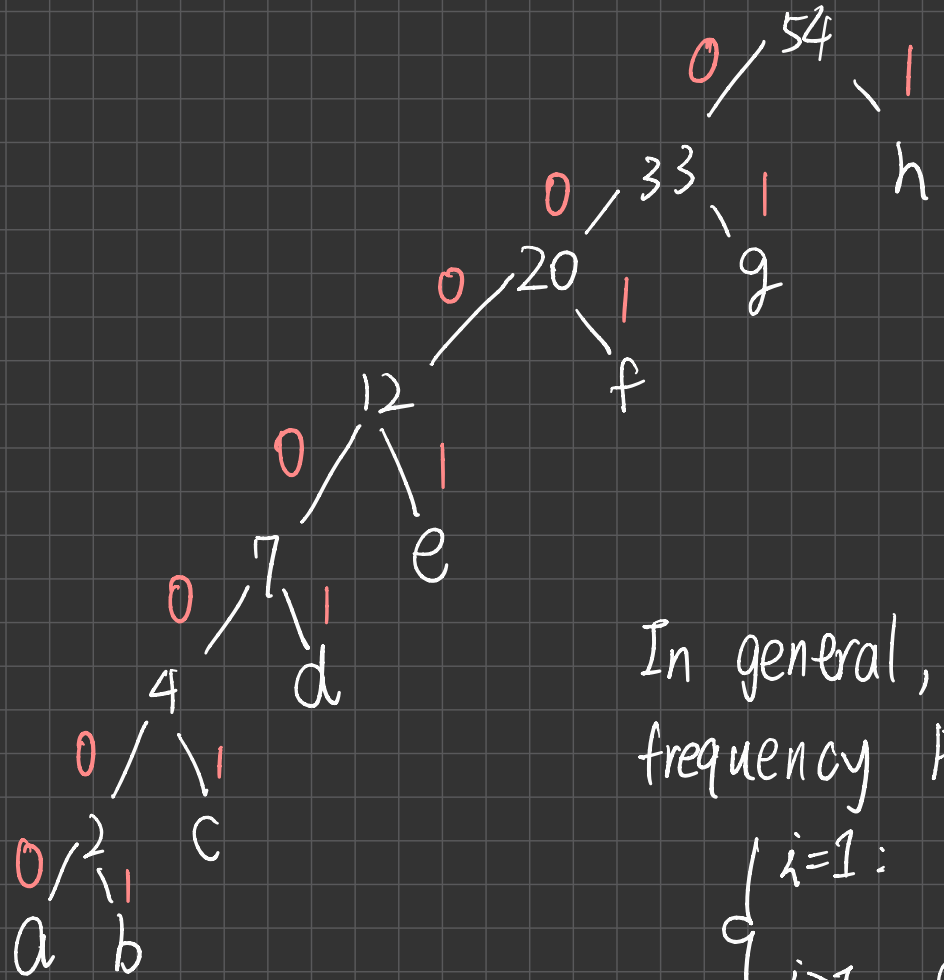
As Figure 15.7 shows, imagine exchanging the positions in T of a and x to produce a tree T', and then exchanging the positions in T' of b and y to produce a

$a.freq \le b.freq$
and $x.freq \le y.freq$
if $x.freq = b.freq$
then
$a.freq \le b.freq = x.freq \le y.freq$
since x and y be two characters having the lowest frequency
we have $b.freq \le y.freq \Rightarrow b.freq = y.freq$
and $a.freq \le x.freq \Rightarrow a.freq = x.freq$
Thus, the freq of a, b, x, y are the same.

15.3-3
What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?



a 0000000
b 0000001
c 000001
d 00001
e 0001
f 001
g 01
h 1

In general, the character with frequency F_i will be

$i=1$: $0\underbrace{\cdots}_{n-1}0$

$i \ge 1$: $0\underbrace{\cdots}_{n-i}01$ #

15.4-1

Write pseudocode for a cache manager that uses the furthest-in-future strategy. It should take as input a set C of blocks in the cache, the number of blocks k that the cache can hold, a sequence $b_1, b_2, \ldots, b_n$ of requested blocks, and the index i into the sequence for the block b_i being requested. For each request, it should print out whether a cache hit or cache miss occurs, and for each cache miss, it should also print out which block, if any, is evicted.

note $b[1:n] = [b_1, b_2 \cdots b_n]$

```
Furthset_In_Future (b[], n, k)
    Let S be a set
    i=1
    while i ≤ n
        if b[i] in S
            print b[i] "cache hit"
        else if |S| < k   // cache is not full
            print b[i] "cache miss"
            S = S ∪ b[i]   // add b[i] to cache
        else
            break   // cache is full
        i = i+1
    let priority[1:n] be the new array
    for j = 1 to n
        priority[j] = 0   // init
    for j = i to n
        priority[ b[j] ] = n+1 - j   // get b[j] priority
    Let Q be a min-priority queue, sort by
        the second key
    for v in S
        ENQUEUE (Q, {v, priority[v]} )
```

cont.

```
m = i
while   i ≤ n
    if b[i] in S
        print b[i] "cache hit"
    else
        print b[i] "cache miss"
        if PQ is not empty
            {Z, p} = DEQUEUE (PQ) // get the lowest priority value
                S = [S - {Z}) ∪ b[i]
                if priority [b[i]] = n+1-i // {b[k] ≠ b[i] | k > i[
                    priority [b[i]] = 0
                ENQUEUE (PQ, {b[i], priority [b[i]])
        else
            Z = b[m]
            S = (S - {Z}) ∪ b[i]
        m = i
    i = i + 1
```

15.4-3

Professor Croesus suggests that in the proof of Theorem 15.5, the last clause in property 1 can change to $C_{S',j} = D_j \cup \{x\}$ or, equivalently, require the block y given in property 1 to always be the block x evicted by solution S upon the request for block b_i. Show where the proof breaks down with this requirement.

Theorem 15.5 (Optimal offline caching has the greedy-choice property)
Consider a subproblem (C, i) when the cache C contains k blocks, so that it is full, and a cache miss occurs. When block b_i is requested, let $z = b_m$ be the block in C whose next access is furthest in the future. (If some block in the cache will never again be referenced, then consider any such block to be block z, and add a dummy request for block $z = b_m = b_{n+1}$.) Then evicting block z upon a request for block b_i is included in some optimal solution for the subproblem (C, i).

Proof Let S be an optimal solution to subproblem (C, i). If S evicts block z upon the request for block b_i, then we are done, since we have shown that some optimal solution includes evicting z.

So now suppose that optimal solution S evicts some other block x when block b_i is requested. We'll construct another solution S' to subproblem (C, i) which, upon the request for b_i, evicts block z instead of x and induces no more cache misses than S does, so that S' is also optimal. Because different solutions may yield different cache configurations, denote by $C_{S,j}$ the configuration of the cache under solution S just before the request for some block b_j, and likewise for solution S' and $C_{S',j}$. We'll show how to construct S' with the following properties:

1. For $j = i + 1, \ldots, m$, let $D_j = C_{S,j} \cap C_{S',j}$. Then, $|D_j| \geq k - 1$, so that the cache configurations $C_{S,j}$ and $C_{S',j}$ differ by at most one block. If they differ, then $C_{S,j} = D_j \cup \{z\}$ and $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$.
2. For each request of blocks $b_i, \ldots, b_{m-1}$, if solution S has a cache hit, then solution S' also has a cache hit.
3. For all $j > m$, the cache configurations $C_{S,j}$ and $C_{S',j}$ are identical.
4. Over the sequence of requests for blocks $b_i, \ldots, b_m$, the number of cache misses produced by solution S' is at most the number of cache misses produced by solution S.

x: evict block x when b_i is requested

y: evict block y when b_j is requested

Note that $j = i+1, \ldots m > i$

If $C_{S'j} = D_j \cup \{x\}$ be changed into $C_{S'j} = D_j \cup \{y\}$,

meaning that when b_{j-1} is requested, the y is evicted.

However, when b_j is requested, y is evicted again!

y is evicted twice is not true, since the value in cache won't be duplicated. #

15-1 Coin changing

Consider the problem of making change for n cents using the smallest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in denominations that are powers of c: the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

d. Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations using the smallest number of coins, assuming that one of the coins is a penny.

a.

Exchange (n)

$n_q = \lfloor n/25 \rfloor$

$n = n \bmod 25$

$n_d = \lfloor n/10 \rfloor$

$n = n \bmod 10$

$n_n = \lfloor n/5 \rfloor$

$n = n \bmod 5$

$n_p = n$

Note: the total amount of coins is $n_q + n_d + n_n + n_p$

and we exchange n_q quarters

n_d dimes

n_n nickels

n_p pennies

To prove that it is a optimal solution, let C be the largest coin when exchange n cents

if $n = 0$, then no coins return

if $n > 0$, let us analyze the problem

$0 < n < 5$, $C = 1$ it could contain only pennies

$5 \leq n < 10$, $C = 5$ if $C \neq 5$, we could use 1 nickel to replace 5 pennies, reducing the number of coins.

$10 \leq n < 25$, $C = 10$ if $C \neq 10$, it contains pennies and nickels, we could change ten dollars of pennies with nickels, reducing the number of coins.

$25 < n$, $C = 25$ if $C \neq 25$, it contains pennies, nickels and dimes. If there are 3 dimes, we should exchange them into 1 quarter and 1 pickel, reducing the cost.

If there are 25 dollars of coins, exchange

then into a quarter, reducing the costs.

⇒ In summary, we exchange n cents into the largest coin and solve the n−c cents subproblem. If yields the optimal solution #

b. Assume that we exchange n cents.

By the algorithm in (a), we get $\lfloor n \bmod c^{i+1} / c^i \rfloor$ of c^i where $i = 0 \sim k-1$ and get $\lfloor n / c^k \rfloor$ of c^k. To prove that the algorithm yields optimal solution, prove the not optimal solution yields not optimal solution.

pf

Let a_i be the number of coin c^i

$j = \max \{ 0 \le i \le k \mid c^i \le n \}$

$\Rightarrow \sum_{i=0}^{j-1} a_i c^i = n$ [not greedy]

since $n \ge c_j$, $\sum_{i=1}^{j-1} a_i c^i \ge c_j$

Suppose the not greedy solution is optimal

Since it is optimal, we could exchange $(a_i - c) c^i$ into $1\ c^{i+1}$ when $a_i > c$ i.e. $a_i < c$ for all i

$\Rightarrow a_i \le c-1$

$$\sum_{i=0}^{j-1} a_i c^i \le \sum_{i=0}^{j-1} (c-1) c^i$$

$$= (c-1) \frac{c^j - 1}{c-1} = c^j - 1 < c^j \ (\times)$$

So the greedy algorithm yields optimal solution

c. If there are only penny, dime and quarter.
30 cents with the greedy algorithm would yield
1 quarter and 5 pennies. However, we could
just use 3 dimes, taking less coins than the
algorithm. It doesn't yield optimal solution. #

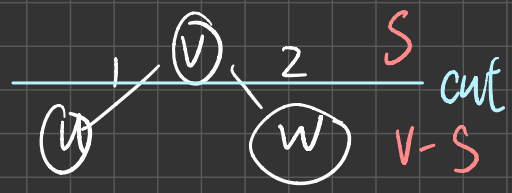
d.
```
EXCHANGE (d[], n, k)   // d = [d1, d2, d3 ... dk]
    create dp[0:n] and p[1:n]
    dp[0] = 0
    for i = 1 to n   // i cents
        dp[i] = ∞
        for j = 1 to k
            if i ≥ d[j] and 1 + dp[i - d[j]] < dp[i]
                dp[i] = dp[i - d[j]] + 1
                p[i] = d[j]
    return dp[n] and p[]
```

```
Find_Coin (p[], d[], n)
    if n = 0 return
    get one d[p[n]] coin
    Find_Coin (p[], d[], n - d[p[n]])
```

Note: The dp[n] contains the least number of coins to be exchanged from n. To find the type of coin, use Find_Coin (p[], d[], dp[n]) #

21.1-2
Professor Sabatier conjectures the following converse of Theorem 21.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.



$G = (\{u, v, w\}, \{(u,w), (v,w)\})$

both (u,v) and (v,w) are safe edges crossing $(S, V-S)$. However, (u,w) is not a light edge for the cut. # (*)

21.1-3
Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Note: $w(i,j)$ means the weight of edge (i,j)

Let T be the MST of G.

$\{(u,v) \in T \mid (u,v)$ is the edge crossing some cut of $(S, V-S)$ and (u,v) is not a light edge $\}$

$\Rightarrow \exists\, w(x,y)$ s.t. $w(x,y) < w(u,v)$

Theorem 21.1
Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

By the thm., we could construct spanning tree T', removing (u,v) from T and adding (x,y)

$\Rightarrow w(T') = w(T) - w(u,v) + w(x,y)$
$< w(T)$ (*)

$\Rightarrow (u,v)$ is a light edge crossing some cut. #

21.2-4

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  create a single list of the edges in G.E
5  sort the list of edges into monotonically increasing order by weight w
6  for each edge (u, v) taken from the sorted list in order
7      if FIND-SET(u) ≠ FIND-SET(v)
8          A = A ∪ {(u, v)}
9          UNION(u, v)
10 return A
```

1. The sorting time could be $O(|V|+|E|)$ by counting sort. $\because |V| = O(|E|) \Rightarrow O(|V|+|E|) = O(|E|)$

The 6-9 lines cost $O(|V|+|E|)\,\alpha(|V|)$

$= O(|E|\,\alpha\,|V|)$

Time Complexity: $O(|E|\,\alpha(|V|))$

2. As above, sorting cost $O(W+|E|)$

The operation of set cost $O(|V|+|E|)\alpha(|V|)$

$= O(|E|\,\alpha\,|V|)$

Time Complexity: $O(|E|\,\alpha\,|V|)$ #

21.2-5

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

```
MST-PRIM(G, w, r)
1  for each vertex u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = ∅
6  for each vertex u ∈ G.V
7      INSERT(Q, u)
8  while Q ≠ ∅
9      u = EXTRACT-MIN(Q)          // add u to the tree
10     for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11         if v ∈ Q and w(u, v) < v.key
12             v.π = u
13             v.key = w(u, v)
14             DECREASE-KEY(Q, v, w(u, v))
```

line 6-7: the time for priority queue insert $|v|$ elements could be $O(|V| \lg\lg |V|)$ by using Emde Boas tree

line 10-14: $O(E)$

1. $O(|E| + |V| \lg\lg |V|)$

2. W is constant $\Rightarrow |V|$ is constant

$\Rightarrow O(|E|)$ #

21-1 Second-best minimum spanning tree

Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of G, and let T be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** is a spanning tree T' such that $w(T') = \min\{w(T'') : T'' \in \mathcal{T} - \{T\}\}$.

a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

b. Let T be the minimum spanning tree of G. Prove that G contains some edge $(u, v) \in T$ and some edge $(x, y) \notin T$ such that $(T - \{(u, v)\}) \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

c. Now let T be any spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

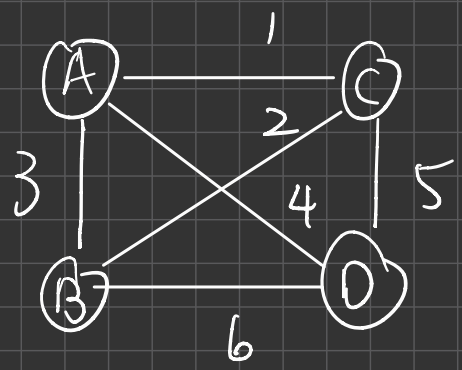
d. Give an efficient algorithm to compute the second-best minimum spanning tree of G.

(a)
pf: MST is unique

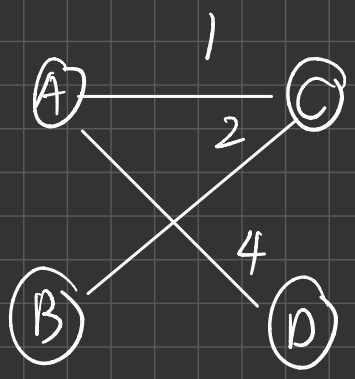
21.1-6

Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

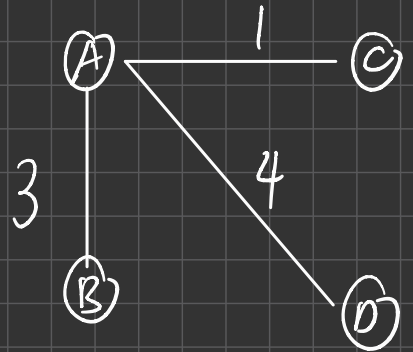
Assuming G has two different MST T and T'.
Let $(u,v) \in T$ and $(u,v) \notin T'$
If we remove (u,v) from T to make T not comected.
By exercise 21.1-3, (u,v) is a unique light edge for some cut $(S, V-S)$. Let (x,y) is the edge crossing $(S, V-S)$ and $(x,y) \in T'$. We have $w(x,y) > w(u,v)$. By Thm. 21.7, we could construct T'' s.t. $w(T'') < w(T')$
$\Rightarrow$ T' is not MST (×) i.e. MST is unique #

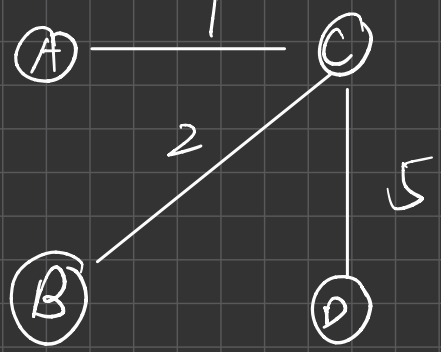


$\Rightarrow$ The MST:



The 2nd best MST:





$\Rightarrow$ 2nd best MST need not be unique

21-1 Second-best minimum spanning tree

Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of G, and let T be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** is a spanning tree T' such that $w(T') = \min\{w(T'') : T'' \in \mathcal{T} - \{T\}\}$.

a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

b. Let T be the minimum spanning tree of G. Prove that G contains some edge $(u, v) \in T$ and some edge $(x, y) \notin T$ such that $(T - \{(u, v)\}) \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

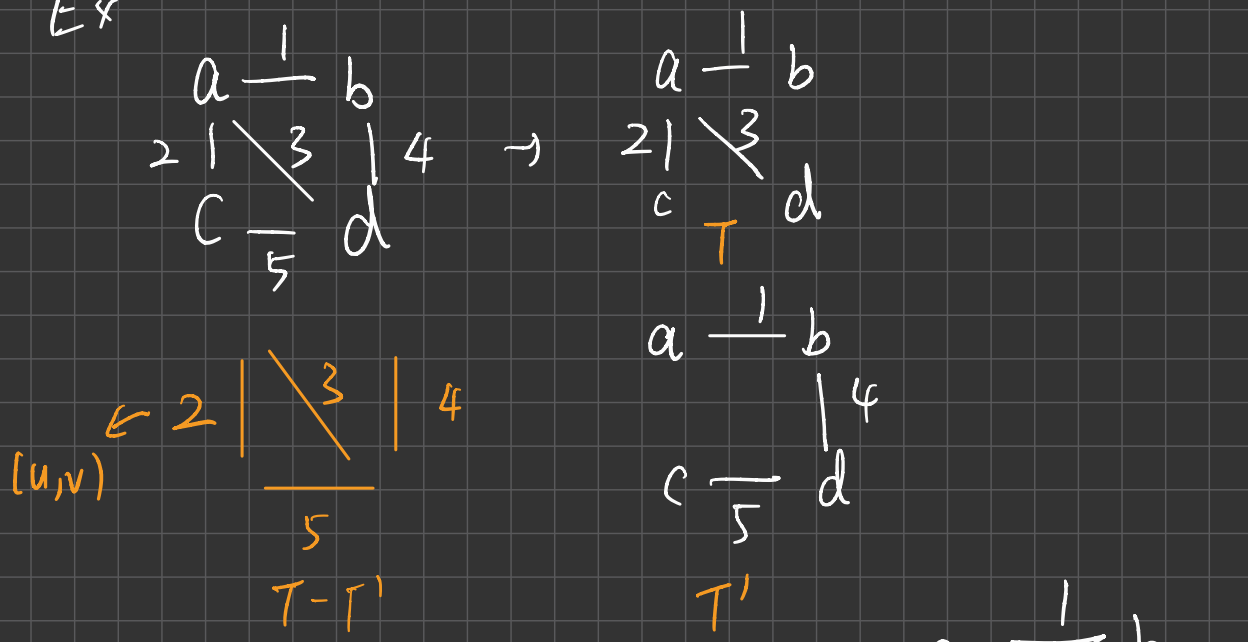
c. Now let T be any spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

d. Give an efficient algorithm to compute the second-best minimum spanning tree of G.

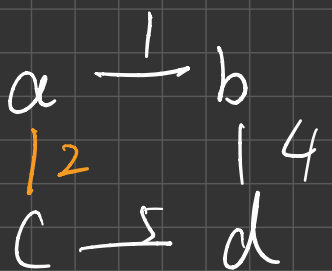
(b)

Let T be the MST of G, suppose T' is the 2nd-best MST which is differene from T by two or more edges. Let (u,v) be the minimum weight edge in $T-T'$. If we add (u,v) to T' we would get a cycle C. This cycle contains some edge (x,y) in $T'-T$

Ex



add (u,v) to T' =>



---

cont.

Claim that $w(x,y) > w(u,v)$

assume $w(x,y) < w(u,v)$, if we add (x,y) to T, we get a cycle C', which contains some edge (u', v') in $T-T'$. The set $T'' = T - \{(u',v')\} \cup \{x,y\}$ forms a spanning tree, we have $w(u',v') < w(x,y)$ $(*)$

cont.

If contradict with (u,v) is the edge with minimum weight in T-T'

So we get w(x,y) > w(u,v), and we could get T''' = T' − {(x,y)} ∪ {(u,v)} which is also a spanning tree. It's not T' nor T. W(T''') < W(T') which yields a better solution than T'. ⇒ T' is not 2nd-best solution #

(T' 有 T 差三條邊，T''' 移除 T' 一條邊加入 (u,v) 絕不是 T or T')

i.e. 2nd-best MST and MST will be different with 1 edge #

c. Now let T be any spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

```
BFS(G, T, w)
  create max[1:|V|][1:|V|]
  for u in G.V
      for v in G.V
          max[u,v] = w(u,v)
  Let Q be a new empty queue
  ENQUEUE(Q,u)
  while Q is not empty
        x = DEQUEUE(Q)
        for v in G.Adj[x]
            if max[u,v] == NULL and v≠u
                if x==u or w(x,v) > w(max[u,x])
                    max[u,v] = (x,v)
                else
                    max[u,v] = max[u,x]
                ENQUEUE(Q,v)
  return max
```

Time Complexity : $O(V^2)$ (fill the max array)

d. Give an efficient algorithm to compute the second-best minimum spanning tree of G.

21-2 Minimum spanning tree in sparse graphs

1. Make MST : $O(|E| + |V| \lg |V|)$ → depending on the Data structure of priority queue.

2. compute max array in (c) : $O(|V|^2)$

3. Find a edge $(u,v) \notin T$ that minimize

$$\underline{w[u,v] - w(max[u,v])} : O(|V|^2)$$

↳ The cost that $cost(T') - cost(T)$

4. Get the $T' = T - \{max[u,v]\} \cup \{u,v\}$

Time complexity : $O(|V|^2)$