2019280373 费杰

2.1-3

Consider the ***searching problem***:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.

Output: An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.

Write pseudocode for ***linear search***, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

```
for i = 1 to A.length
        if A[i] == v
                return i
return NIL
```

At the start of each iteration of the for loop, we are sure that the subarray $A[1 \ldots i-1]$ does not have elements equal to v.

- Initialization: The first element of an array is either v or not v.
- Maintenance: $A[i]$ is either v or not v. If it is v, then we found the solution, if it is not v, then subarray $A[1 \ldots i]$ oes not have elements equal to v, we can go to the next step.
- Termination: When the loop terminates, we either return NIL, because every elements of the array is not equal to v or we return the index I of an element equal to v.

2.2-3

Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

Since the probability of an element being equal to v does not depend on the position of the element in the array and equals to $\frac{1}{n}$, the average number of elements to be checked is

$$\frac{\sum(1+2+\ldots+n)}{n} = \frac{n*(n+1)}{2*n} = \frac{n+1}{2}$$

The worst case is when all elements of the array should be checked, there are n of them.

Based on the written above, for the average case, the running time is $\theta(n)$, for the worst case $\theta(n)$ again.

3.1-3

Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

Because O() notation tells that the running time of algorithm is less than C * n^2, where C is a constant, while at least means that running time of algorithm is more or equal to C * n^2.

4.

$$\lim_{n->\infty} \frac{2n^2+3n+1}{C*n^2} = \lim_{n->\infty} \frac{2n^2}{C*n^2} + \lim_{n->\infty} \frac{3n}{C*n^2} + \lim_{n->\infty} \frac{1}{C*n^2} = \lim_{n->\infty} \frac{2}{C} + 0 + 0 = \frac{2}{C}$$

C.3-1

Suppose we roll two ordinary, 6-sided dice. What is the expectation of the sum of the two values showing? What is the expectation of the maximum of the two values showing?

$$E[X] = 1 * PrX = 1 + 2 * PrX = 2 + \ldots + 12 * PrX = 12 =$$

$$= 1*0 + 2*\frac{1}{36} + 3*\frac{2}{36} + 4*\frac{3}{36} + 5*\frac{4}{36} + 6*\frac{5}{36} + 7*\frac{6}{36} + 8*\frac{5}{36} + 9*\frac{4}{36} + 10*\frac{3}{36} + 11*\frac{2}{36} + 12*\frac{1}{36} = 7$$

$$E[X] = \sum_{x=1}^{6} (x * P(X = x)) = \sum_{x=1}^{6} (x * (P(X_1 = x, X_2 \le x) + P(X_2 = x, X_1 < x)) = \sum_{x=1}^{6} (x * (\frac{1}{6} * \frac{x}{6} + \frac{1}{6} * \frac{x-1}{6})) = \frac{161}{36}$$

C.3-2

An array $A[1 \ldots n]$ contains n distinct numbers that are randomly ordered, with each permutation of the n numbers being equally likely. What is the expectation of the index of the maximum element in the array? What is the expectation of the index of the minimum element in the array?

The Probability of max element index equal to i is P(X = i) = 1/n,

$$E[X] = \sum_{i=1}^{n} (i * P(X = i)) = \frac{n*(n+1)}{2*n} = \frac{n+1}{2}$$

Same answer for the minimum.

6.

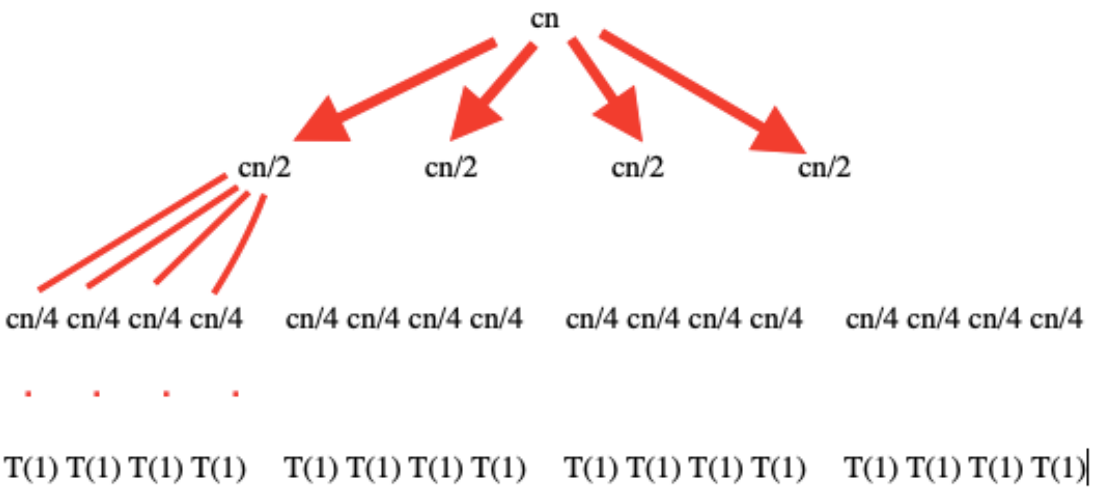
Let the height of complete 3-ary tree with n nodes is equal to h.

Then $n = 1 + 3 + \ldots + 3^{h-1} = \frac{3^h - 1}{3-1}; 2n = 3^h - 1; h = \log_3(2n+1)$

Student id: 2019280373 Name: 费杰

4.4-7

Draw the recursion tree for $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, where c is a constant, and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.



Depth is $\lg n$.

Total time: $\sum_{i=0}^{\lg n-1} 2^i cn + cn = cn * \frac{2^{\lg n} - 1}{2 - 1} + cn = \theta(n^2)$

4-1 Recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(n/2) + n^4$.

b. $T(n) = T(7n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-2) + n^2$.

We use the following theorem:

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

So, the answers are:
b) $b = \frac{10}{7}$, $\log_{\frac{10}{7}} 1 = 0$, $n = \Omega(n^{\epsilon})$, $\epsilon \leq 1 \Rightarrow T(n) = \theta(n)$
c) $\log_4 16 = 2$, $n^2 = \theta(n^2) \Rightarrow T(n) = \theta(n^2 \lg n)$
d) $n^2 = \Omega(n^{\log_3 7 + \epsilon})$, $\log_3 7 < 2 \Rightarrow T(n) = \theta(n^2)$

4-2 Parameter-passing costs
Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time $= \Theta(1)$.
2. An array is passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(q - p + 1)$ if the subarray $A[p \,..\, q]$ is passed.

a. Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.

Use Master theorem (except second, for which we can draw a recursion tree):
1. $T(n) = T(n/2) + \theta(1) \Rightarrow T(n) = \theta(\lg(n))$

2. $T(n) = T(n/2) + cN = T(n/4) + 2cN = \ldots = \sum_{i=0}^{\lg n - 1} cN = cN \lg n \Rightarrow T(n) = \theta(n \lg(n))$

3. $T(n) = T(n/2) + \theta(n) \Rightarrow T(n) = \theta(n)$

2:

cN The depth of the tree is $\lg n$
|
cN
|
cN
.
.
.
cN

Student id: 2019280373
Name: 费杰

5.2-3

Use indicator random variables to compute the expected value of the sum of n dice.

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1+2+3+4+5+6}{6} = 3.5n$$

5.2-5

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A. (See Problem 2-4 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

Let i, j be the positions of two elements, P(i,j) be the probability of inversion. There are C_n^2 pairs in the array, and for each of the pairs the probability of $A[i] > A[j]$ is $\frac{1}{n(n-1)}$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{n*(n-1)}{2*n(n-1)} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} (n-i) = \frac{(n-1+1)*(n-1)}{2*2} = \frac{n(n-1)}{4}$$

5-2 Searching an unsorted array

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If $A[i] = x$, then we terminate; otherwise, we continue the search by picking a new random index into A. We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A. Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

```
Random-Search(A, n, x)
A_picked[n] = Zeros
num_picked = 0
j = random(1, n)
While (A[j] != x || num_picked < n)
        j = random(1, n)
        num_picked += (A_picked[j] == 0) ? 1 : 0
        A_picked[j] = 1
Return (num_picked == n) ? Nan : j
```

b. Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates?

Bins are the elements of the array A_picked, while balls are the indexes j . How many balls must we toss on average until the bin A[i] contains a ball? Using this model, we get the answer: n.

c. Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

Same as in previous question, the answer is n/k.

d. Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we have checked all elements of A and RANDOM-SEARCH terminates?

Bins are the elements of the array A_picked, while balls are the indexes j . The balls must be tossed to every of the bins at least once. Using this model, we get the answer: $n(ln(n) + O(1))$

Now consider a deterministic linear search algorithm, which we refer to as DETERMINISTIC-SEARCH. Specifically, the algorithm searches A for x in order, considering $A[1], A[2], A[3], \ldots, A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

e. Suppose that there is exactly one index i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

The average case is $\sum_{i=1}^{n} \frac{i}{n} = (1+n)/2$, the worst case is n (search every element).

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
9	19	13	5	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	8	19	12	13	7	4	21	2	6	11
9	5	8	7	12	13	19	4	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	2	19	12	21	13	6	11
9	5	8	7	4	2	6	12	21	13	19	11
9	5	8	7	4	2	6	11	21	13	19	12

8.4-4 ★

We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

Probability of a point being in each of the buckets must be the same. The area of the unit circle is $\pi * 1^2 = \pi$. Since we need to have n buckets, we must divide circle to n parts with area $\frac{\pi}{n}$.

Also each of the buckets must have different area from center.
The idea of dividing the circle is in dividing the circle to n rings.
Each of the rings has specific radius (inner and outer). Let the i^{th} ring have radiuses a_i and a_{i+1}.

Then $a_0 = 0; a_{n+1} = 1$. Let's prove that $a_i = \frac{\sqrt{i}}{\sqrt{n}}$ by induction.

Base is obvious.
Assuming that the area of the disks k and k + 1 are the same (0 < k + 1 < n), we get:

$$\pi(a_{k+1}^2 - a_k^2) = \pi(a_k^2 - a_{k-1}^2)a_{k+1} = \sqrt{2a_k^2 - a_{k-1}^2} = \sqrt{\frac{2k}{n} - \frac{k-1}{n}} = \frac{\sqrt{k+1}}{\sqrt{n}}$$

So the point with a distance d is in the bucket number $\lfloor d^2 n \rfloor + 1$

15.2-1

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

Matrix	A1	A2	A3	A4	A5	A6
Dimension	5 x 10	10 x 3	3 x 12	12 x 5	5 x 50	50 x 6

p = <5, 10, 3, 12, 5 , 50, 6>

Following (15.7) m[i, i] = 0, i = 1..6

$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j)$, i < j

Put the values into the table for m[i,j] and for s[i, j]:

0	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

0	1	2	3	4	5	6
1	0	1	2	2	4	2
2		0	2	2	2	2
3			0	3	4	4
4				0	4	4
5					0	5
6						0

$m[1,2] = m[1,1] + m[2,2] + p_0 p_1 p_2 = 0 + 0 + 5 * 10 * 3 = 150$

m[2,3] = 10*3*12 = 360

m[3,4] = 3 * 12 * 5 = 180

m[4,5] = 12 * 5 * 50 = 3000

m[5, 6] = 5 * 50 * 6 = 1500

m[1,3] = min(0 + 360 + 5 * 10 * 12, 150 + 0 + 5 * 3 * 12) = min(960, 330) = 330

m[2,4] = min(0 + 180 + 150, 360 + 0 + 180) = 330

m[1,4] = min(0 + 330 + 250, 150 + 180 + 75, 330 + 0 + 300) = 405

...

m[1,6] = min(2250, 2010, 2550, 2055, 3155) = 2010

According to the table, s[1,6] = 2, s[3,6] = 4

$$A_{1..6} = (A_1A_2)((A_3A_4)(A_5A_6))$$

Code for this task:

```
void Ans(int i, int j) {
    if (i == j) cout << i << " ";
    else
    {
        cout << "(";
        Ans(i, s[i][j]);
        Ans(s[i][j] + 1 , j);
        cout << ")";
    }
}

int main() {
    int p[7] = {5, 10, 3, 12, 5, 50, 6};
    for (int i = 0; i < 6; i++) {
        m[i][i] = 0;
    }
    int j;
    int q;
    for (int l = 2; l < 7; l++) {
        for (int i = 0; i < 6 - l + 2; i++) {
            j = i + l - 1;
            m[i][j] = 1000000;
            for (int k = i; k < j; k++) {
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
    Ans(1, 6);
    return 0;
}
```

15-3-3

Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize,

the number of scalar multiplications. Does this problem exhibit optimal substructure?

"The problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems."

First we note that subproblems' solutions are not connected: each of the parenthesization splits the solution to two subchains, which is the sequence of matrices with so as to maximize the number of scalar multiplications.

Let one of the subchains we got provide not an optimal solution. Since those two subchains got on this level of parenthesization are not affected by each other, we can change the solution for this particular subchain, which will provide us a better solution for the whole chain.

15.4-4

Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

We don't need the previous rows of the c table on each step. We need only c[i - 1, j - 1], c[i, j - 1], c[i - 1, j] - so we need to store only the (i -1)-st row and i-th row of the c table, i.e. 2min(m,n) of entries.

To use only one row, we put c[i - 1, j - 1] into an extra variable (Let it be a). Since c[i, j -1] is already computed, we only need to update a = c[i -1, j - 1] (= c[i - 1, j] on the next step)

25.2-4

As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL'(W)
1  n = W.rows
2  D = W
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              d_ij = min(d_ij, d_ik + d_kj)
7  return D
```

If we improved (decreased) some value in the distance matrix, we could not thereby degrade the shortest path length for any other pairs of vertices processed later.

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

It means that we don’t need to create the matrix on each step, all of the values are stored in one matrix d.

Student id 2019280373

Name 费杰

16.1-2

Suppose that instead of always selecting the first activity to finish, we 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.

Let the set of activities $S = \{a_1, \ldots a_n\}$, where each a_i is defined by starting time s_i and finishing time f_i.
Let's construct a new set $S' = a'_1, \ldots a'_n$ using bijective mapping:
$a_i <-> a'_i$
$a_i = [s_i, f_i) <-> a'_i = [f_i, s_i)$
Firstly, subset of S is mutually compatible if and only if corresponding subset of S' is mutually compatible.
As in the original algorithm, we select the first activity a'_m to finish that is compatible with all previously selected activities. Since $a'_m <-> a_m$, we select activity a_m in the set S.
Writing down the sequence of the activities, and following Theorem 16.1, we get that a'_m is included in some maximum-size subset of mutually compatible activities of S'_k, so a_m is included in some maximum-size subset of mutually compatible activities of S_k, so we get optimal solution for S', which leads us to the optimal solution for S as well.
We use the same greedy algorithm for S', and using one-to-one correspondence we get the greedy algorithm for S as well.

16-2 Scheduling to minimize average completion time

Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} c_i$. For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$. If task a_1 runs first, however, then $c_1 = 3$, $c_2 = 8$, and the average completion time is $(3 + 8)/2 = 5.5$.

a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Our algorithm is greedy:
We first run the task with the smallest processing time, then the task with the second smallest processing time and so on, while all of the tasks are not finished running.
In other words:
1. Let a = sorted_array(S)
2. let spent_time = 0, answer = 0
3. for i from 1 to n:
    spent_time += a_i
    answer += spent_time

return (answer / n)
The running time of this algorithm is O(n * lgn + n) = O(n * lgn)

Let's prove that greedy solution is optimal.
Suppose we have an optimal solution. We change the first event of this solution with the smallest in the sequence of events. Since all the other events stay in the initial position, the total time is reduced by the difference of these events multiplied by the number of events lying between these two, so the average time also becomes smaller.

23.2-8

Unfortunately, professor Borden failed, and his algorithm failed too.

Suppose we have only three vertices: v_1, v_2 and v_3.

	v1	v2	v3
v1	-	1	2
v2		-	10
v3			-

In Borden algorithm it is said vertices must be partitioned into two sets $V_1, V_2, ||V_1| - |V_2|| \leq 1$. Here we choose v_1 to put into V_1, and choose v_2, v_3 to put into V_2.
In the second subgraph $G_2 = (V_2, E_2)$ the edge of weight (v_2, v_3) of weight 10 is chosen. We can choose between (v_1, v_2) or (v_1, v_3) connecting two spanning trees into one, the minimum-weight edge is (v_1, v_2). The total weight is 11.

But the minimum spanning tree is $((v_1, v_2), (v_1, v_3))$ with the total weight of 3 < 11.