AMA2222 Principles of Programming

Leung Man Kin, Adam
Instructor

adam.leung@polyu.edu.hk
TU720

Chapter 8: Algorithm II
recursion, divide and conquer,
merge sort, dynamic programming

Recursion

When a function calls itself, it is making a **recursive call**. Process using such function is called a **recursion**.

The cases for which an answer is explicitly known is called the **base case**. The case for which the solution is expressed in terms of a smaller version of itself is called the **recursive or general case**.

Consider the following example: factorial.

One may consider $n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$
and use a for-loop to compute.

What about using recursion instead?

Using recursion, consider for a natural number n, we have:

Recursive case:
$n! = n \times (n-1)!$

Base case:
$0! = 1.$

fac: Oh, what is 3 factorial?
Let me recall what is 2
factorial and multiply by 3.

call

return 6

fac: Oh, what is 2 factorial?
Let me recall what is 1
factorial and multiply by 2.

call

return 2

fac: Oh, what is 1 factorial?
Let me recall what is 0
factorial and multiply by 1.

call

return 1

fac: Aha, 0 factorial
is just 1

return 1

Example program 8.1
Calculate factorial by recursion

```
 1 //8.1 factorial
 2 #include <iostream>
 3 using namespace std;
 4 int fac(int n)
 5 {
 6           if (n==0) return 1;
 7           else return n*fac(n-1);
 8 }
 9 int main()
10 {
11           int num;
12           cout << "Input the value of n: ";
13           cin >> num;
14           cout << "The factorial of " << num << " is " << fac(num) <<endl;
15           return 0;
16 }
```

In the previous example, each recursive case only calls the function for once. So you may consider writing a for-loop might be more convenient.

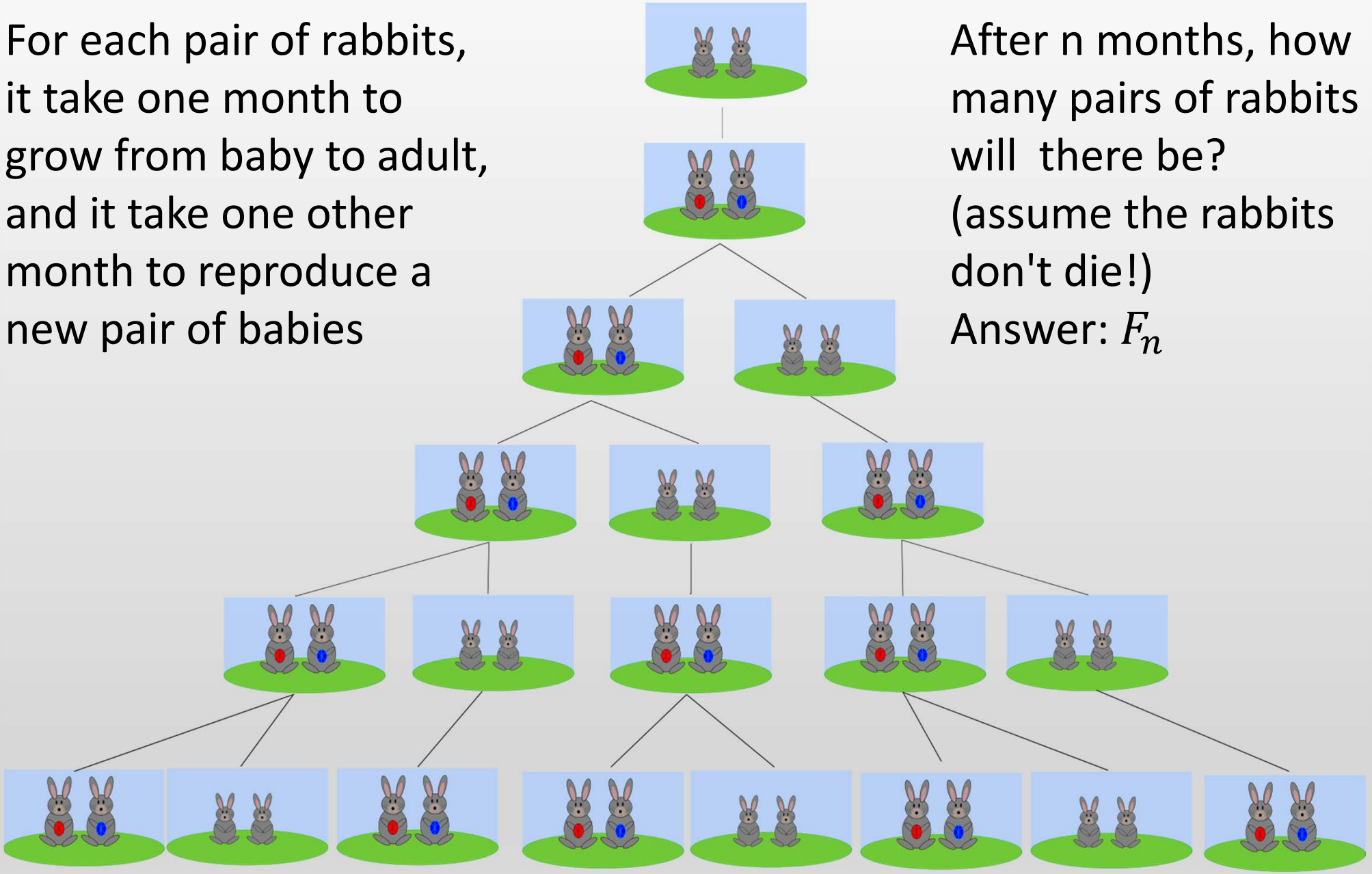
However, some problems might require the recursive case to call the function for more than once. Such problem can hardly be written using for-loop.

One famous example is know is Fibonacci sequence:

$F_0 = 1, F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

We have 1, 1, 2, 3, 5, 8, 13, … every number is the sum of the two consecutive numbers in front.

For each pair of rabbits, it take one month to grow from baby to adult, and it take one other month to reproduce a new pair of babies



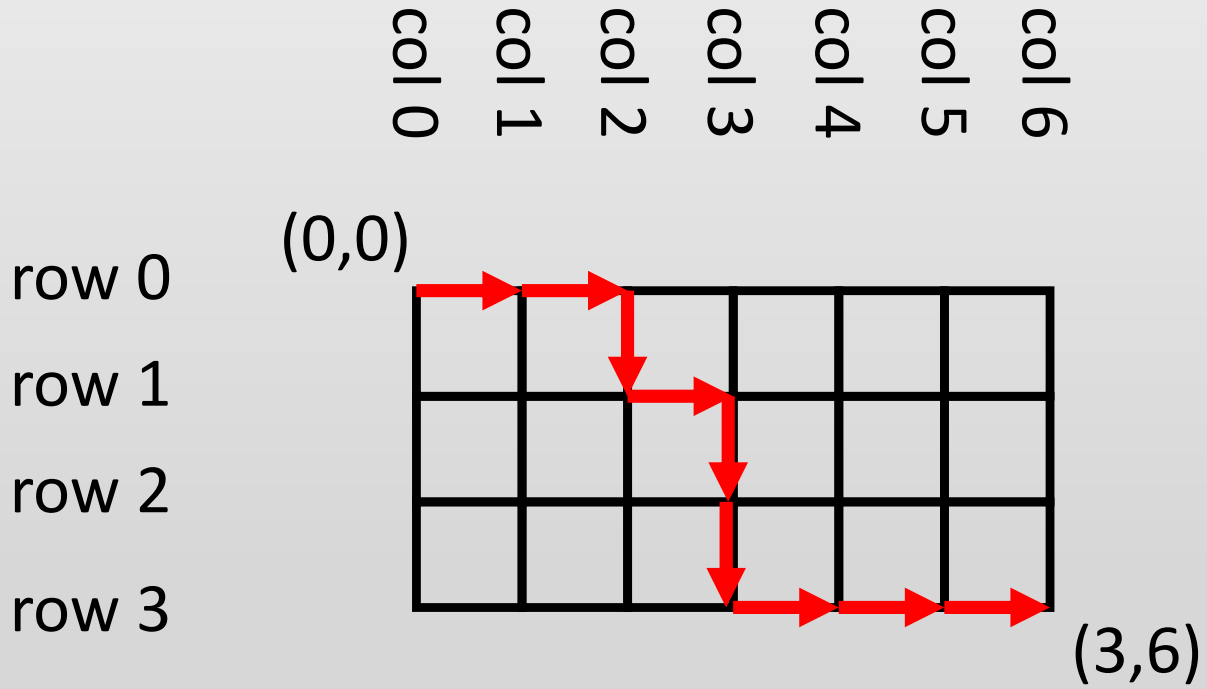
After n months, how many pairs of rabbits will there be? (assume the rabbits don't die!)
Answer: F_n

Example program 8.2

Write a program to evaluate the n-th Fibonacci number.

```
 1 //8.2 Fibonacci number
 2 #include <iostream>
 3 using namespace std;
 4 int fib(int n)
 5 {
 6          if ((n==0)||(n==1)) return 1;
 7          else return fib(n-1)+fib(n-2);
 8 }
 9 int main()
10 {
11          int n;
12          cout << "Input a number: ";
13          cin >> n;
14          cout << "The " << n << "-th Fibonacci number is "   << fib(n) <<endl;
15          return 0;
16 }
```

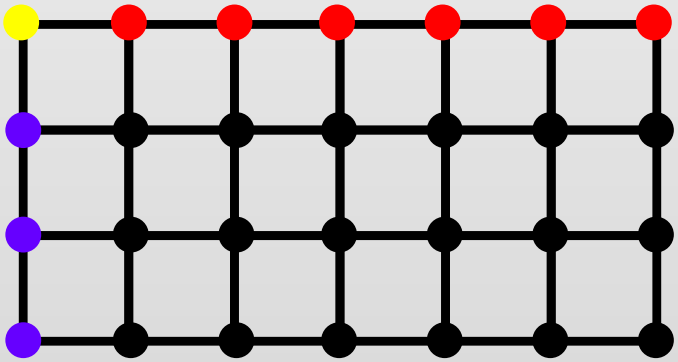
Recall a combinatoric problem. Each vertex and edge below represents a junction and a road respectively. If we are only allowed to go either east or south at each junction, how many possible ways are there to go from the starting point (0,0) to the ending point (3,6)?



Let f(m,n) be a function that gives the number of possible ways going from the starting point to point (m,n).

Base case (0,0):
only one way, which is stand and don't move

Base case (0,n):
only one way, going straight east from the starting point



Base case (m,0):
only one way, going straight south from the starting point

General case (m,n):
either go south from (m-1,n), or go east from (m,n-1).
Add up these two numbers of ways.

Example program 8.3

Write a program to evaluate the number of ways going from (0,0) to (m,n)

```
1  //8.3 Number of ways
2  #include <iostream>
3  using namespace std;
4  int way(int m, int n)
5  {
6           if ((m==0)||(n==0)) return 1;
7           else return way(m-1,n)+way(m,n-1);
8  }
9  int main()
10 {
11          int m,n;
12          cout << "Input the row number: ";
13          cin >> m;
14          cout << "Input the column number: ";
15          cin >> n;
16          cout << "The number of ways going to (" << m << ","
17 << n << ") is " << way(m,n) <<endl;
18          return 0;
19 }
```

Divide and conquer

Based on the idea of recursion, we can develop the concept of divide-and-conquer, where a problem is solved recursively applying three steps at each level of the recursion:

(i) Divide the problem into a number of subproblems that are smaller instances of the same problem.

(ii) Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

(iii) Combine the solutions to the subproblems into the solution for the original problem.

To illustrate the idea of divide-and-conquer and these three steps, we will use the following problem.

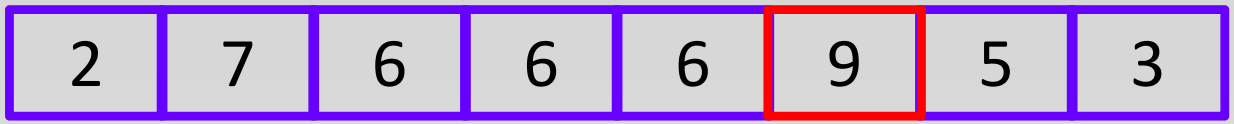
Example 8.4 Finding maximum value

Write a function maxv() which inputs an array of integers and its size, then return the maximum element in this array.

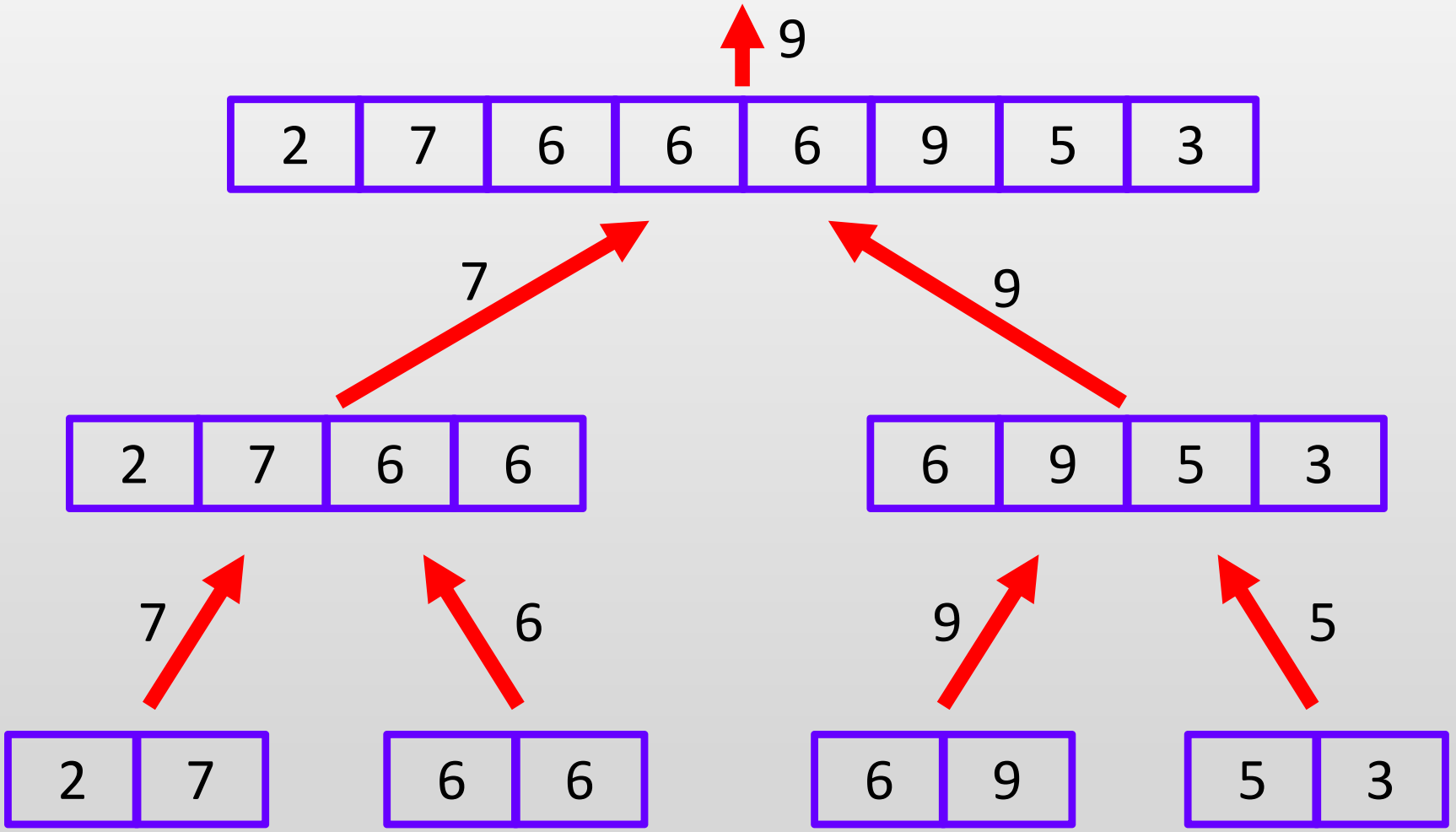
For example, consider the following array:

```
int arr[] = {2, 7, 6, 6, 6, 9, 5, 3};
```

Then `maxv(arr,8)` should give return value 9.

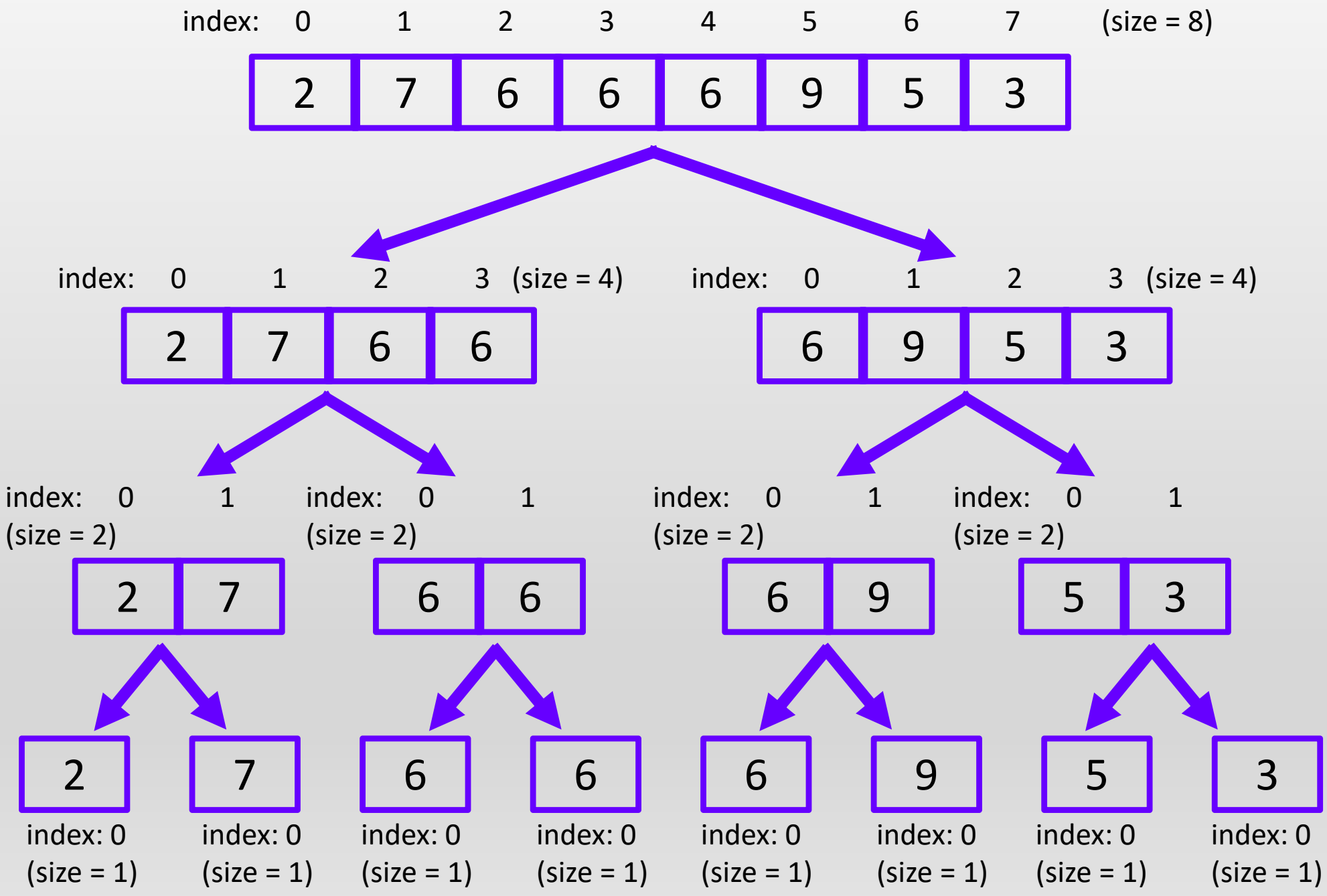


The idea is to divide the array into two subarrays of half the size, and take the maximum between the two subarray maximum values.

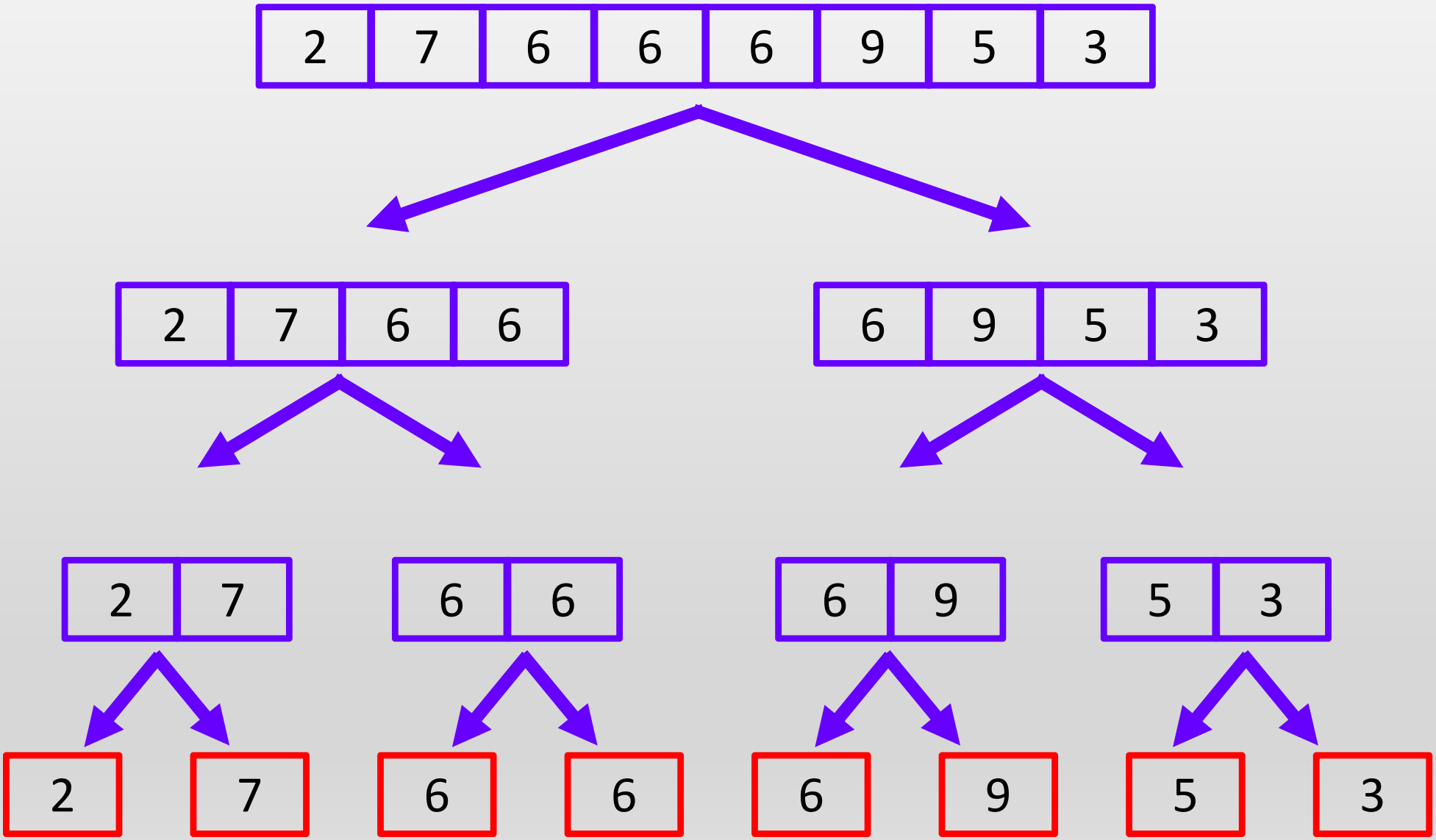


If we further divide the subarray until they have only one element, then it is automatically the maximum value of the subarray.

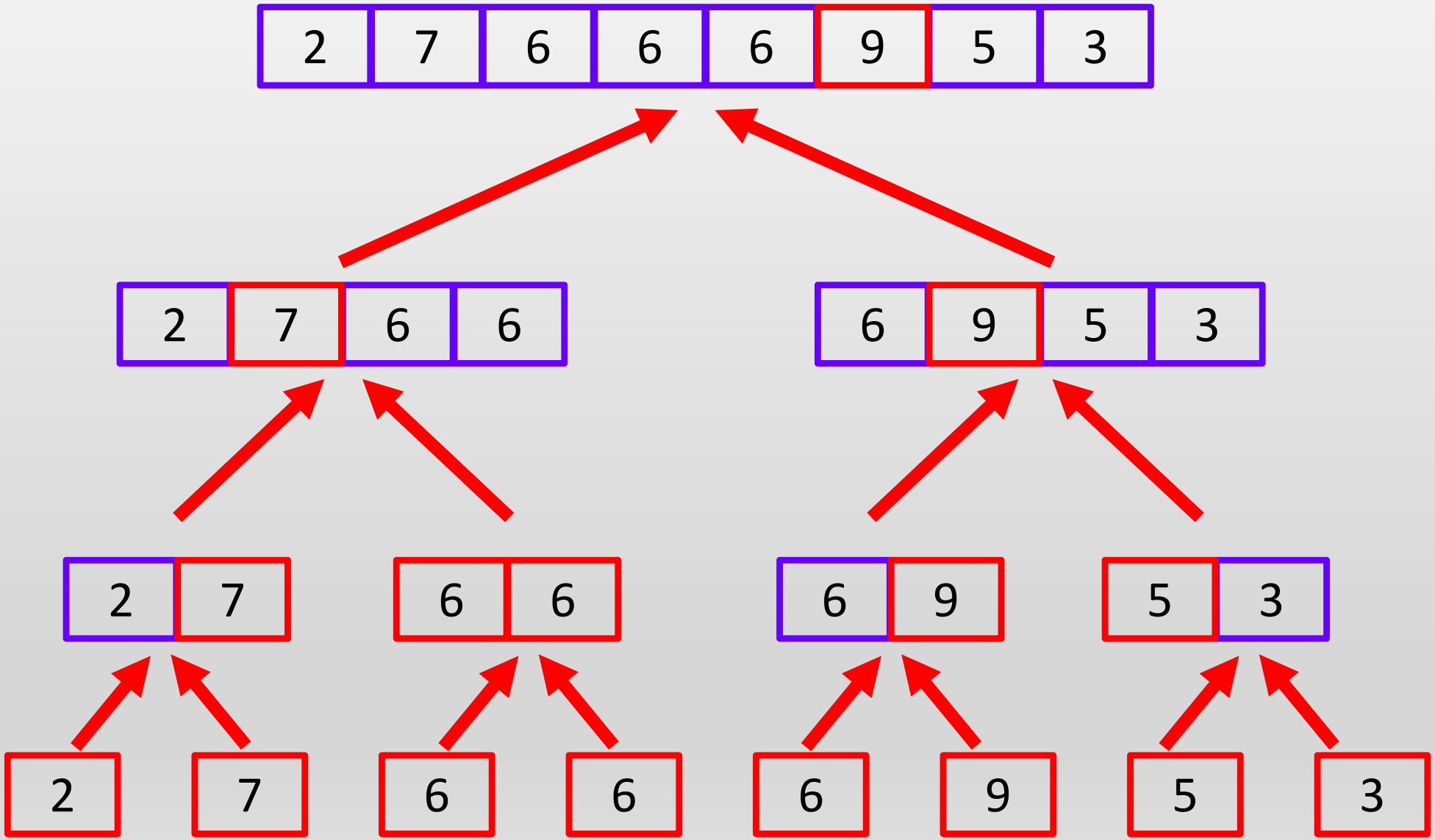
(i) divide : split the array into two subarrays from the index s/2



(ii) conquer : if the subarray become size 1, it contains only one single element which is directly the maximum value.



(iii) combine : combine the maximum values of the two subarrays underneath and pick the greater one using `max( )` operator.



Example program 8.4 maximum value of array

```
1  //example 8.4 maximum value of array
2  #include <iostream>
3  using namespace std;
4  int maxv(int a[], int s)
5  {
6          if (s==1) return a[0];
7          int b[s/2];
8          int c[s-s/2];
9          for (int i=0; i<s/2; i++)
10                 b[i]=a[i];
11         for (int i=s/2; i<s; i++)
12                 c[i-s/2]=a[i];
13         return max(maxv(b,s/2),maxv(c,s-s/2));
14 }
15 int main ()
16 {
17         int arr[] = {2,7,6,6,6,9,5,3};
18         cout << "The maximum value is " << maxv(arr,8);
19         return 0;
20 }
```

(conquer) the case with a single element in the subarray.

(divide) create two subarrays with half the original size and copy the elements accordingly

(combine) return the greater value among the maximum values of the two subarrays

In the previous chapter, we have learnt some sorting algorithm. Next we will introduce **merge sort** making use of the idea of divide and conquer.

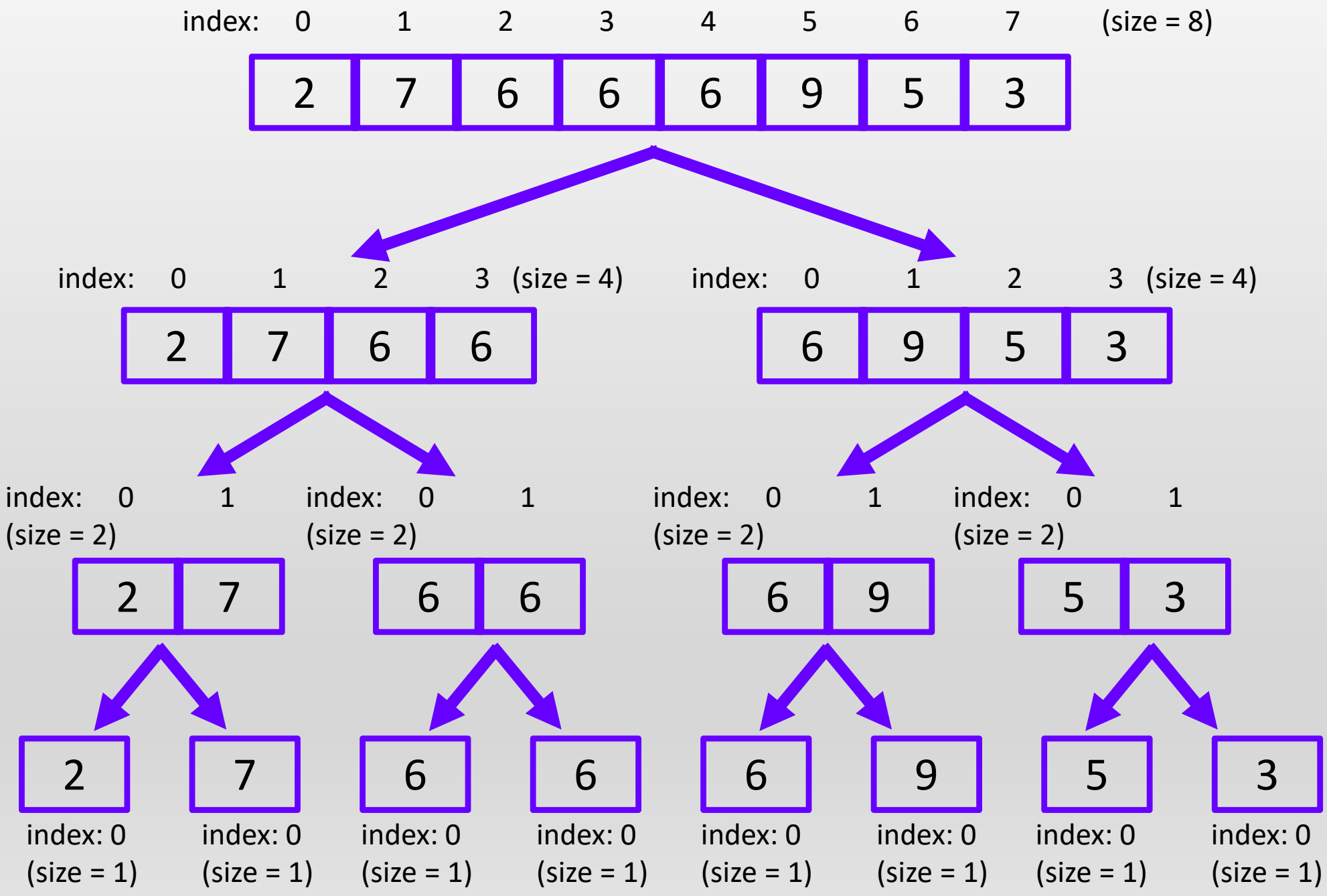
In merge sort, an array is split into two subarrays for doing merge sort. The sorted result will be merged together, hence resulting a sorted array. Notice that if a subarray contains one single element, it is automatically a sorted array.

For example, consider the following array:

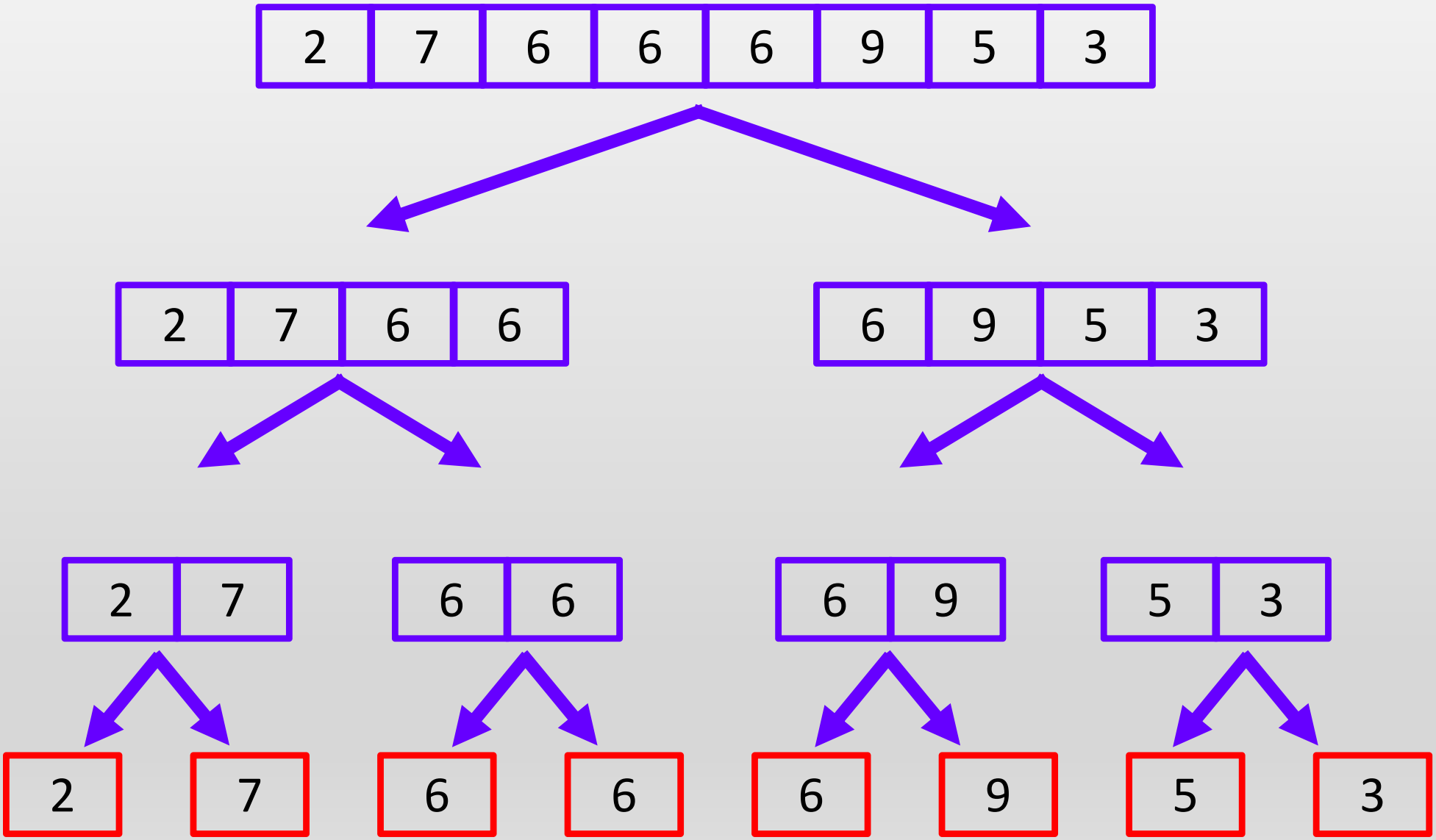
```
int arr[] = {2, 7, 6, 6, 6, 9, 5, 3};
```

After executing `mergeSort(arr,8)` the array should become `{2, 3, 5, 6, 6, 6, 7, 9}`.

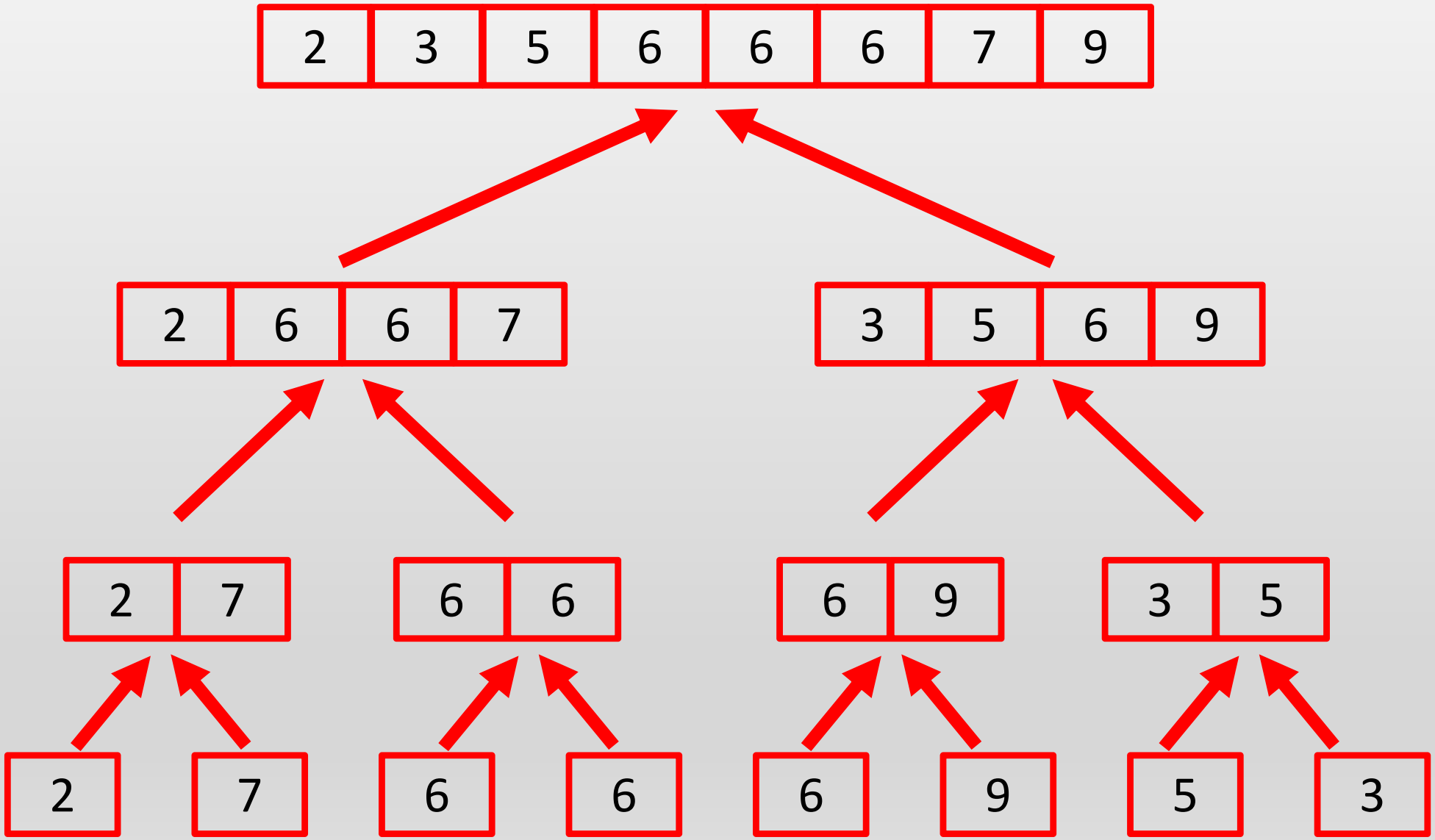
(i) divide : split the array into two subarrays from the index s/2



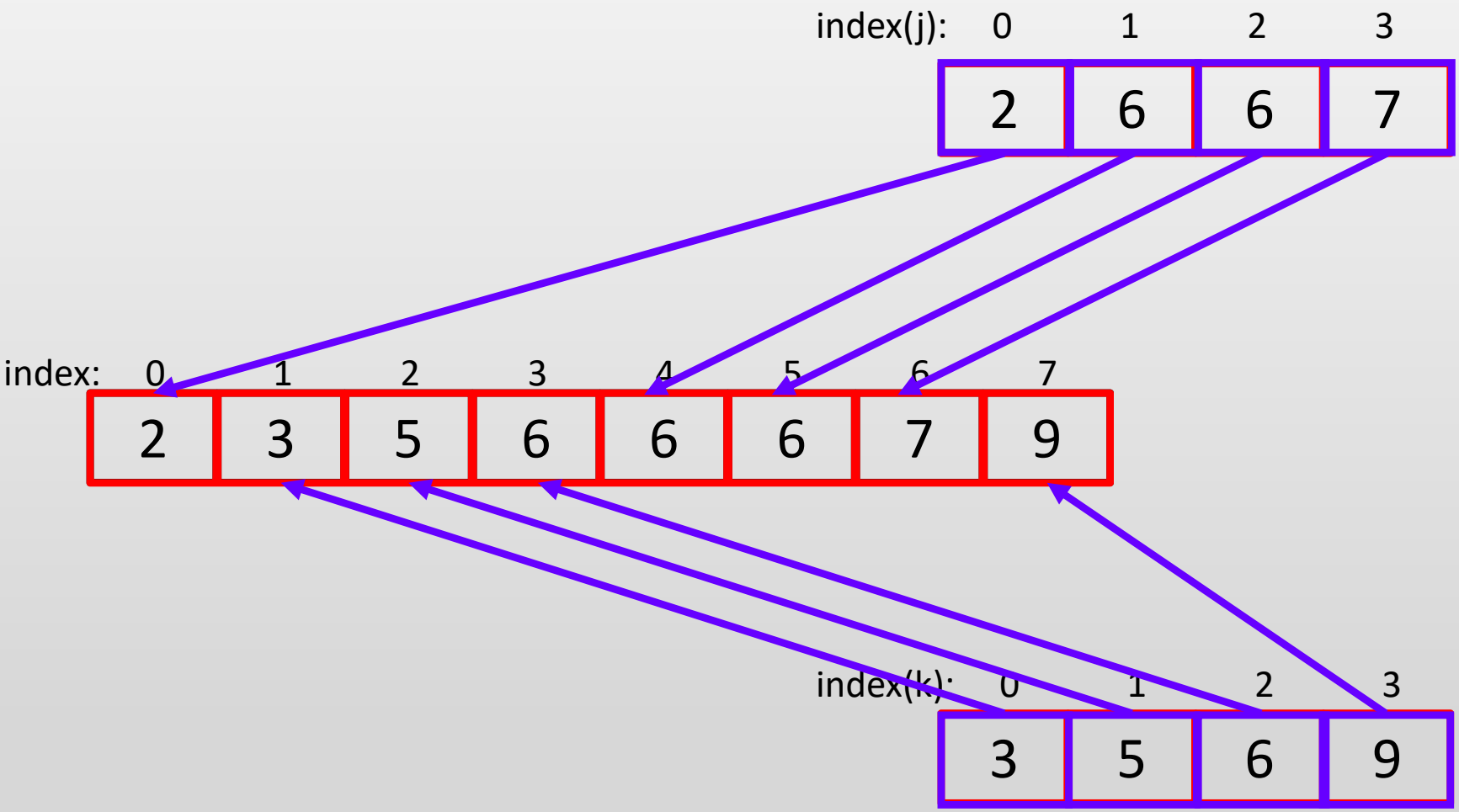
(ii) conquer : if the subarray become size 1, it contains only one single element which is already sorted in ascending order.



(iii) combine : merge the two sorted subarrays together into a combined array in ascending order.



To illustrate the merge process, we will consider the very last step which merges two sorted subarrays of size 4.



Example 8.5 merge sort

```
1  //example 8.5 merge sort
2  #include <iostream>
3  using namespace std;
4  void mergeSort(int a[], int s)
5  {
6          if (s==1) return;
7          int b[s/2];
8          int c[s-s/2];
9          for (int i=0; i<s/2; i++)
10                 b[i]=a[i];
11         for (int i=s/2; i<s; i++)
12                 c[i-s/2]=a[i];
13         mergeSort(b,s/2);
14         mergeSort(c,s-s/2);
```

(conquer) the case with a single element in the subarray is already sorted, just terminate the function.

(divide) create two subarrays with half the original size and copy the elements accordingly

(combine) before we do the merging, use merge sort to sort the two subarrays first

```
15          int i=0, j=0, k=0;
16          while (i<s && j<s/2 && k<s-s/2){
17                  if (b[j]<c[k]){
18                          a[i]=b[j];
19                          j++;
20                  }
21                  else {
22                          a[i]=c[k];
23                          k++;
24                  }
25                  i++;
26          }
27          while (i<s && j==s/2){
28                  a[i]=c[k];
29                  k++;
30                  i++;
31          }
32          while (i<s && k==s-s/2){
33                  a[i]=b[j];
34                  j++;
35                  i++;
36          }
37  }
```

use i, j, k as the indexes of a[], b[], c[] respectively

as long as there are elements remaining in b[] and c[], put the smaller one to a[i]

if b[] has no more elements left, put all remaining elements of c[] into a[].

if c[] has no more elements left, put all remaining elements of b[] into a[].

```
38 int main()
39 {
40          int arr[] = {2,7,6,6,6,9,5,3};
41          mergeSort(arr,8);
42          for (int i=0; i<8; i++)
43                    cout << arr[i] << " ";
44          return 0;
45 }
```

Dynamic programming

Dynamic programming is a method which solves problems by combining the solutions to subproblems. Notice that "programming" in this context refers to a tabular method (eg linear programming), not to writing computer code.

As we have learnt, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

In contrast, dynamic programming applies when the subproblems overlap. Such algorithm solves each sub-subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each sub-subproblem.

Dynamic programming can be applied to optimization problems. Such problems can have many possible solutions, each with a corresponding value. We wish to find a solution with the optimal (minimum or maximum) value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Example 8.6 Suppose you have 60 minutes in an exam to work on these four problems. The marks and expected time needed is given by the table below. Which question(s) should you try to complete in order to maximize your total marks?

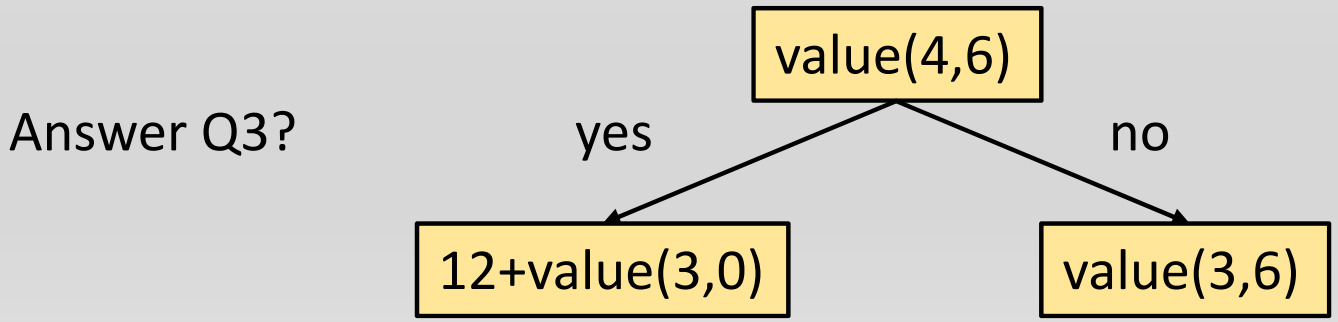
Question number	0	1	2	3
Marks	7	9	5	12
Time needed (unit: 10 min)	3	4	2	6

In this question, marks represent the value of each item and time is a constraint. This problem can be converted to other context. For example, maximizing the total value of items of different sizes put in a **knapsack** with limited capacity.

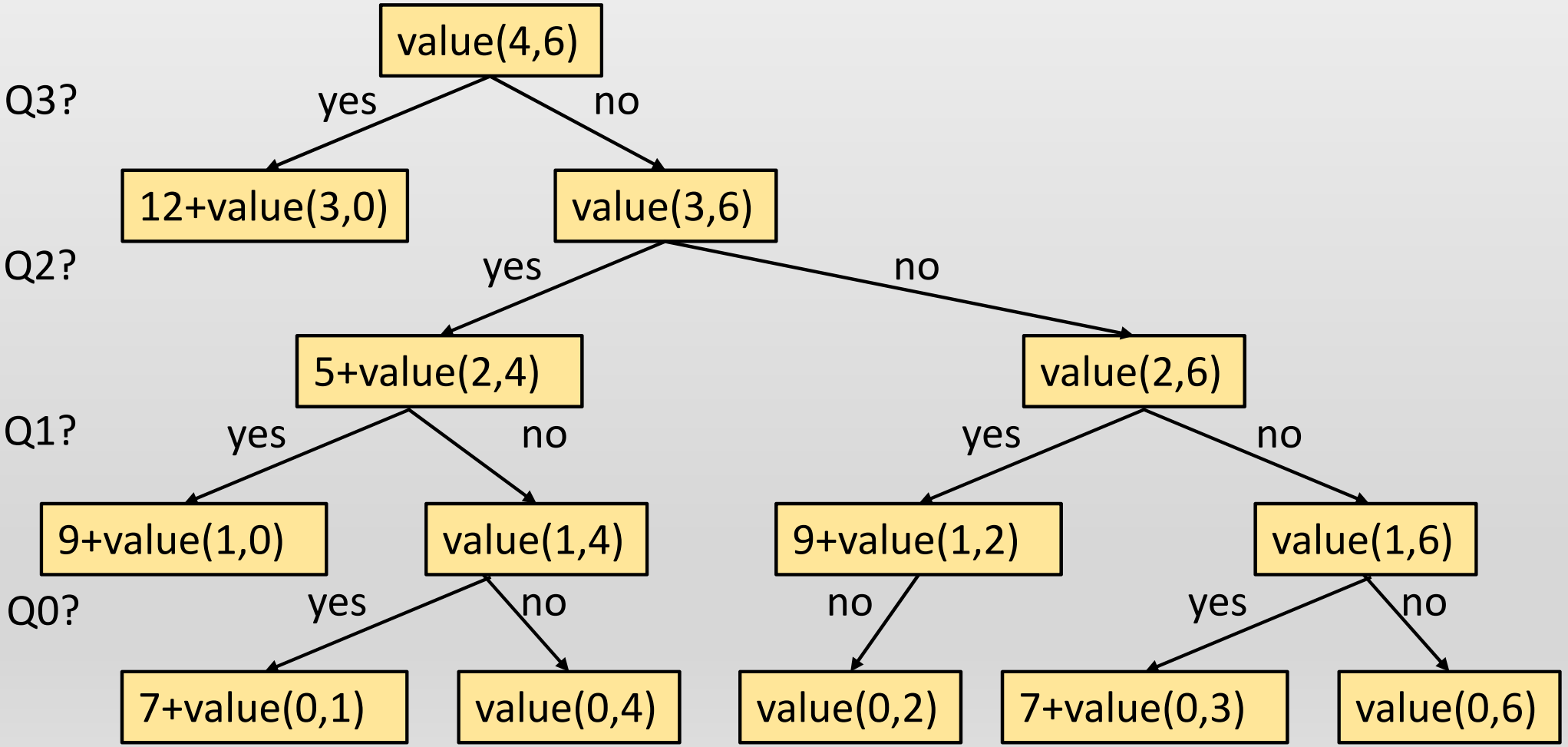
Let `value(n,t)` be a function which represents the maximum total value with n items available for choosing and time t remaining. The problem asks for `value(4,6)`.

Notice that `value(n,0)` and `value(0,t)` must give zero since there are no time left (t=0) and no items left (n=0).

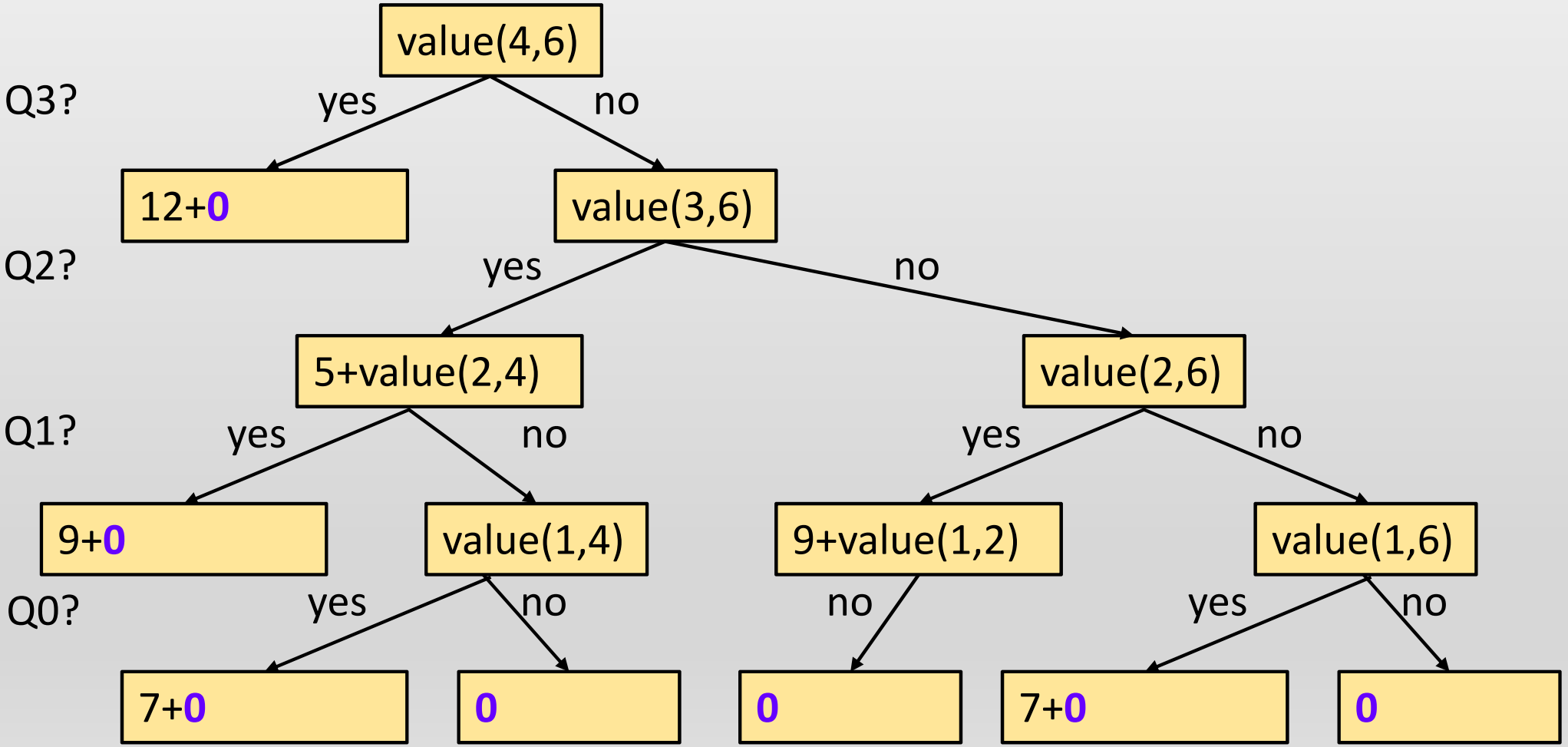
Starting from the last item, we can consider whether we want this item or not. If yes, we can earn 12 marks and have 0 minutes left. If no, we still have 60 minutes left. No matter yes or no, the number of questions left becomes 3.



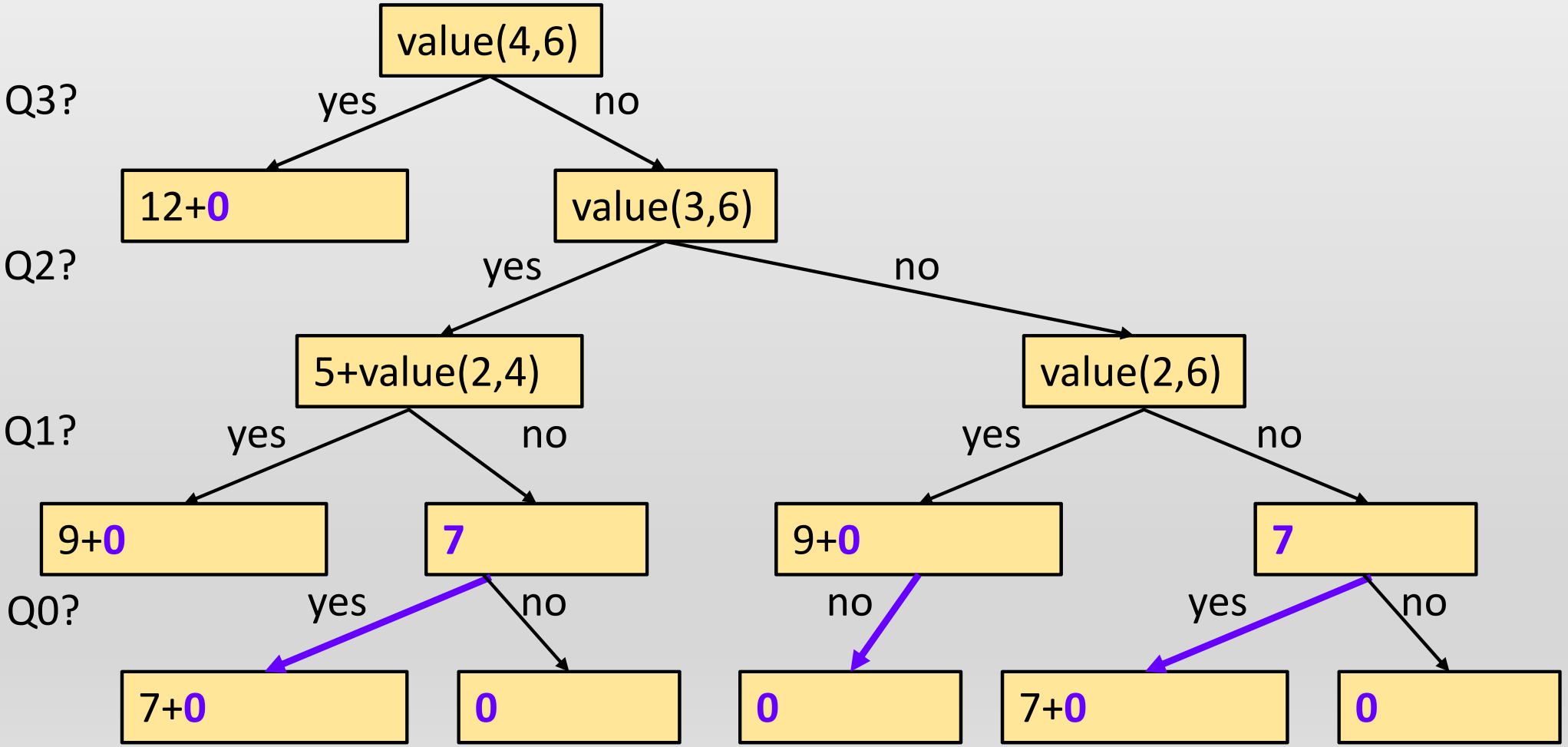
The results of `value(n,t)` follows a recursive relation which can be represented in the tree diagram below. Notice that the node terminates when either n=0 or t=0. Also, if the time needed exceeds t then we are unable to include such item.



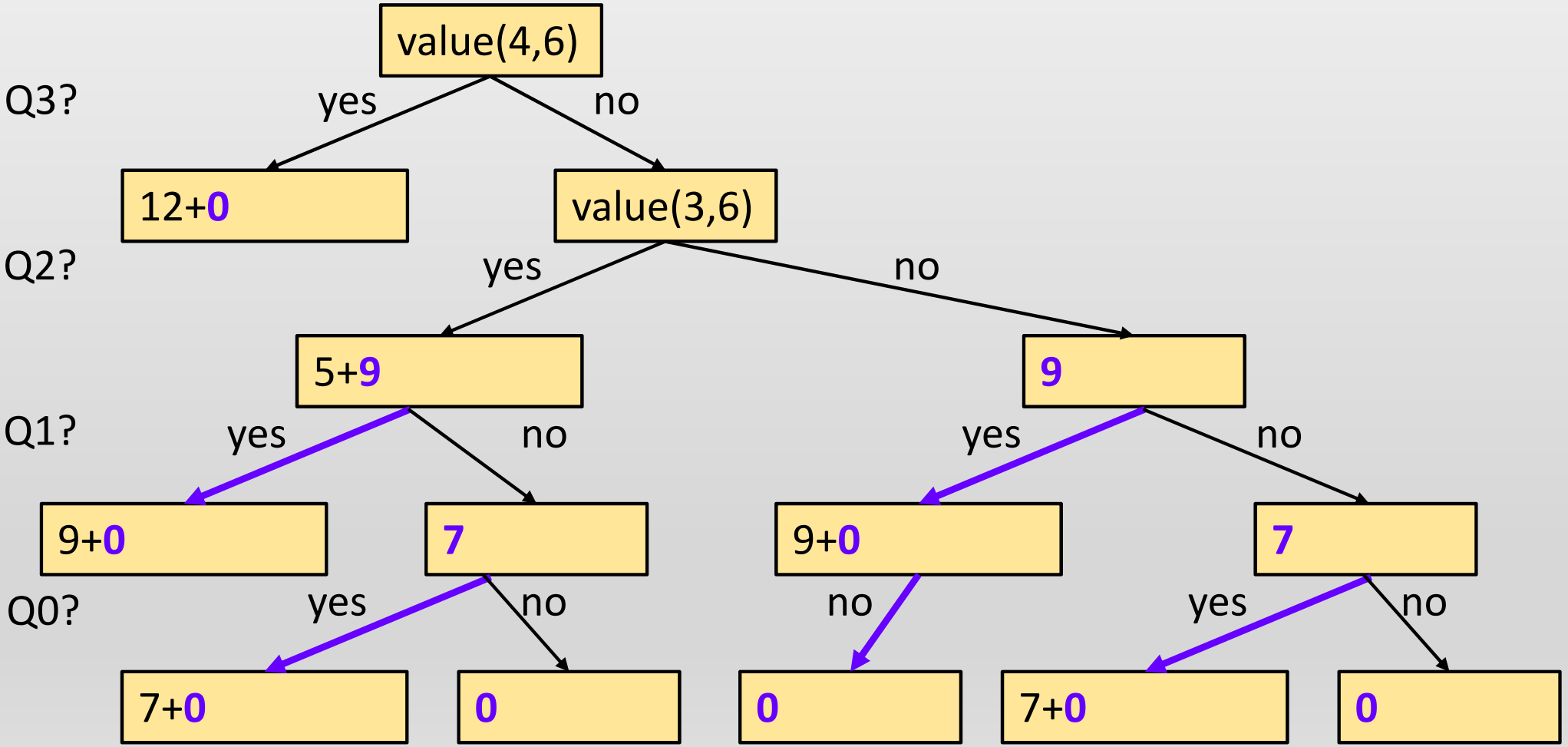
In the terminal nodes, replace those terms with `value(n,0)` or `value(0,t)` to be 0.



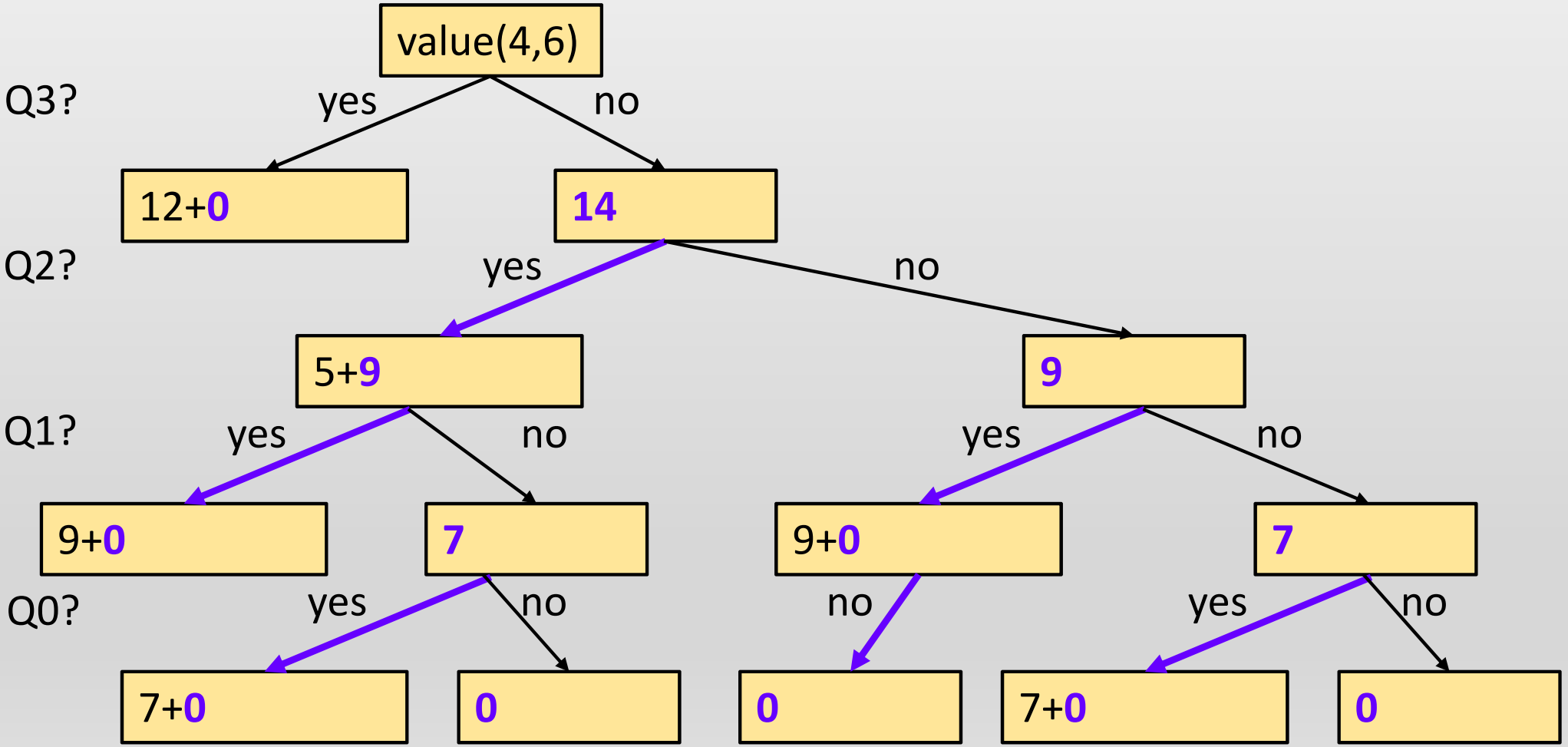
In the nodes above, allocate `value(n,t)` with the maximum value underneath.



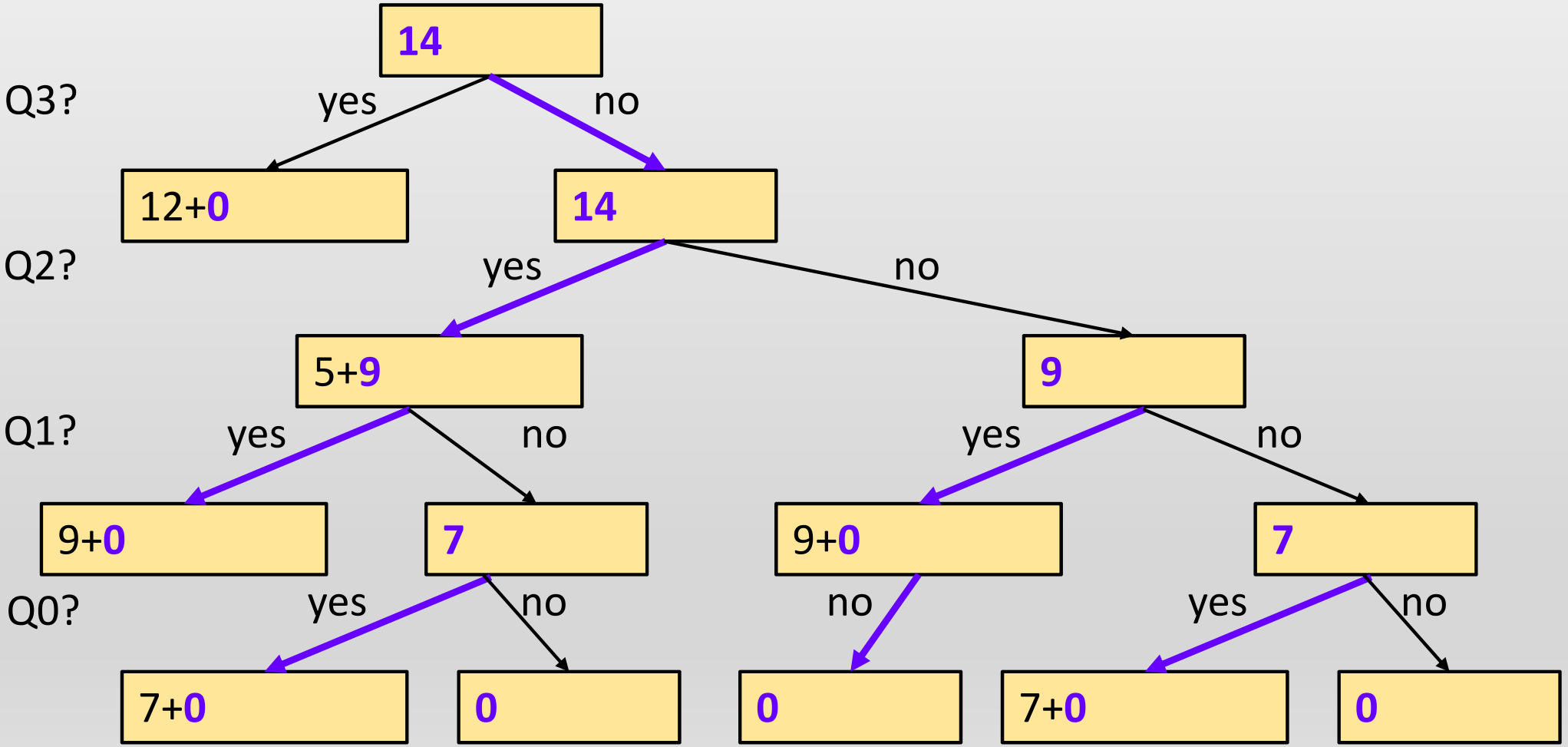
In the nodes above, allocate `value(n,t)` with the maximum value underneath.



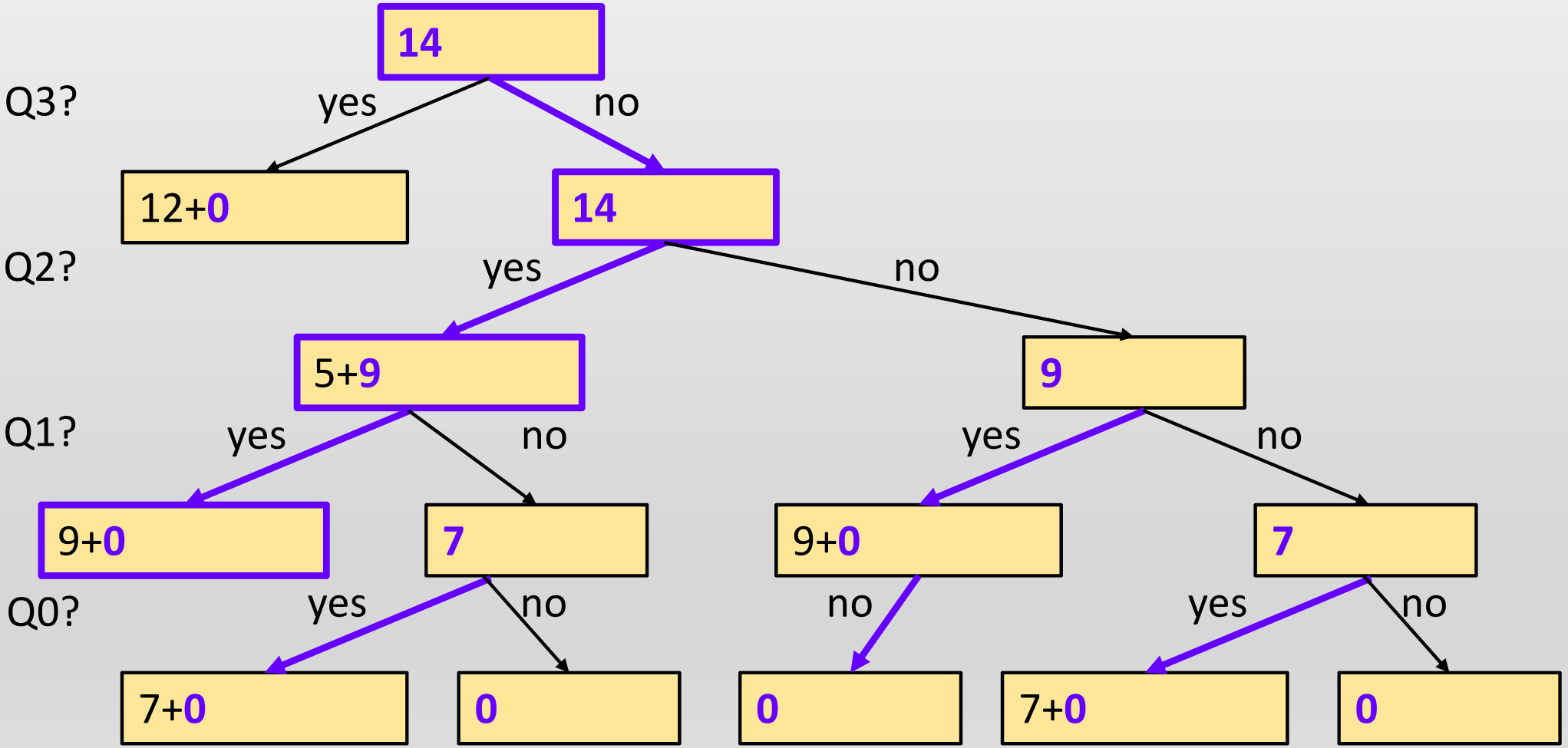
In the nodes above, allocate `value(n,t)` with the maximum value underneath.



In the nodes above, allocate `value(n,t)` with the maximum value underneath.



Going back to the original node, we can deduce `value(4,6)` to be 14, which is achieved by choosing Q1 and Q2 only.



Example 8.6 marks optimization

```
1  //eg8.6  marks optimization
2  #include<iostream>
3  using namespace std;
4  int value(int v[], int c[], int n, int t)
5  {
6          if (n==0 || t==0) return 0;
7          if (t>=c[n-1])
8                  return max(v[n-1]+value(v,c,n-1,t-c[n-1]),value(v,c,n-1,t));
9          else
10                 return value(v,c,n-1,t);
11 }
12 int main()
13 {
14         int marks[] = {7,9,5,12};
15         int time[] = {3,4,2,6};
16         cout << value(marks, time, 4, 6);
17         return 0;
18 }
```

no items left or no time left

check if the time left is enough for the question

compare the values, should I do this question to get the mark, or should I skip the question and save the time?

if not, we can only skip the question

Classwork 8.1 Knapsack problem

You are considering to bring the following items in a picnic. However, the capacity of your knapsack is only 18L. Find the optimization solution with maximum total price. You may modify from the previous example program.

items	gas stove	bread loaf	fry pan	steak	chair	wine	melon
price	7	1	3	5	4	6	2
volume (L)	8	3	4	1	10	2	5