AMA2222 Principles of Programming

Leung Man Kin, Adam
Instructor

adam.leung@polyu.edu.hk
TU720

Chapter 7: Algorithm I
concept of algorithm, swapping,
Euclidean algorithm, sorting, searching

Concept of algorithm

What is an **algorithm**?
It is a process or set of rules to be followed in calculations or other problem-solving operations.

An algorithm can be implemented by programming for operation by a computer, but not necessarily!

Here we will go over some famous algorithms used by mathematicians and scientists, from the time without a computer to the time with computer!

Think of the algorithm design before writing into a program!

Example of algorithms in daily life:

How to find the highest mark among a pile of exam paper?

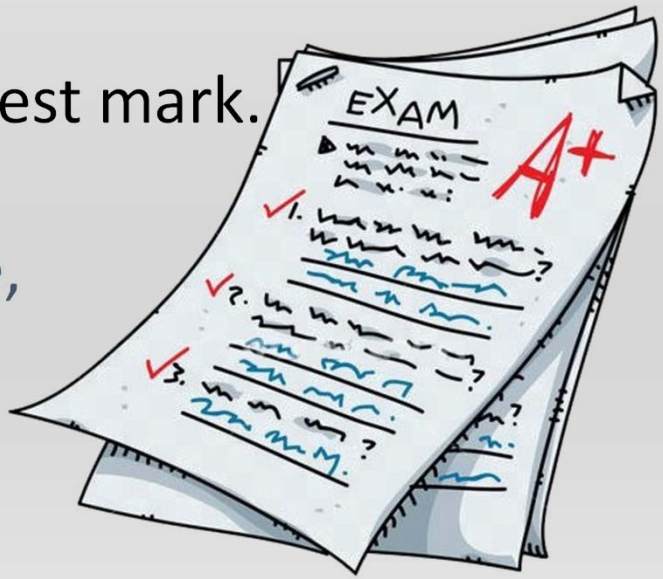
(i) take the first exam paper from the pile on your hand, assume it is the student with highest mark.

(ii) read the next paper on the pile, check if it has higher mark than the paper on your hand.

(iii) if yes, replace the paper on your hand by this paper with higher mark.

(iv) go back to (ii) if there are still some exam paper left, otherwise go to next step.

(v) the paper on your hand is with the highest mark.



Remark: (i) is initial step, (ii)-(iii) is repetitive,
(iv) is continuation condition checking,
(v) is terminal step.

Swapping

First we will learn some algorithm for **swapping**, which means exchanging the values stored in two variables. In C++, we can use the built-in function `swap` by the syntax below:

```
swap(var1, var2)
```

Compare the following and deduce the updated value of x and y. Which of these algorithms can swap values of x, y? Assume x=1 and y=2 in our example.

Example 7.1a

```
x = y;
y = x;
```

Example 7.1b

```
temp = x;
x = y;
y = temp;
```

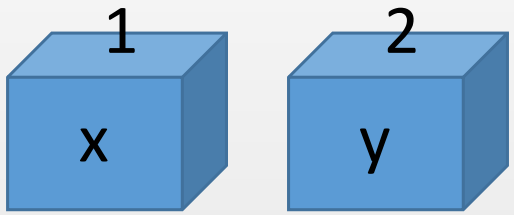
Example 7.1c

```
x = x + y;
y = x - y;
x = x – y;
```

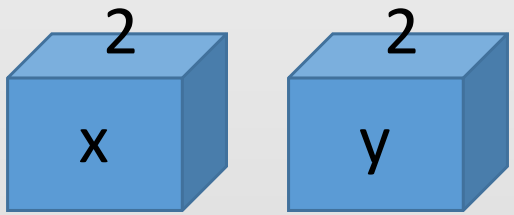
Example 7.1a

```
x = y;
y = x;
```

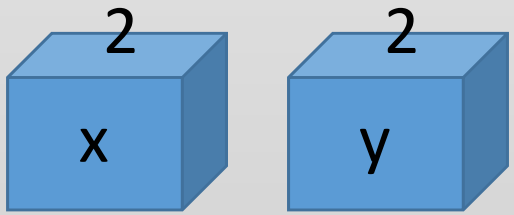
```
x = 1;
y = 2;
```




```
x = y;
```



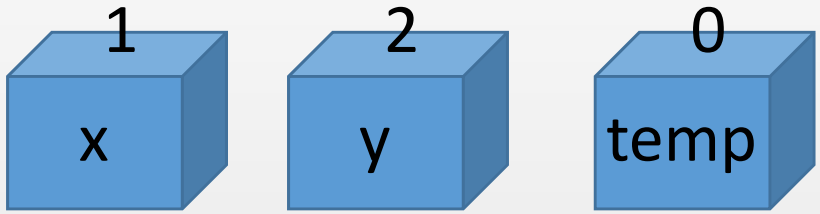

```
y = x;
```



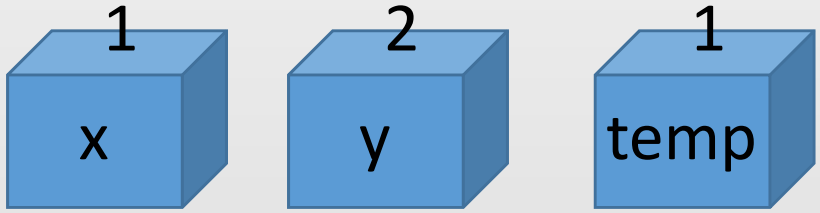
Example 7.1b

```
temp = x;
x = y;
y = temp;
```

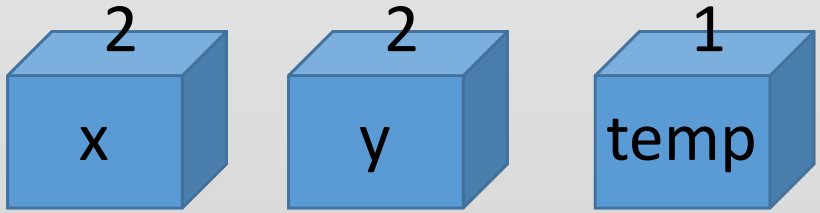
```
x = 1;
y = 2;
```



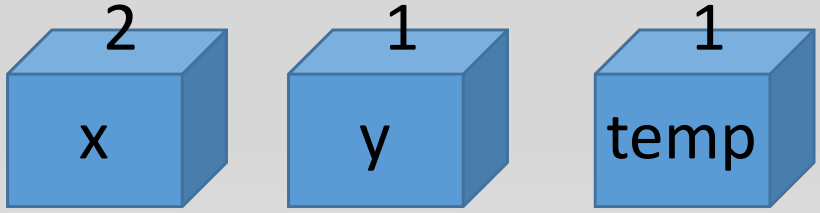

```
temp = x;
```




```
x = y;
```



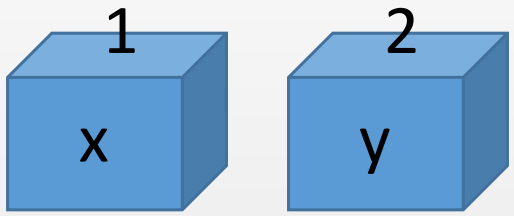

```
y = temp;
```



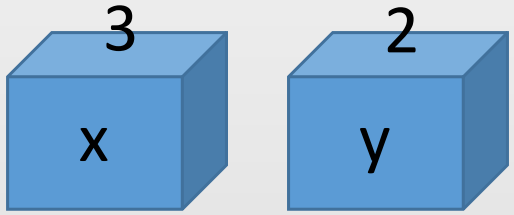
Example 7.1c

```
x = x + y;
y = x - y;
x = x – y;
```

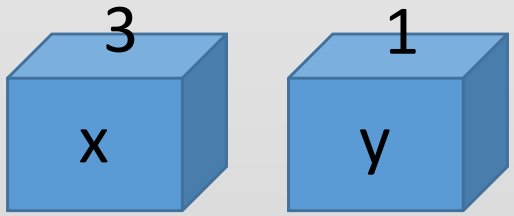
```
x = 1;
y = 2;
```



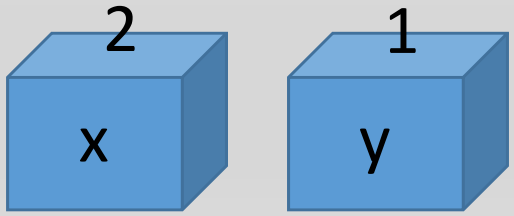

```
x = x + y;
```




```
y = x - y;
```




```
x = x - y;
```



Euclidean algorithm

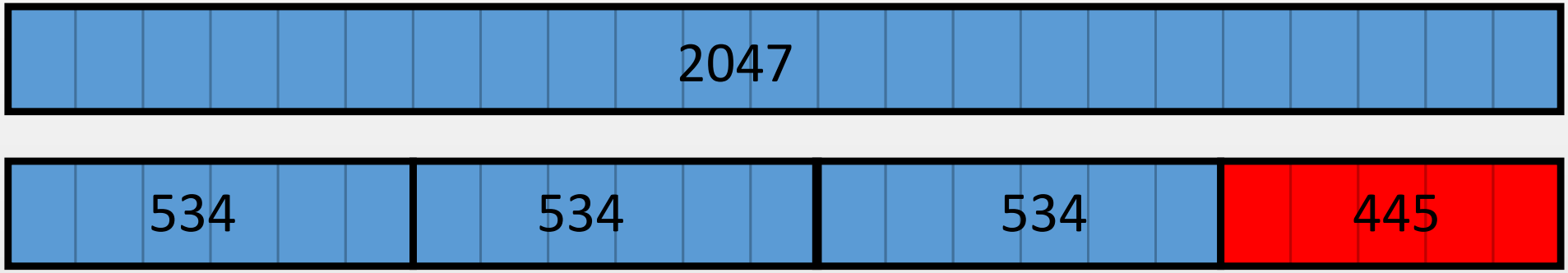
Euclidean algorithm is an algorithm to find the greatest common factor of two given distinct positive integers.

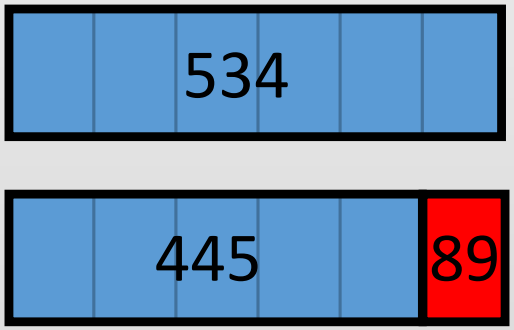
(i) Divide the bigger number by the smaller one, replace by the remainder.
(ii) Repeat (i) until the remainder is zero, then the divisor is the greatest common factor.

Euclid of Alexandria, BC325-BC265

For example, to find gcd of 534 and 2047:

```
2047    534     // 2047%534 = 445
445     534     // 534%445 = 89
445     89      // 445%89 = 0
0       89      // 89 is the gcf of 2047 and 534
```






```
2047    534     // 2047%534 = 445
445     534     // 534%445 = 89
445     89      // 445%89 = 0
0       89      // 89
```

Therefore 89 is the gcf of 2047 and 534

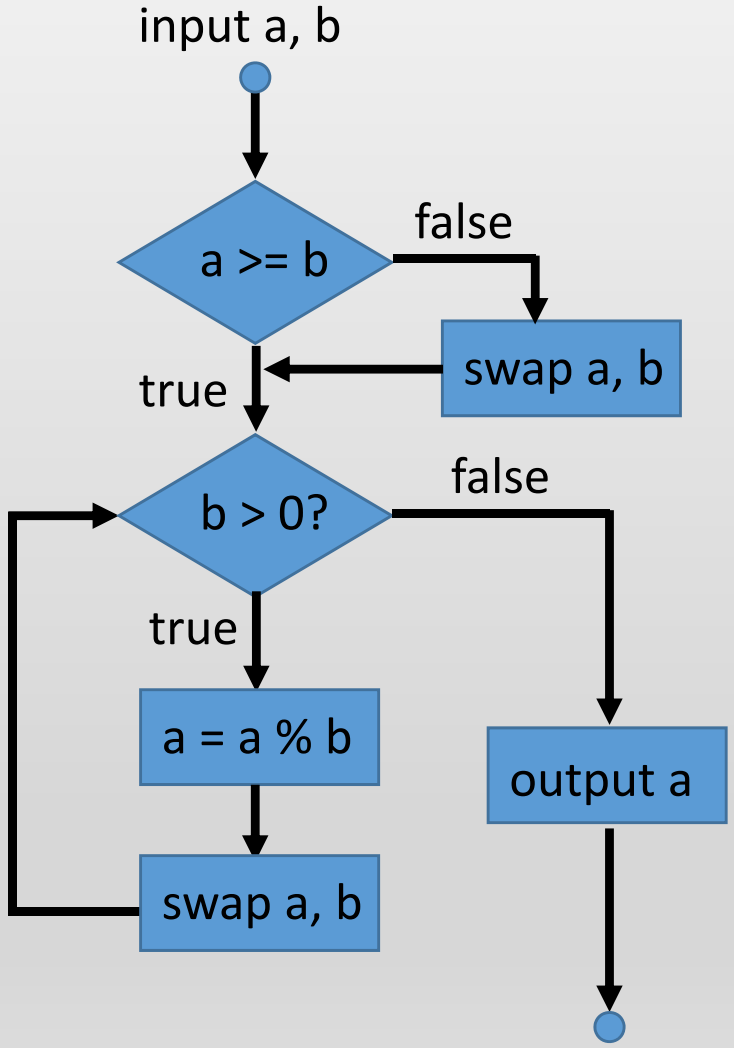
To convert his into our programming code, we need to set up variables and design the flow chart.

initial state:
a = 2047 b = 534

after first iteration:
a = 534 b = 445

after second iteration:
a = 445 b = 89

after third iteration:
a = 89 b = 0



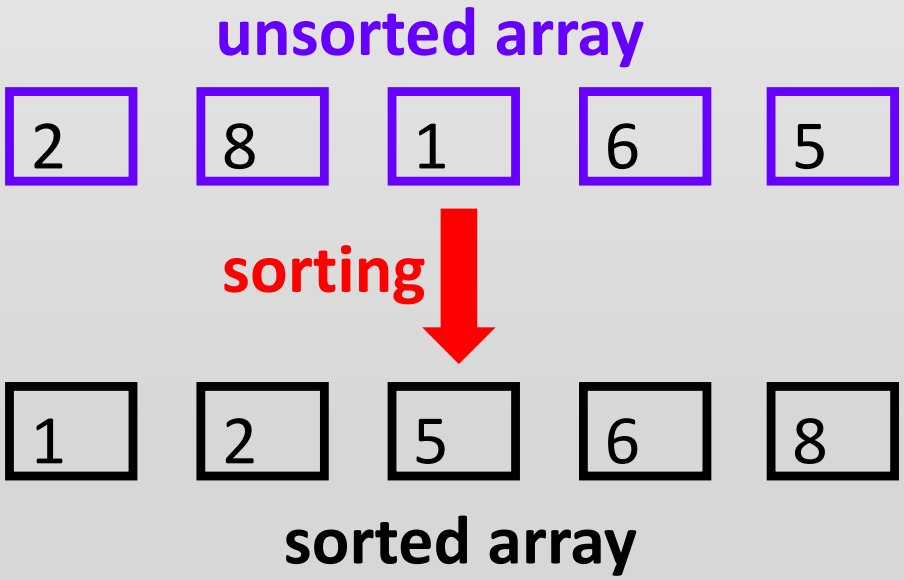
The while loop is ended. Output the answer a.

Example program 7.2 Euclidean algorithm

```
 1 #include <iostream>
 2 using namespace std;
 3 int main ()
 4 {
 5          int a, b;
 6          cout << "Enter two numbers: ";
 7          cin >> a >> b;
 8          if (a<b)
 9                    swap(a,b);
10          while (b>0)
11          {
12                    a = a % b;
13                    swap(a,b);
14          }
15          cout << "The greatest common factor is " << a;
16          return 0;
17 }
```

Sorting algorithm

Sorting is a process that rearrange elements in an array with a certain order, commonly ascending order. There are many different algorithms of sorting, for example, selection sort, bubble sort and insertion sort.

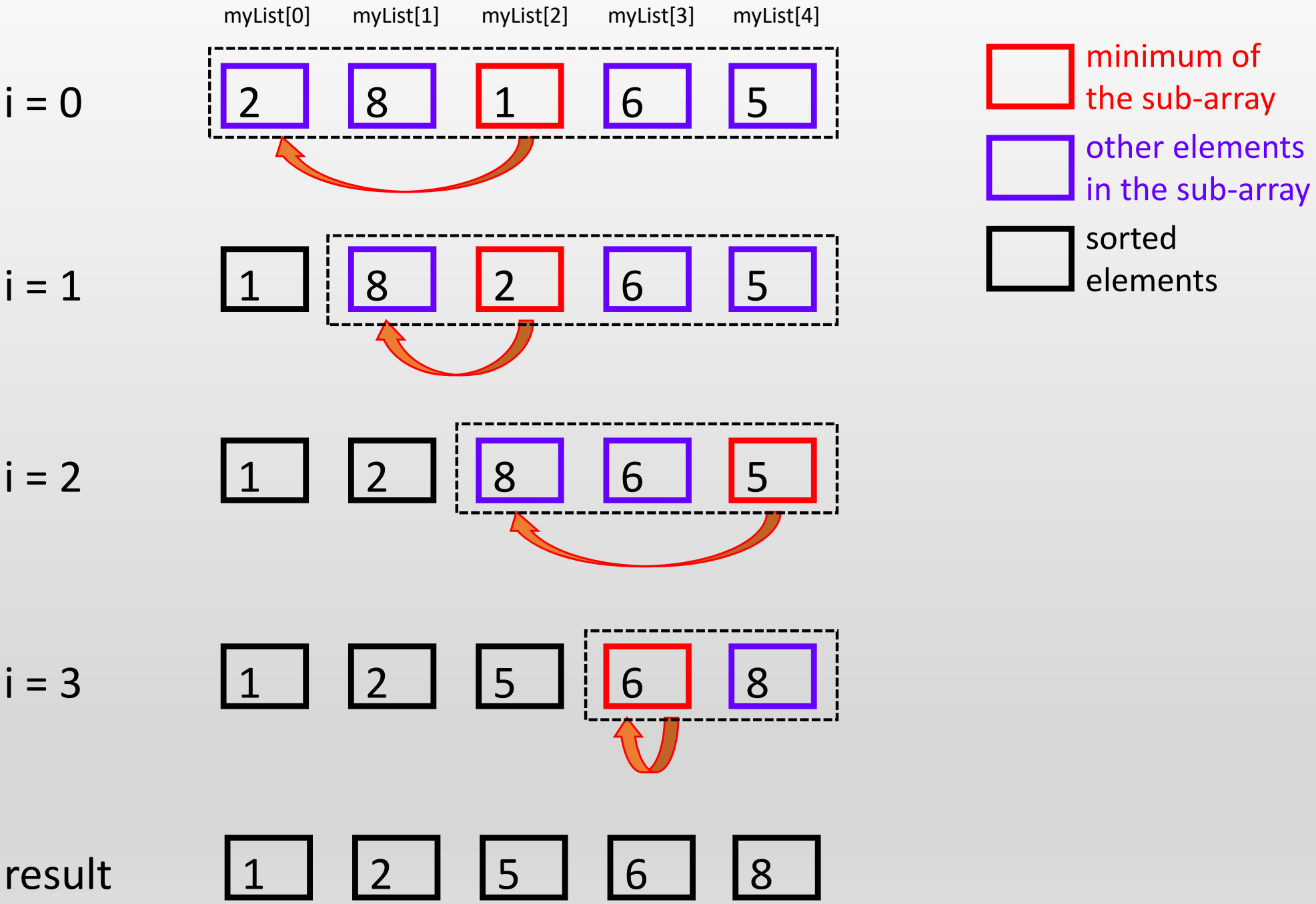


Selection sort

The idea of **selection sort** is to always pick up the minimum value from the pool of elements remaining. The picked up value become the first element among this sub-array.

By repeating this process, each element would be the minimum among the sub-array starting from itself to the end. In other words, the array is sorted in ascending order.

In the i-th iteration, the i-th element of the array is swapped with the minimum element from the i-th element to the last element. This iteration is done for i = 0 to size-2.


myList[0]
myList[1]
myList[2]
myList[3]
myList[4]
i = 0
2
8
1
6
5
i = 1
1
8
2
6
5
i = 2
1
2
8
6
5
i = 3
1
2
5
6
8
result
1
2
5
6
8
minimum of the sub-array
other elements in the sub-array
sorted elements

Example program 7.3 Selection sort

```
1  // 7.3 Selection sort
2  #include <iostream>
3  using namespace std;
4  void selectionSort(double list[], int size)
5  {
6          for (int i = 0; i < size - 1; i++)
7                  for (int j = i + 1; j < size; j++)
8                          if (list[j] < list[i])
9                                  swap(list[j],list[i]);
10 }
11 void printArray(double list[], int size)
12 {
13         for (int i = 0; i < size; i++)
14                 cout << list[i] << " ";
15 }
16 int main()
17 {
18         int size = 5;
19         double mylist[] = {2, 8, 1, 6, 5};
20         selectionSort(mylist, size);
21         cout << "The sorted array: ";
22         printArray(mylist, size);
23         cout << endl;
24         return 0;
25 }
```

Iterations from i=0 to i=size-2, which means from the beginning to the second last element.

check the elements on the right of list[i].

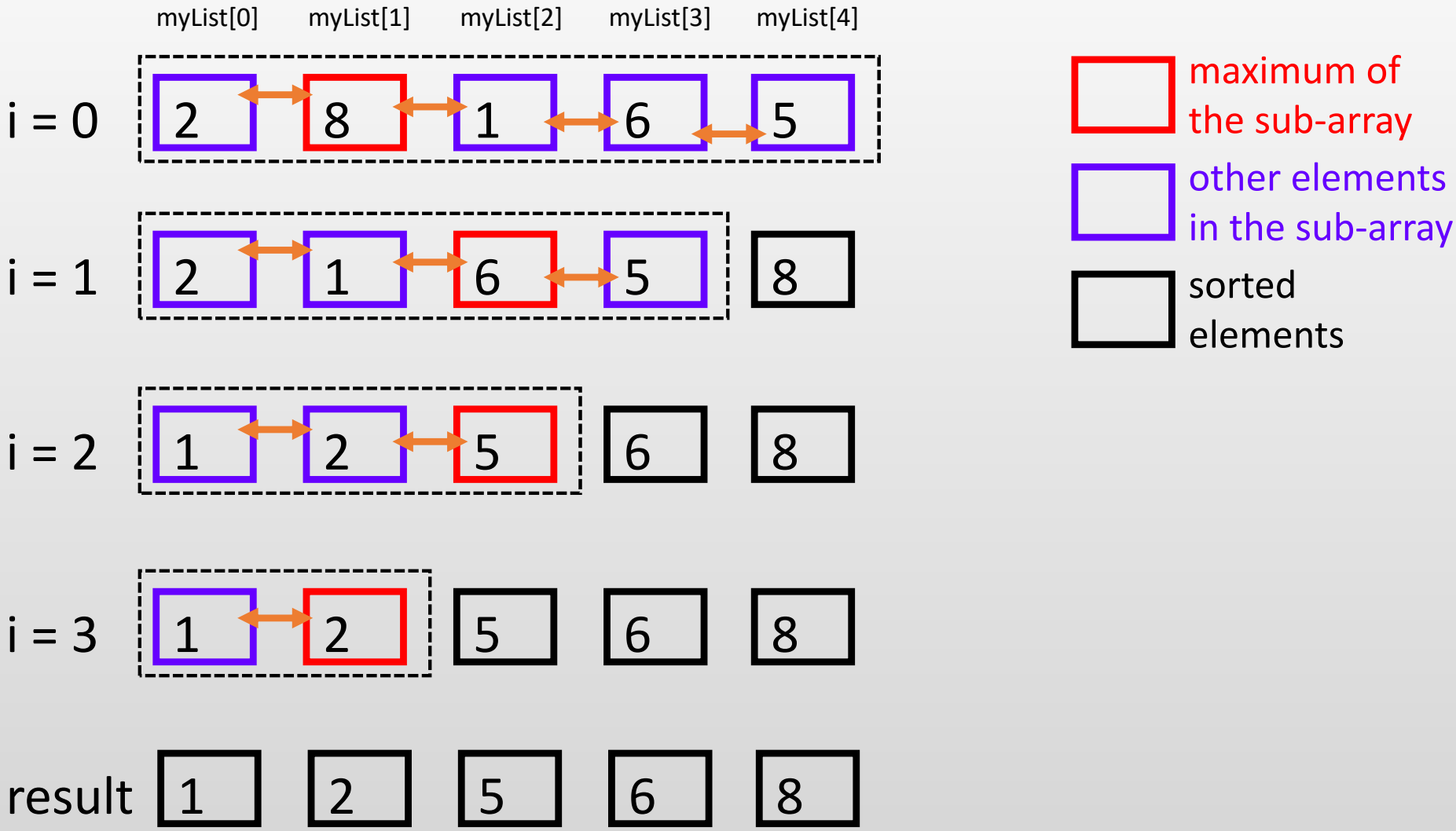
If an element smaller than list[i] is found, swap. This would guarantee list[i] being smallest compared to all elements on its right.

The sorted array: 1 2 5 6 8

Bubble sort

The idea of **bubble sort** is to use a series of pairwise comparison and swapping between consecutive elements in order to put the biggest value to the end of the array. That's why it is also called sinking sort.

In the i-th iteration, we do pairwise swapping of each element to its next element to make them ascending, starting from the 0-th element up to the (size-2)-th element. This iteration is done for i = 0 to size-2.


myList[0]
myList[1]
myList[2]
myList[3]
myList[4]
i = 0
2
8
1
6
5
i = 1
2
1
6
5
8
i = 2
1
2
5
6
8
i = 3
1
2
5
6
8
result
1
2
5
6
8
maximum of the sub-array
other elements in the sub-array
sorted elements

Example program 7.4 Bubble sort

```
1  // 7.4 Bubble sort
2  #include <iostream>
3  using namespace std;
4  void bubbleSort(double list[], int size)
5  {
6          for (int i = 0; i < size - 1; i++)
7                  for (int j = 0; j < size-1-i; j++)
8                          if (list[j] > list[j+1])
9                                  swap(list[j],list[j+1]);
10 }
11 void printArray(double list[], int size)
12 {
13         for (int i = 0; i < size; i++)
14                 cout << list[i] << " ";
15 }
16 int main()
17 {
18         int size = 5;
19         double mylist[] = {2, 8, 1, 6, 5};
20         bubbleSort(mylist, size);
21         cout << "The sorted array: ";
22         printArray(mylist, size);
23         cout << endl;
24         return 0;
25 }
```

Iterations from i=0 to i=size-2, which means from the beginning to the second last element.

do consecutive pairwise comparison

the biggest element would be swapped every time until it sink to the end

The sorted array: 1 2 5 6 8

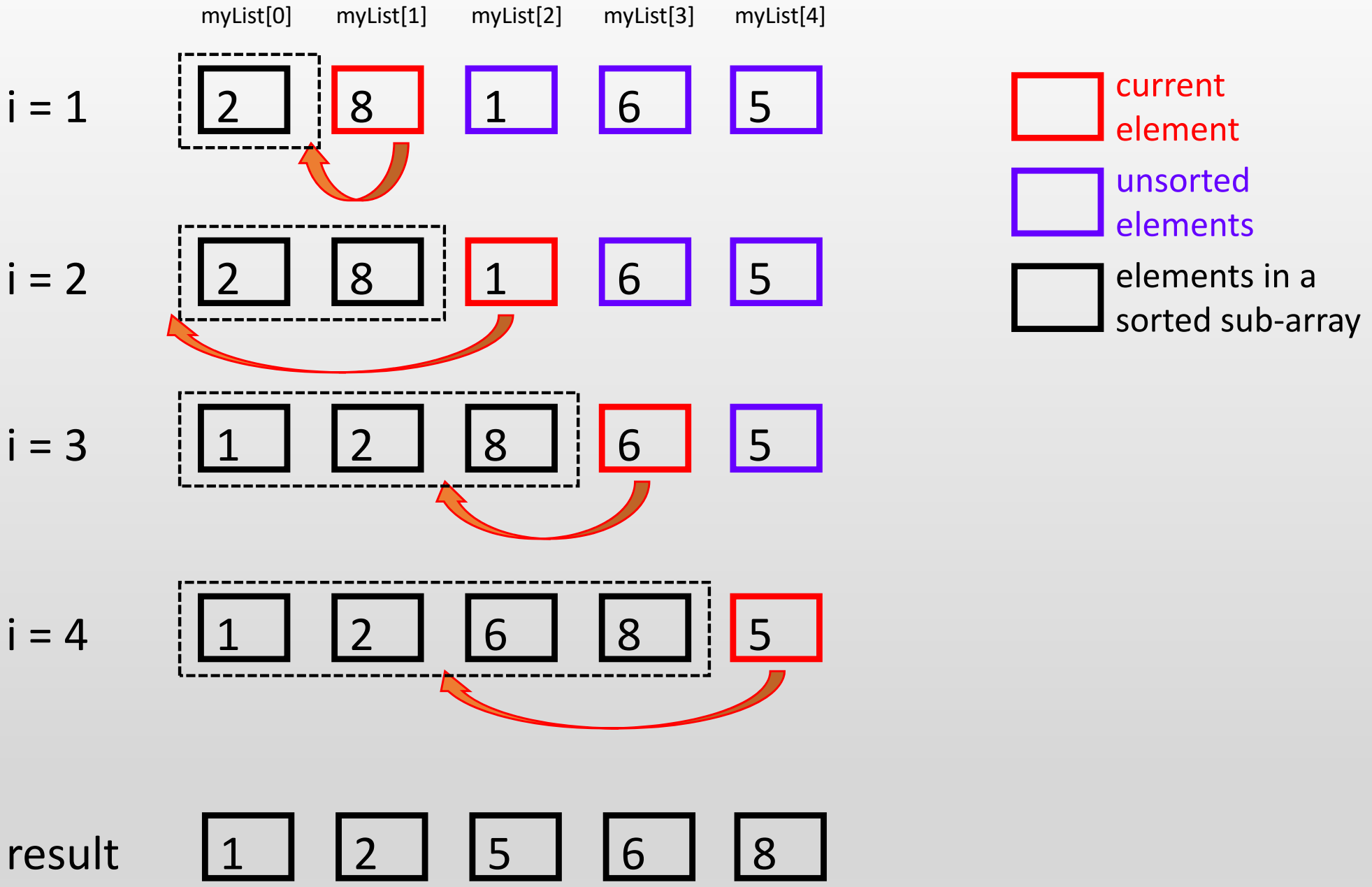
Insertion sort

The idea of **insertion sort** is to append the elements one by one according to ascending order. After appending to a suitable position, the elements after in in the sub-array have to be shifted.

If the array only contains one element, it is already sorted.

In the i-th iteration, we consider adding the i-th element of the array into the sorted sub-array indexed from 0 to i-1. Suppose this element should be put in position index pos. We will then shift each elements of the sub-array at index pos to i-1 by 1.

After doing this iteration for i being 1 to size-1, the whole array is sorted.


myList[0]
myList[1]
myList[2]
myList[3]
myList[4]
i = 1
2
8
1
6
5
i = 2
2
8
1
6
5
i = 3
1
2
8
6
5
i = 4
1
2
6
8
5
result
1
2
5
6
8
current element
unsorted elements
elements in a sorted sub-array

Example program 7.5 Insertion sort

```
1  // 7.5 Insertion sort
2  #include <iostream>
3  using namespace std;
4  void insertionSort(double list[], int size)
5  {
6          for (int i = 1; i < size; i++){
7                  int pos = i;
8                  double temp = list[i];
9                  for (int j = 0; j < i; j++)
10                         if (list[i] < list[j]){
11                                 pos = j;
12                                 break;}
13                 for (int j=i; j>pos; j--)
14                         list[j] = list[j-1];
15                 list[pos] = temp;
16         }
17 }
```

Iterations from i=1 to i=size-1, the initial step i=0 can be skipped as it is already sorted

store the index and value of the current element

find the suitable position where we should insert the current element

shift the elements backward and insert the current element to the position

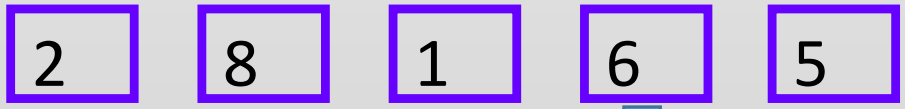
```
18 void printArray(double list[], int size)
19 {
20          for (int i = 0; i < size; i++)
21                    cout << list[i] << " ";
22 }
23 int main()
24 {
25          int size = 5;
26          double list[] = {2, 8, 1, 6, 5};
27          insertionSort(list, size);
28          cout << "The sorted array: ";
29          printArray(list, size);
30          cout << endl;
31          return 0;
32 }
```

```
The sorted array: 1 2 5 6 8
```

Searching arrays

Searching is a process that we are looking for a specific element (or elements) in an array. Two commonly used approaches are **linear search** and **binary search**.

For linear search, we just apply a for-loop to go over all elements in the array one-by-one, and compare with the **key**.



Is there an element valued 6 in the array?

Yes, at position index 3.

Is there an element valued 7 in the array?

No, there isn't.

Example program 7.6 Searching an array by linear search

```
1  // 7.6 Linear search
2  #include <iostream>
3  using namespace std;
4  int linearSearch(int list[], int key, int size)
5  {
6          for (int i = 0; i < size; i++)
7          {
8                  if (key == list[i])
9                          return i;
10         }
11         return -1;
12 }
13 int main()
14 {
15         int size = 5;
16         int list[] = {2, 8, 1, 6, 5};
17         int key = 6;
18         int x = linearSearch(list, key, size);
19         if (x >= 0)
20                 cout << "The " << x << "-th element is " << key << endl;
21         else cout << key << " is not in the array" << endl;
22         return 0;
23 }
```

Checking all elements indexed with i = 0 to i = size-1

Once an element that equals to the key is found, the function terminates and return the index of such element.

At the end of all iterations, if no such element is found, return -1.

The complexity of using linear search however, has same order as the array size. That is, the time needed grows linearly with the number of elements in the array.

Binary search is a process useful for searching a specific element in a sorted array. Its complexity is just proportional to the logarithm of array size.

In each iteration, consider the following cases:

- If the key is less than the middle element, search only the first half of the array.
- If the key is equal to the middle element, done with a match.
- If the key is greater than the middle element, search only the second half of the array.

Is there an element valued 6 in the array?

The median is 5,
the key is greater

Search the second half

The median is 8,
the key is smaller

Search the first half

The median is 6,
match the key

Example program 7.7
Searching an array by binary search

```
1  // 7.7 Binary search
2  #include <iostream>
3  using namespace std;
4  int binarySearch(int list[], int key, int size)
5  {
6            int low = 0;
7            int high = size - 1;
8            while (high >= low)
9            {
10                     int mid = (low + high) / 2;
11                     if (key < list[mid])
12                              high = mid - 1;
13                     else     if (key == list[mid])
14                                       return mid;
15                              else low = mid + 1;
16           }
17           return -1;
18 }
19 int main()
20 {
21           int size = 7;
22           int list[] = {1, 2, 3, 5, 6, 8, 9};
23           int key = 6;
24           int x = binarySearch(list, key, size);
25           if (x >= 0) cout << "The " << x << "-th element is " << key << endl;
26           else cout << key << " is not in the array" << endl;
27           return 0;
28 }
```

beginning: low is 0 high is 6

iteration 0: mid = (0+6)/2 = 3 list[3] is 5

key > list[mid] therefore low = 3+1 = 4, high = 6

iteration 1: mid = (4+6)/2 = 5 list[5] is 8

key < list[mid] therefore low = 4, high = 5-1 = 4

iteration 2: mid = (4+4)/2 = 4 list[4] is 6

key = list[mid] therefore return mid which is 4

What if the key is changed to 7, which cant be found from the array?

beginning: low is 0 high is 6

iteration 0: mid = (0+6)/2 = 3 list[3] is 5

key > list[mid] therefore low = 3+1 = 4, high = 6

iteration 1: mid = (4+6)/2 = 5 list[5] is 8

key < list[mid] therefore low = 4, high = 5-1 = 4

iteration 2: mid = (4+4)/2 = 4 list[4] is 6

key > list[mid] therefore low = 4+1 = 5

Since low>high, it violates the continuation condition, terminating the while loop. Hence return -1.

After learning some basic sorting and searching algorithm, we will look into the application of sorting and searching in problem solving.

Example 7.8 Prompt the user to enter the array size and elements of an array of double values. Evaluate the median value. Refer to the sample below:

```
Enter the array size: 5
Enter the array elements: -1.5 4.7 3.4 2.2 -0.9
The median is 2.2

Enter the array size: 4
Enter the array elements: -1.5 4.7 3.4 2.2
The median is 2.8
```

We will first sort the array in ascending order.

For odd number of terms, median refers to the value lying in the middle.

Sample 1:

-1.5	-0.9	2.2	3.4	4.7

median = 2.2

For even number of terms, median refers to the average of the two values lying in the middle.

Sample 2:

-1.5	2.2	3.4	4.7

median =
(2.2 + 3.4)/2
= 2.8

```
1  // 7.8 median
2  #include <iostream>
3  using namespace std;
4  void selectionSort(double list[], int size)
5  {
6          for (int i = 0; i < size - 1; i++)
7                  for (int j = i + 1; j < size; j++)
8                          if (list[j] < list[i])
9                                  swap(list[j],list[i]);
10 }
11 int main()
12 {
13         int size;
14         cout << "Enter the array size: ";
15         cin >> size;
16         double a[size];
17         cout << "Enter the array elements: ";
18         for (int i=0; i<size; i++)
19                 cin >> a[i];
20         selectionSort(a, size);
21         double m;
22         if (size%2==1)
23                 m = a[size/2];
24         else
25                 m = (a[size/2-1]+a[size/2])/2;
26         cout << "The median is " << m;
27         return 0;}
```

keep the sorting function

ask user to enter the size first, then define an integer array of such size and ask user to enter its elements

sort the array a[]

evaluate the median depending on the parity (odd/even) of size

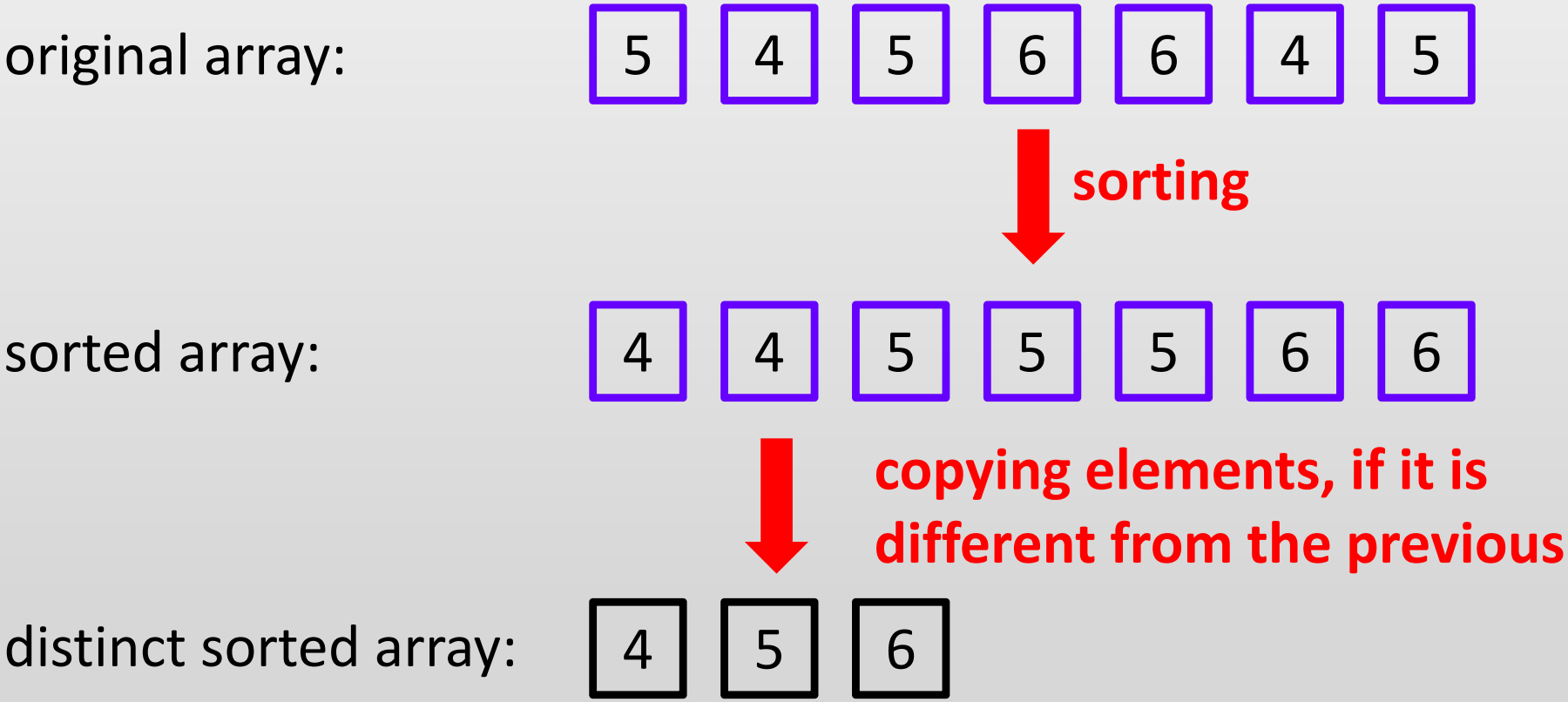
Example 7.9 Prompt the user to enter the array size and elements of an array of positive integers. Display the distinct values in ascending order. Refer to the sample below:

```
Enter the array size: 7
Enter the array elements: 5 4 5 6 6 4 5
The distinct elements in ascending order: 4 5 6
```

Notice that in the array above, some elements are duplicated. We need to form a new array with those distinct elements and sorted in ascending order.

Before we look into the program, we shall first try to solve the problem manually using the sample and develop an algorithm.

```
Enter the array size: 7
Enter the array elements: 5 4 5 6 6 4 5
The distinct elements in ascending order: 4 5 6
```



keep the sorting and printing functions, change the data type of list[] to int

```
1  // 7.9 sorting distinct elements
2  #include <iostream>
3  using namespace std;
4  void selectionSort(int list[], int size)
5  {
6          for (int i = 0; i < size - 1; i++)
7                  for (int j = i + 1; j < size; j++)
8                          if (list[j] < list[i])
9                                  swap(list[j],list[i]);
10 }
11 void printArray(int list[], int size)
12 {
13         for (int i = 0; i < size; i++)
14                 cout << list[i] << " ";
15 }
16 int main()
17 {
18         int size;
19         cout << "Enter the array size: ";
20         cin >> size;
21         int a[size];
22         cout << "Enter the array elements: ";
23         for (int i=0; i<size; i++)
24                 cin >> a[i];
```

ask user to enter the size first, then define an integer array of such size and ask user to enter its elements

```
25          selectionSort(a, size);
26          int b[size];
27          b[0]=a[0];
28          int i=1;
29          for (int j=1; j<size; j++)
30                  if (a[j] != a[j-1]){
31                          b[i] = a[j];
32                          i++;}
33          cout << "The distinct elements in ascending order: ";
34          printArray(b, i);
35          cout << endl;
36          return 0;
37  }
```

sort the original array a[]

define a new array b[] with the same size and same starting element as a[]

use i as the index of the next element in b[] to be updated

go over all remaining elements in a[], check if it is distinct by comparing to previous element

if yes, copy this element into b[i] and go to the next index by increment i

notice that the final updated value of i represents number of valid elements in b[]