Divide & Conquer

- Divide and Conquer: one of the most practical strategies to solve problems.

- Given a function to compute on n inputs the divide and conquer strategy suggests splitting the inputs into k distinct subsets yielding k, $1 < k \leq n$ subproblems which can be solved recursively.

- Control abstraction: a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meaning is left undefined.

```
Global n :integer;
        A: array[1..n] of integer;
procedure Divide-And-Conquer(p,q:integer);
var m:integer;
begin
        if SMALL(p,q) then  //input size is small enough//
                G(p,q)                          //solve directly//
        else
        begin
                m ← DIVIDE(p,q);   // p ≤ m < q
                COMBINE(Divide-And-Conquer(p,m),
                                Divide-And-Conquer(m+1,q))
        end
end.
```

- Computing time for the Divide-And-Conquer is naturally described by the recurrence relation:

$$T(n) = \begin{cases} g(n), & n \quad small \\ 2T(n/2) + f(n), & otherwise \end{cases}$$

Binary Search:

- Instance I = $(n, a_1, a_2, \ldots, a_n, x)$ is divided in to sub instances. One possibility is to pickup the index k and obtain three sub instances: $I_1 = (k-1, a_1, \ldots, a_{k-1}, x)$, $I_2 = (1, a_k, x)$ and $I_3 = (n-k, a_{k+1}, \ldots, a_n, x)$.

- If $x = a_k$ then instances I_1 and I_3 need not be solved and similarly other conditions can be obtained.

```
function BinSearch1(A:arraytype; n:integer; x:item): integer;
        var:lower, upper, middle:integer;
begin
        lower←1; upper←n;
        repeat
            middle ← (lower+upper) div 2;
            if (x > A[middle]) then
                lower ← middle + 1;
            else
                upper ← middle - 1;
        until ((A[middle] = x) or (lower > upper));
            if (A[middle = x) then
                BinSearch1 ← middle;
           else BinSearch1 ← 0;
end.
```

```
function BinSearch2(A:arraytype; n:integer; x:item):integer;
var: lower, upper, middle:integer; found:boolean;
begin
        lower←1; upper←n; found ← false; BinSearch2←0;
        while((lower ≤ upper) and not found) do
        begin
           middle ← (lower+upper) div 2;
           if (A[middle] = x) then
           begin
               BinSeacrh2 ← middle;  found ← true;
           end;
           else
           if (x > A[middle]) then  lower ← middle + 1;
           else upper ← middle - 1;
        end
end.
```

```
function BinSearch(A:arraytype; n:integer; x:item):integer;
var: lower, upper, middle:integer; found:boolean;
begin
        lower←1; upper←n; found ← false;
        while(( lower ≤ upper ) and not found) do
        begin
                middle ← (lower+upper) div 2;
                if (A[middle] = x) then found ← true;
                else
                if (x > A[middle]) then lower ← middle + 1;
                else  upper ← middle - 1;
        end
        if found then BinSearch ← middle
        else BinSearch ← 0;
end.
```

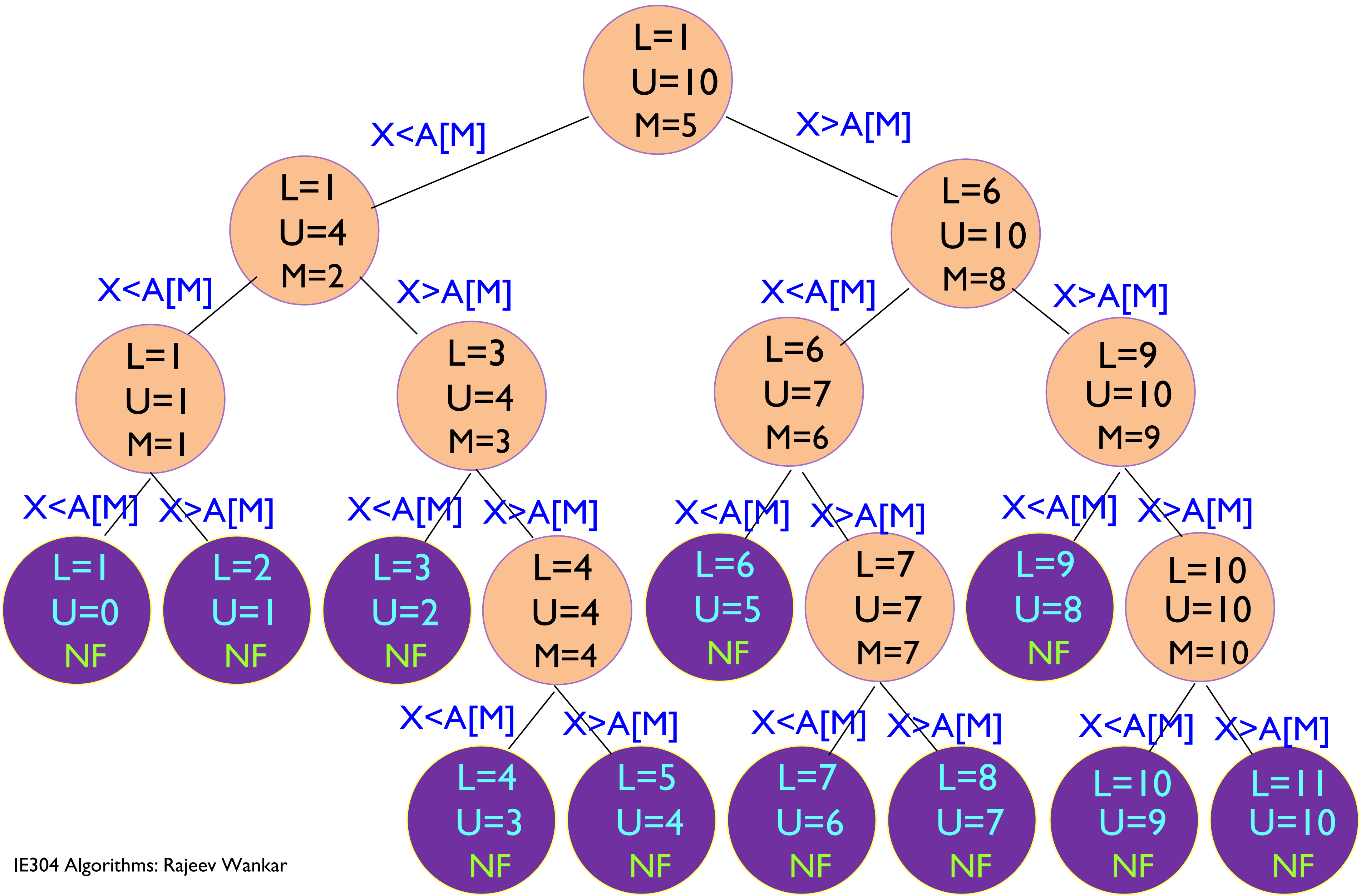
- Let n = number of sorted elements in array A
- Internal nodes in tree associated with “BinSearch” = n
- Leaf nodes in tree associated with “BinSearch” = n+1
- Levels in tree associated with “BinSearch” = k(say) = $\log_2(2n+1+1)$

Successful Searches:

Best case: one comparison: Θ(1)

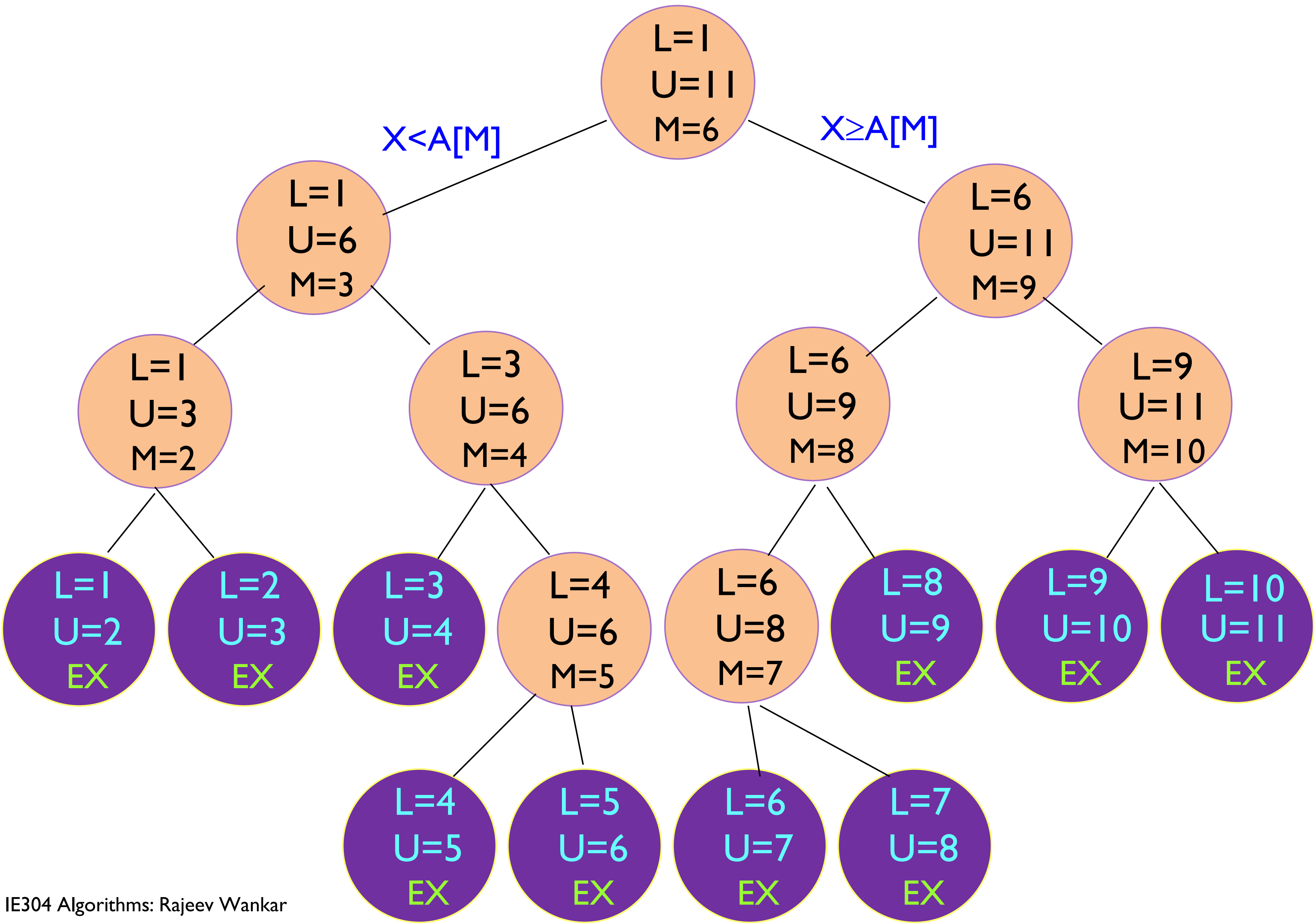
Worst case: $2\lfloor \log(n+1) \rfloor - 1$ comparisons: Θ(log n)

Average case: Θ(log n) comparisons: Θ(log n)



L=1 U=10 M=5
X<A[M]
X>A[M]
L=1 U=4 M=2
L=6 U=10 M=8
X<A[M]
X>A[M]
X<A[M]
X>A[M]
L=1 U=1 M=1
L=3 U=4 M=3
L=6 U=7 M=6
L=9 U=10 M=9
X<A[M]
X>A[M]
X<A[M]
X>A[M]
X<A[M]
X>A[M]
X<A[M]
X>A[M]
L=1 U=0 NF
L=2 U=1 NF
L=3 U=2 NF
L=4 U=4 M=4
L=6 U=5 NF
L=7 U=7 M=7
L=9 U=8 NF
L=10 U=10 M=10
X<A[M]
X>A[M]
X<A[M]
X>A[M]
X<A[M]
X>A[M]
L=4 U=3 NF
L=5 U=4 NF
L=7 U=6 NF
L=8 U=7 NF
L=10 U=9 NF
L=11 U=10 NF

```
function ModBinSearch(A:arraytype; n:integer; x:item):integer;
var: lower, upper, middle:integer;
begin
lower←1; upper←n+1; //upper is always one more than is possible//
while( lower < (upper − 1)) do
begin
    middle ← (lower+upper) div 2;
    if (x < A[middle]) then  //only one comparison in the loop//
          upper← middle;
    else
          lower ← middle;
end
if (x = A[lower]) then  ModBinSearch ← lower   //x is present//
else ModBinSearch ← 0;                          //x is not present//
end.
```


L=1
U=11
M=6
X<A[M]
X≥A[M]
L=1
U=6
M=3
L=6
U=11
M=9
L=1
U=3
M=2
L=3
U=6
M=4
L=6
U=9
M=8
L=9
U=11
M=10
L=1
U=2
EX
L=2
U=3
EX
L=3
U=4
EX
L=4
U=6
M=5
L=6
U=8
M=7
L=8
U=9
EX
L=9
U=10
EX
L=10
U=11
EX
L=4
U=5
EX
L=5
U=6
EX
L=6
U=7
EX
L=7
U=8
EX

Merge sort

- The idea of a merge sort is to divide an array in half, sort each half, and then merge the two halves into a single sorted array.

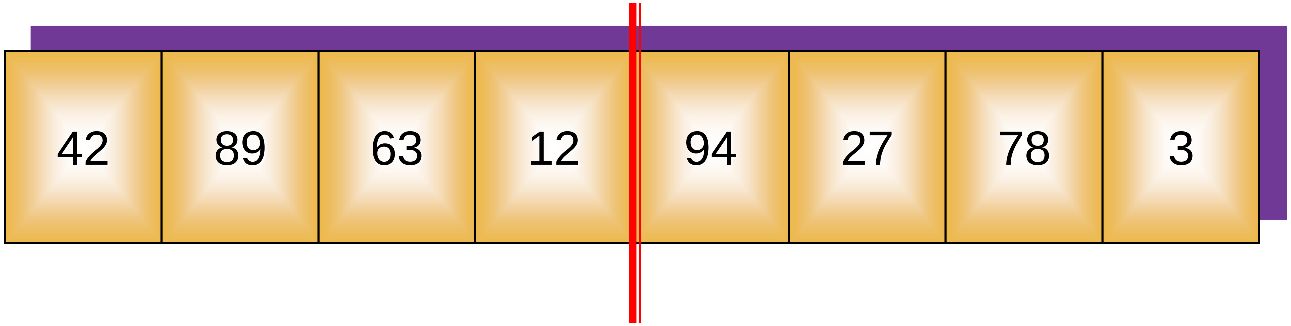
- How do we sort each half ?

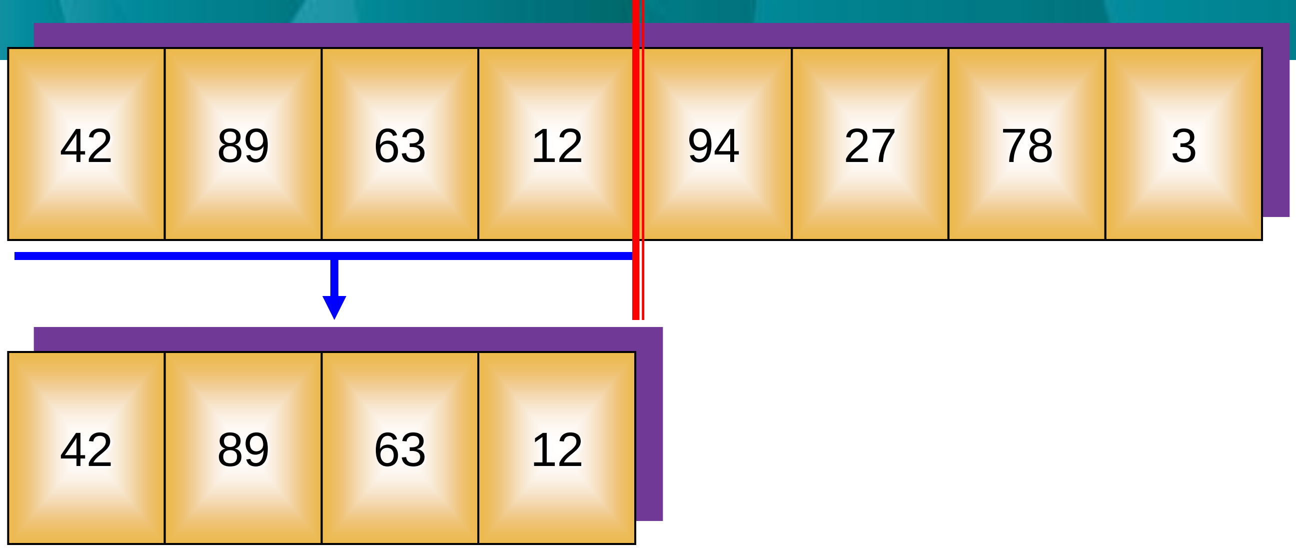
    - **Using merge sort**

- How do we merge sorted halves

42	89	63	12	94	27	78	3

Merge sort Example


42
89
63
12
94
27
78
3

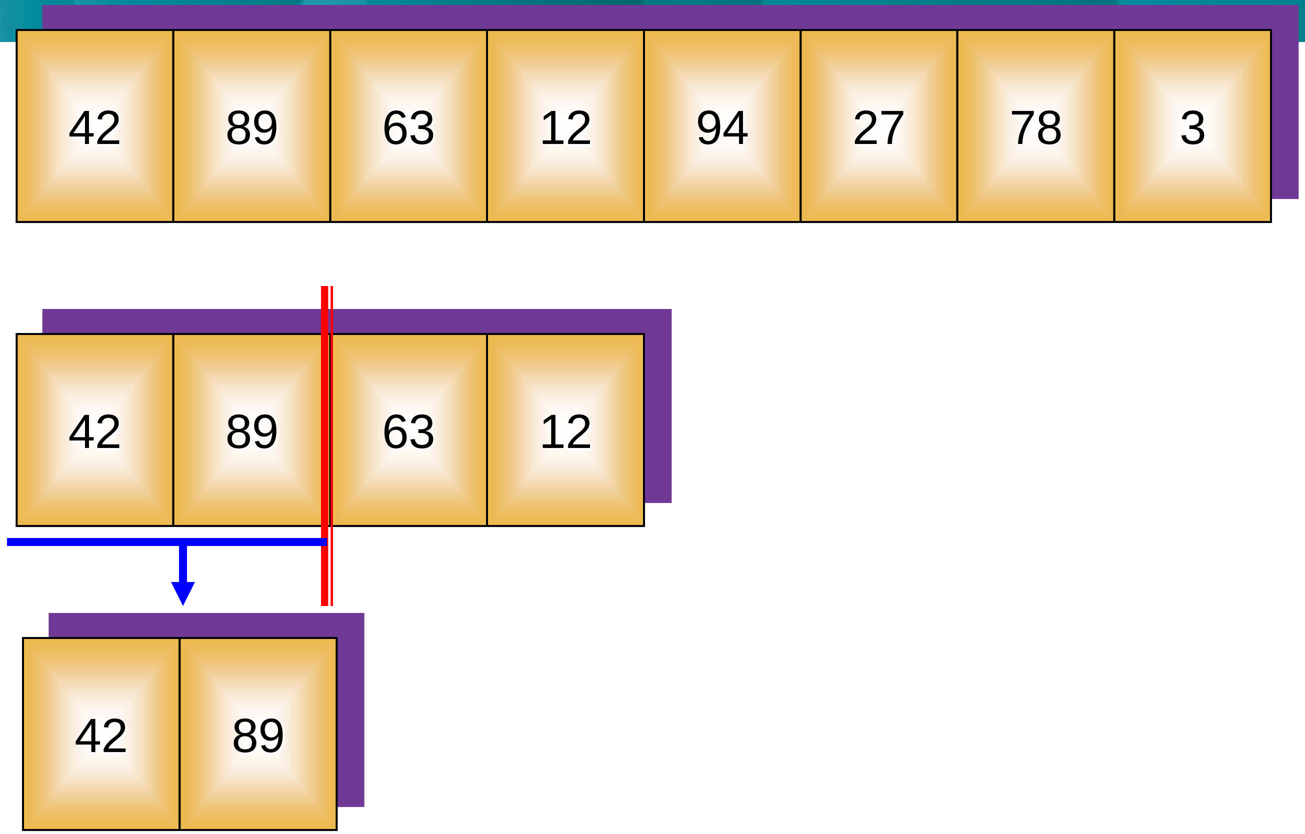

42
89
63
12
94
27
78
3
42
89
63
12

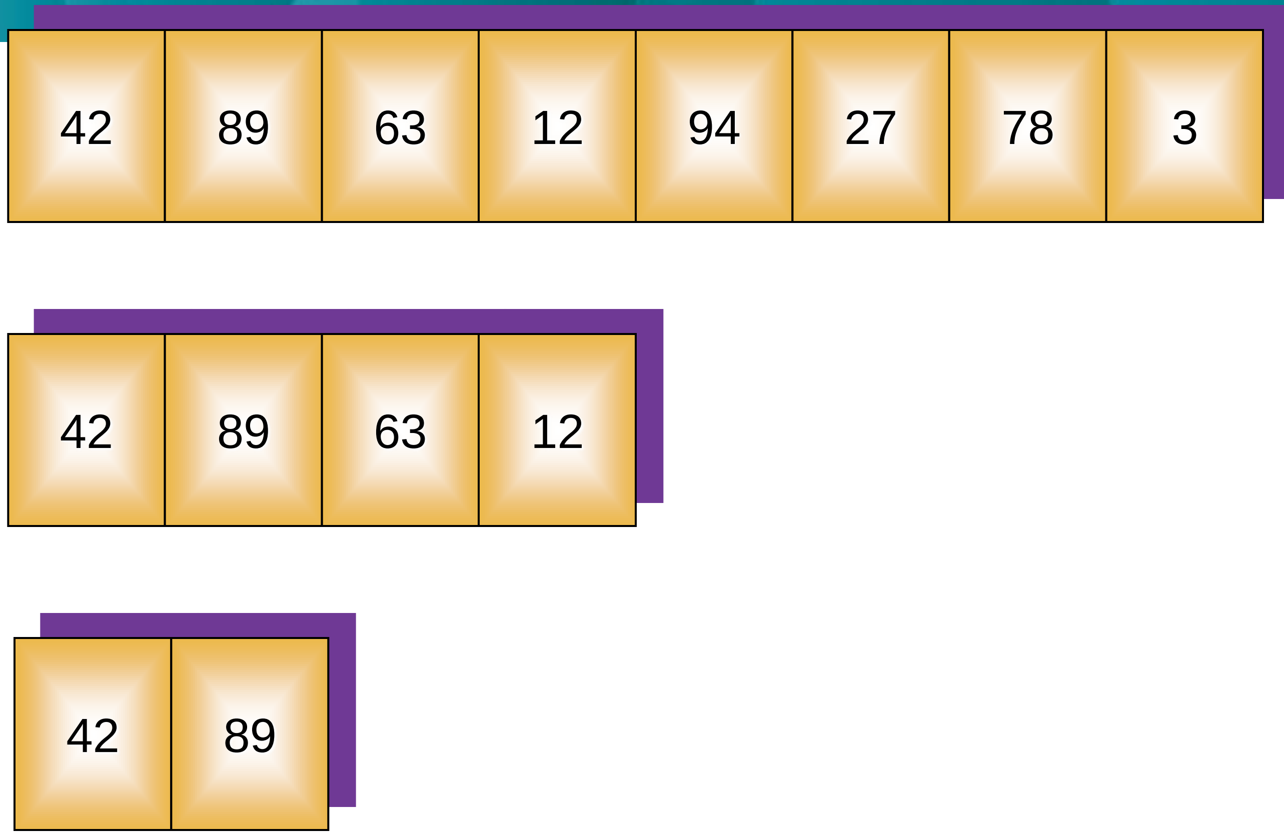
42	89	63	12	94	27	78	3

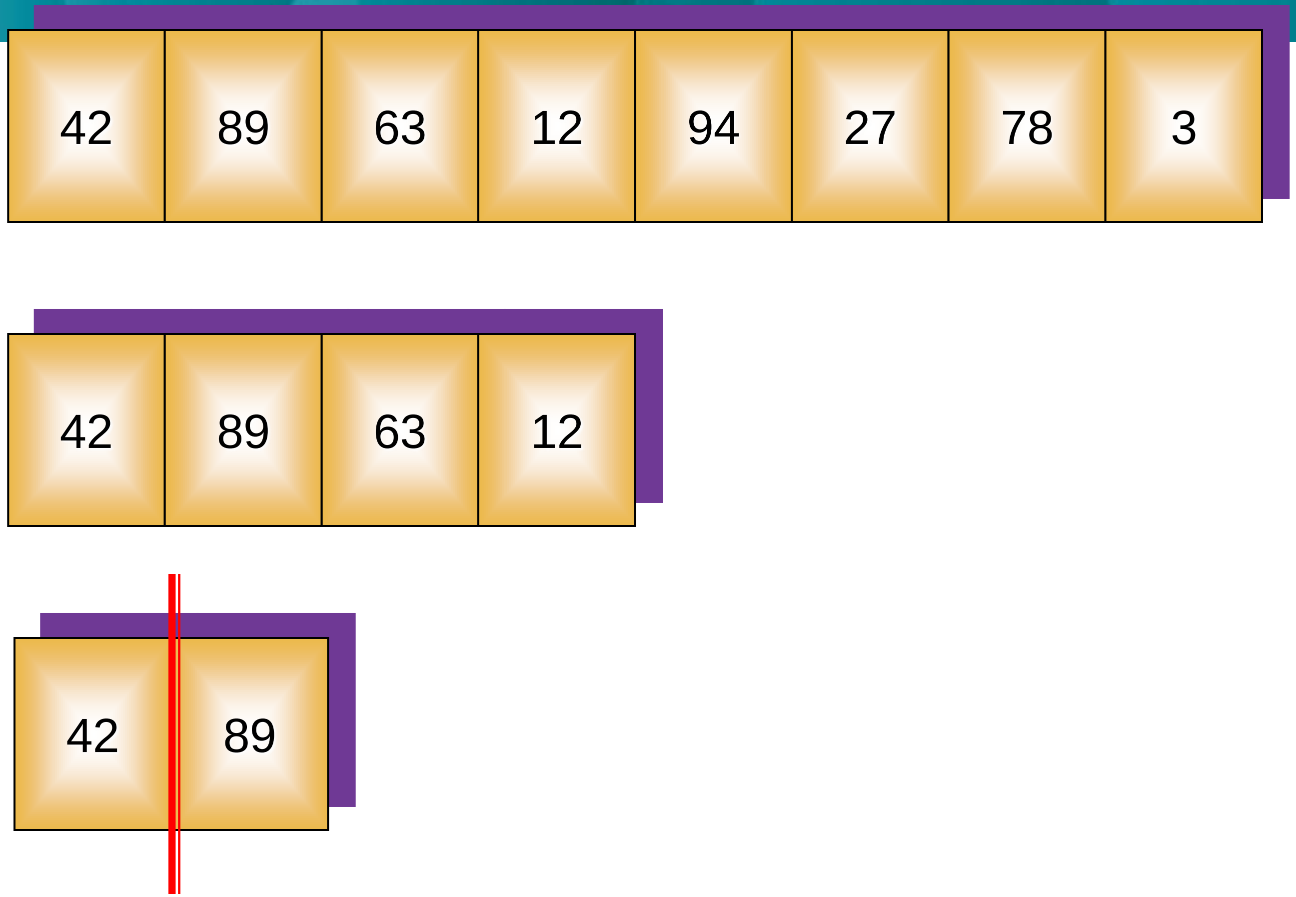
42	89	63	12

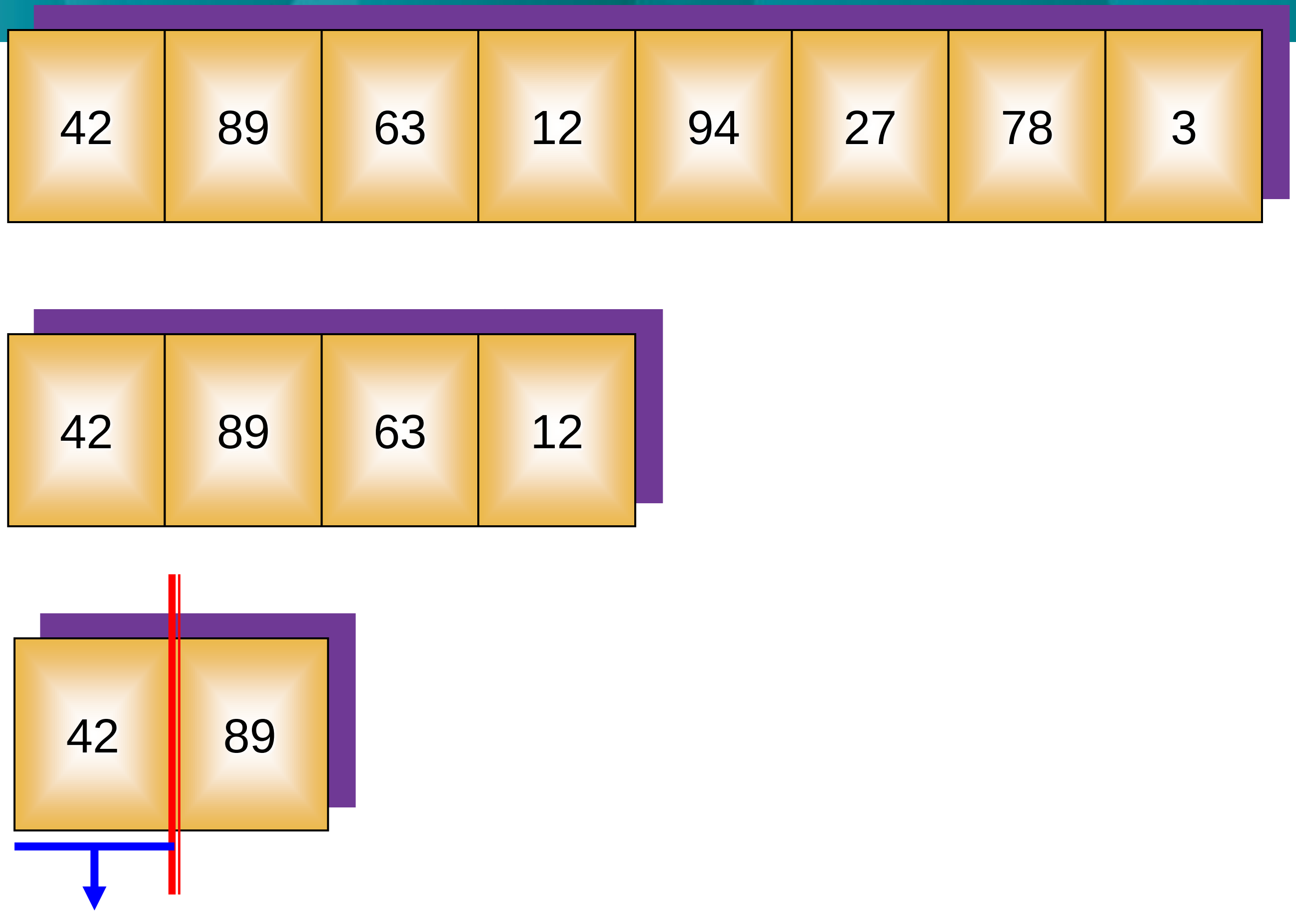
42	89	63	12	94	27	78	3

42	89	63	12


42 89 63 12 94 27 78 3
42 89 63 12
42 89


42 89 63 12 94 27 78 3
42 89 63 12
42 89


42
89
63
12
94
27
78
3
42
89
63
12
42
89

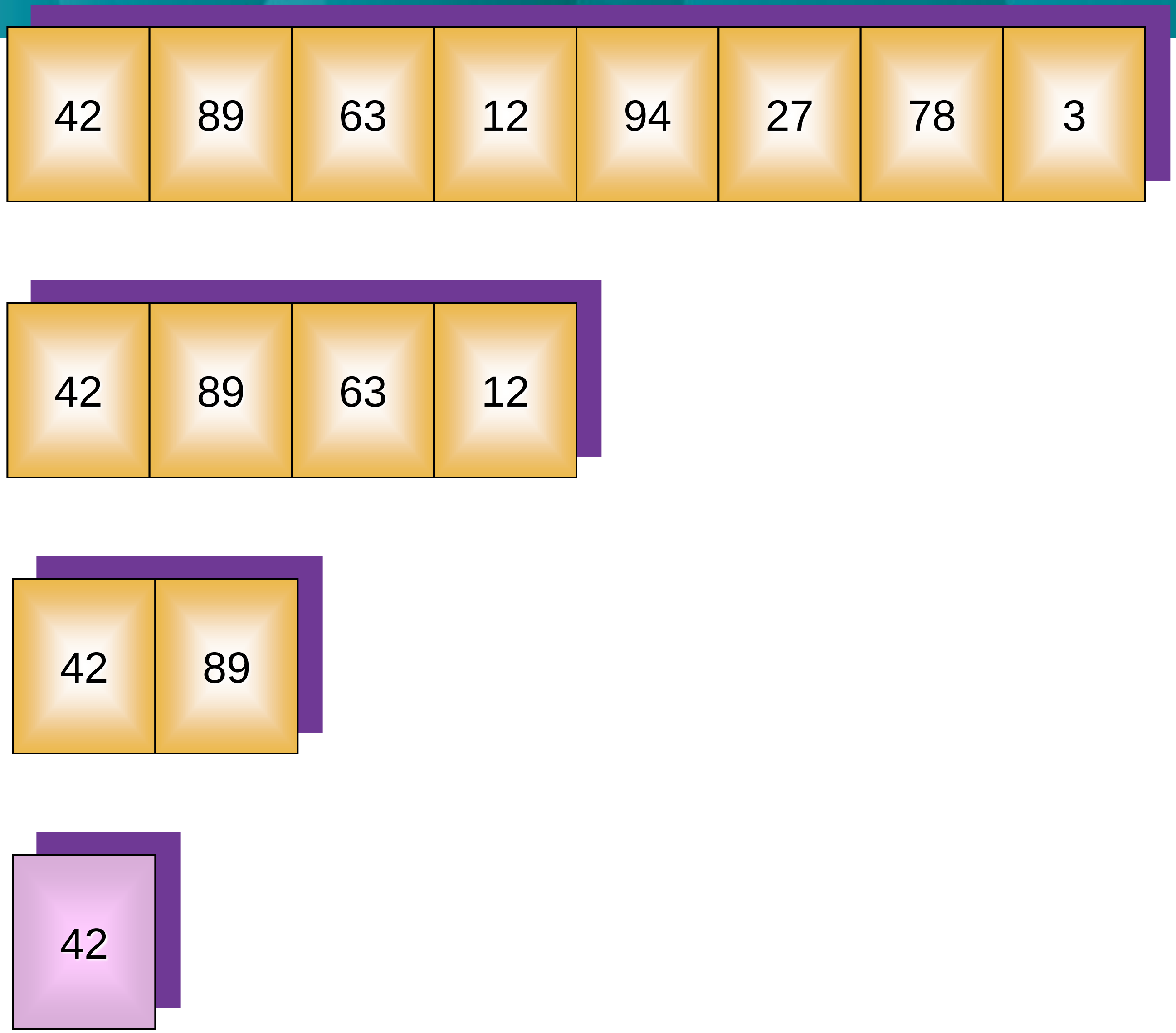

42 89 63 12 94 27 78 3
42 89 63 12
42 89

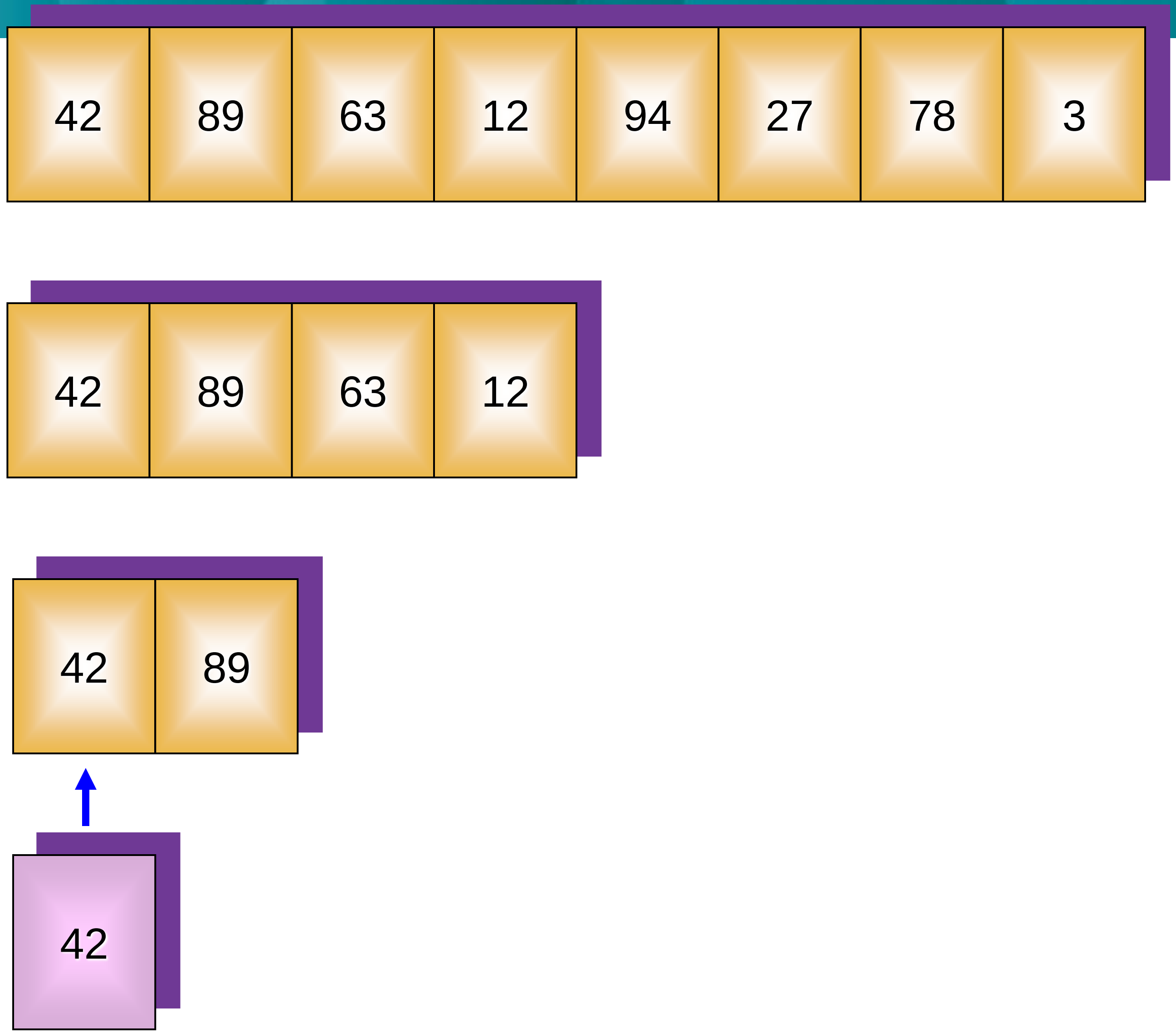
42	89	63	12	94	27	78	3

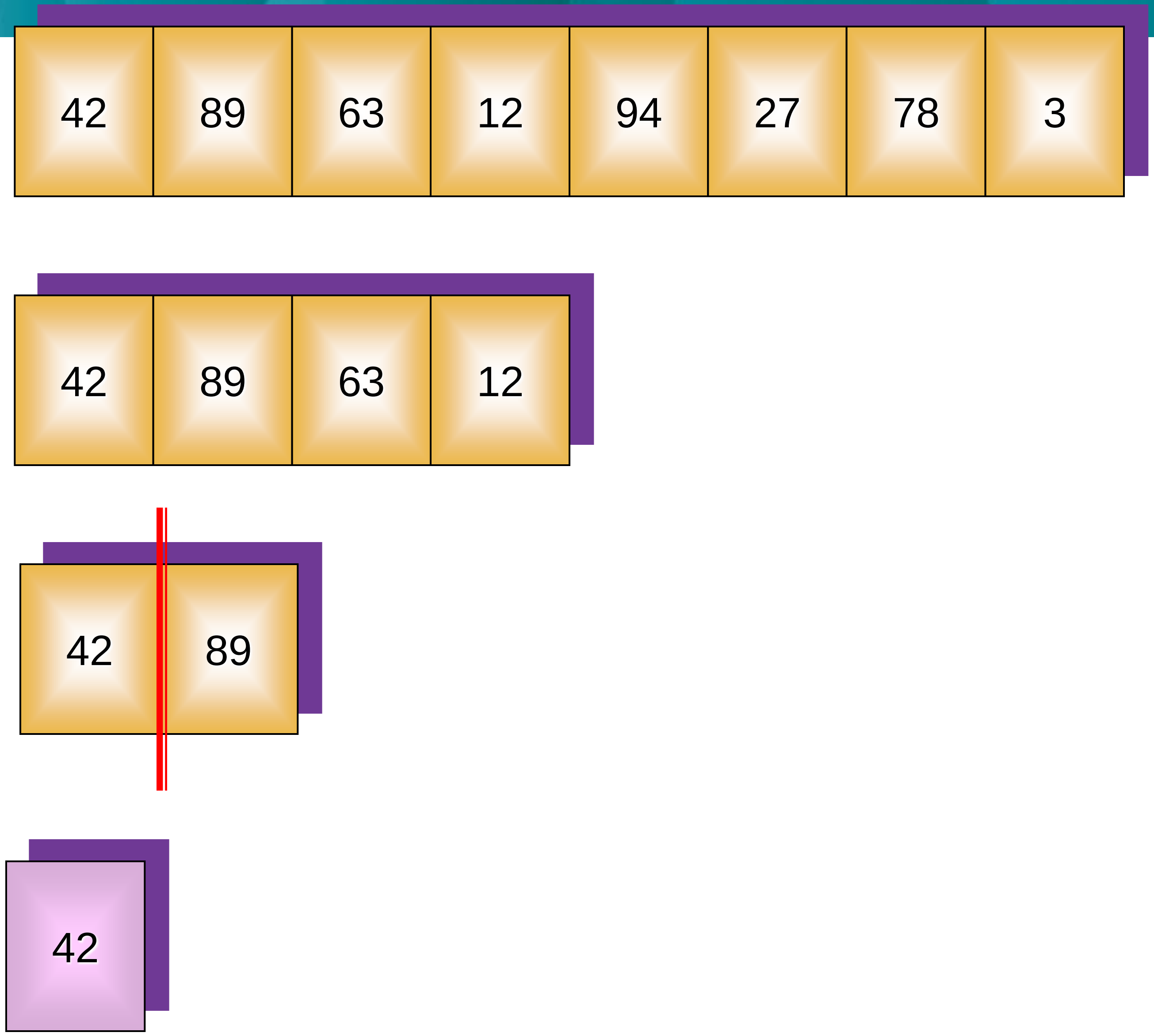
42	89	63	12

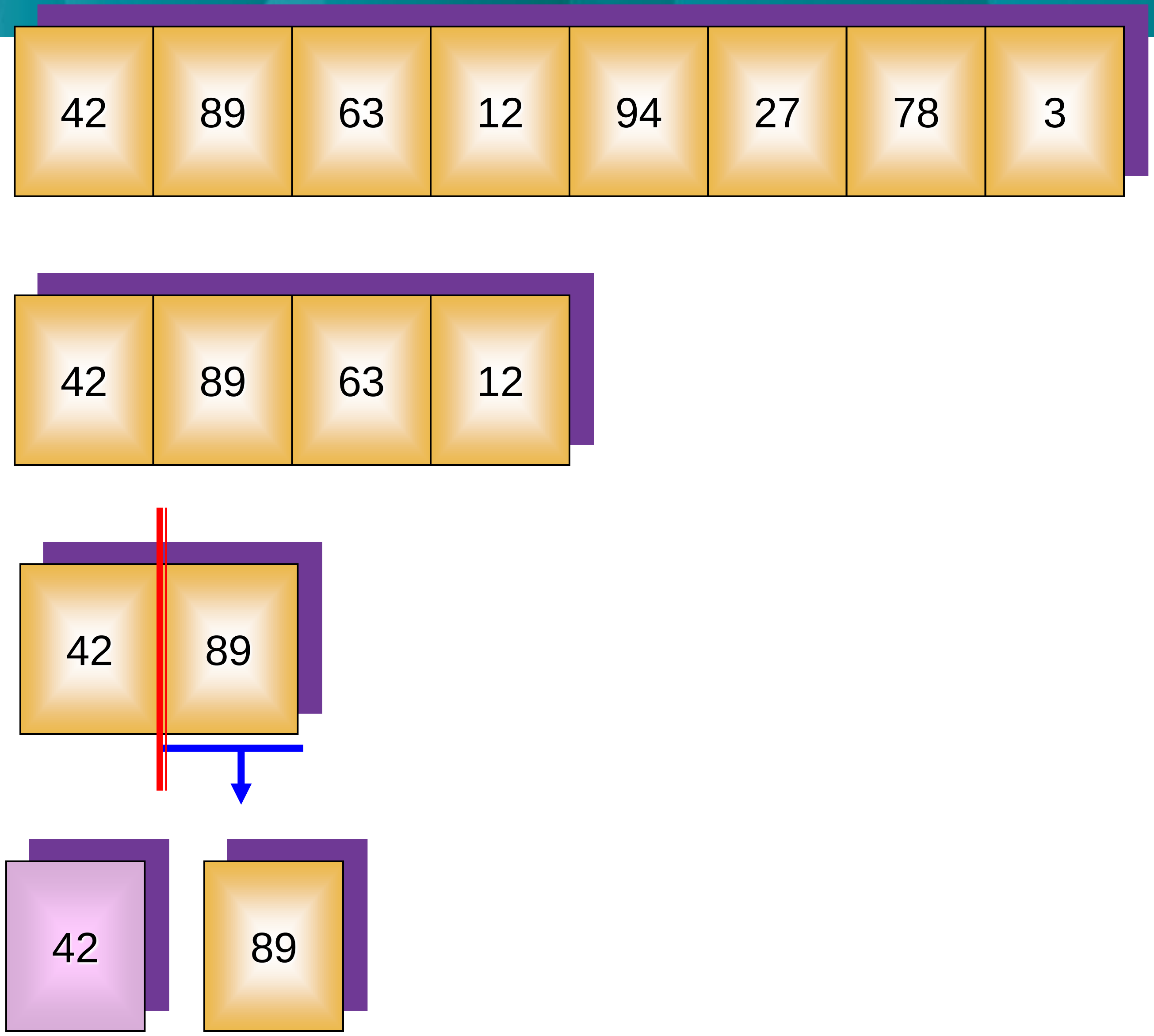
42	89

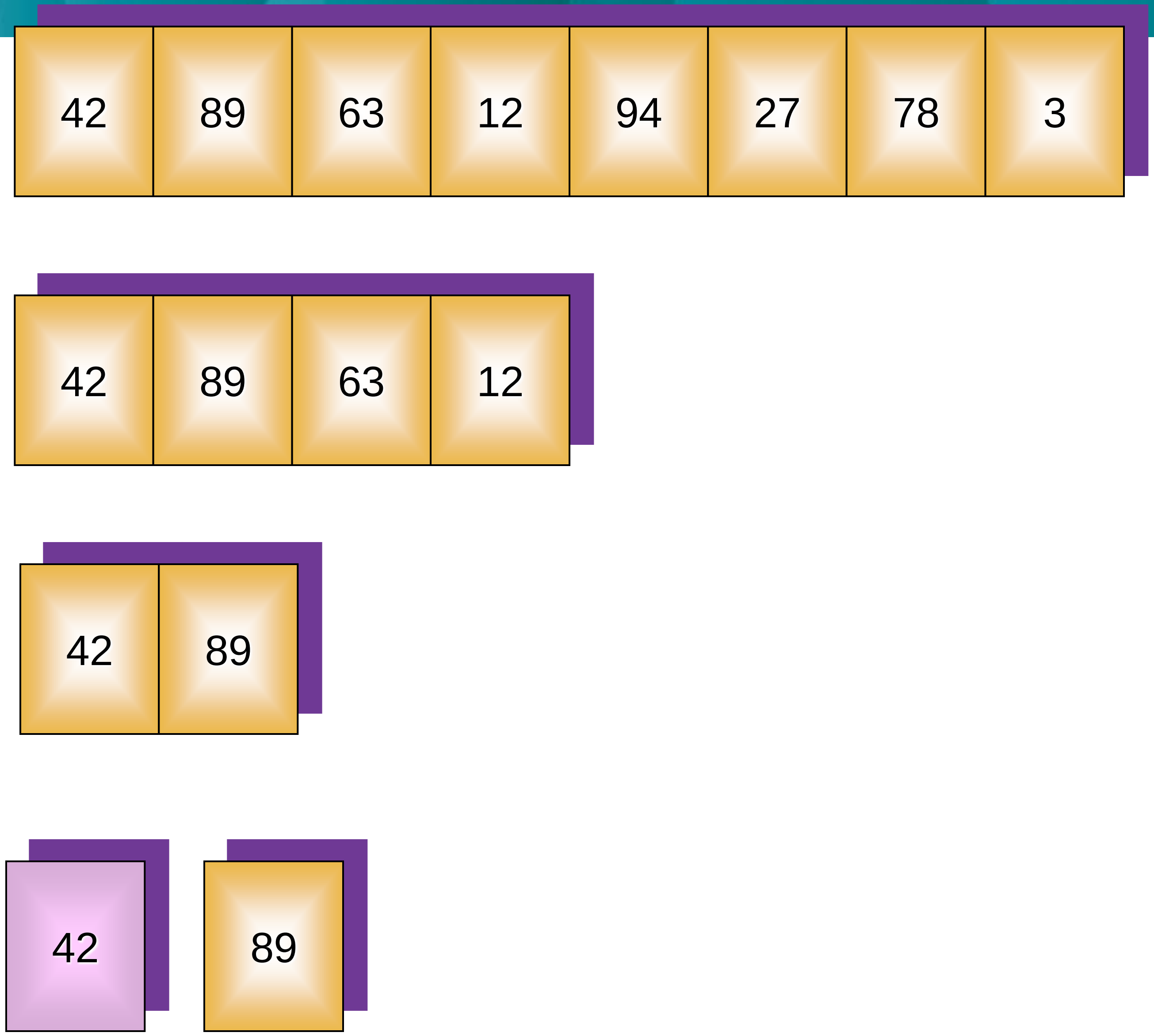
42

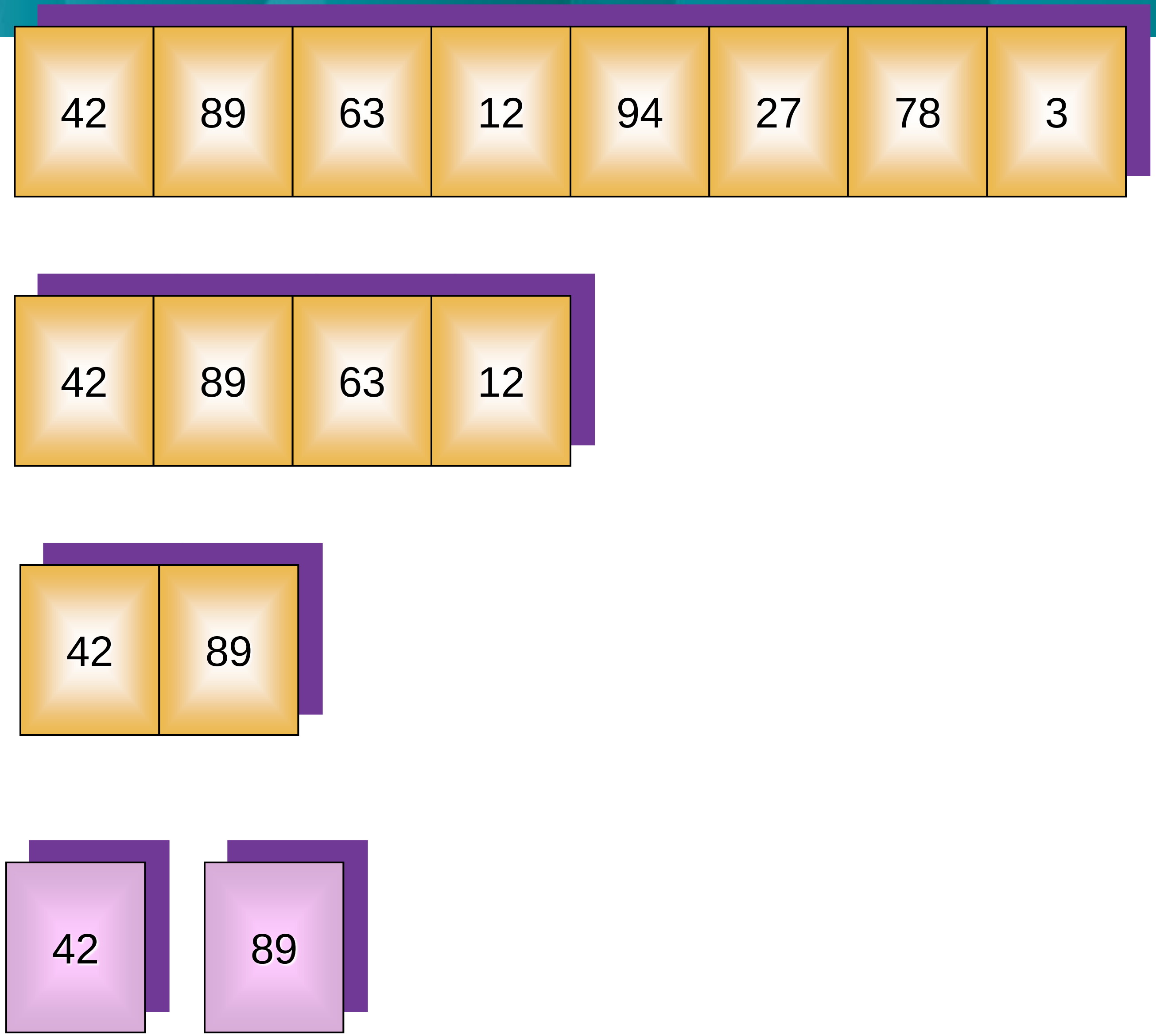

42 89 63 12 94 27 78 3
42 89 63 12
42 89
42

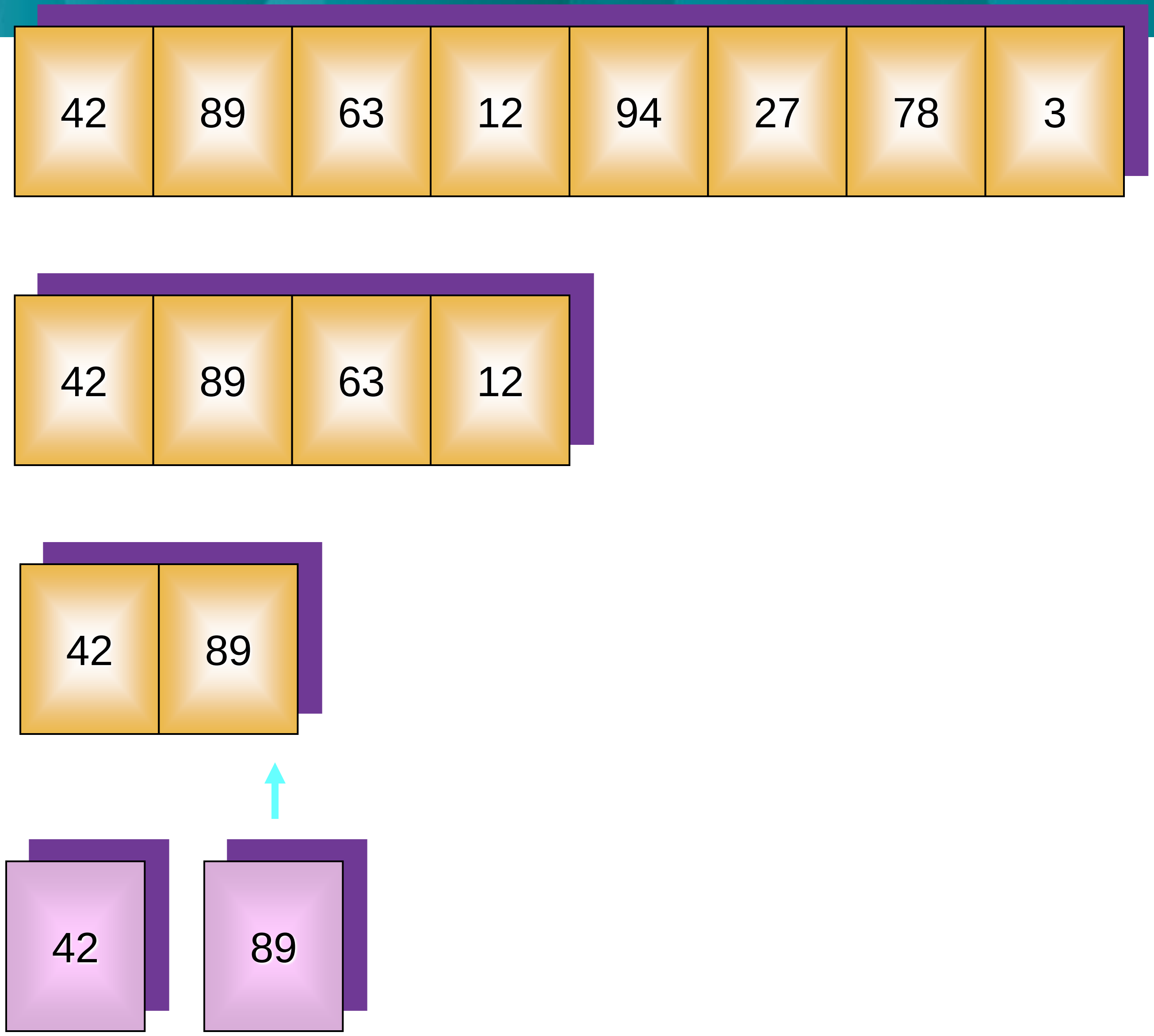

42 89 63 12 94 27 78 3
42 89 63 12
42 89
42

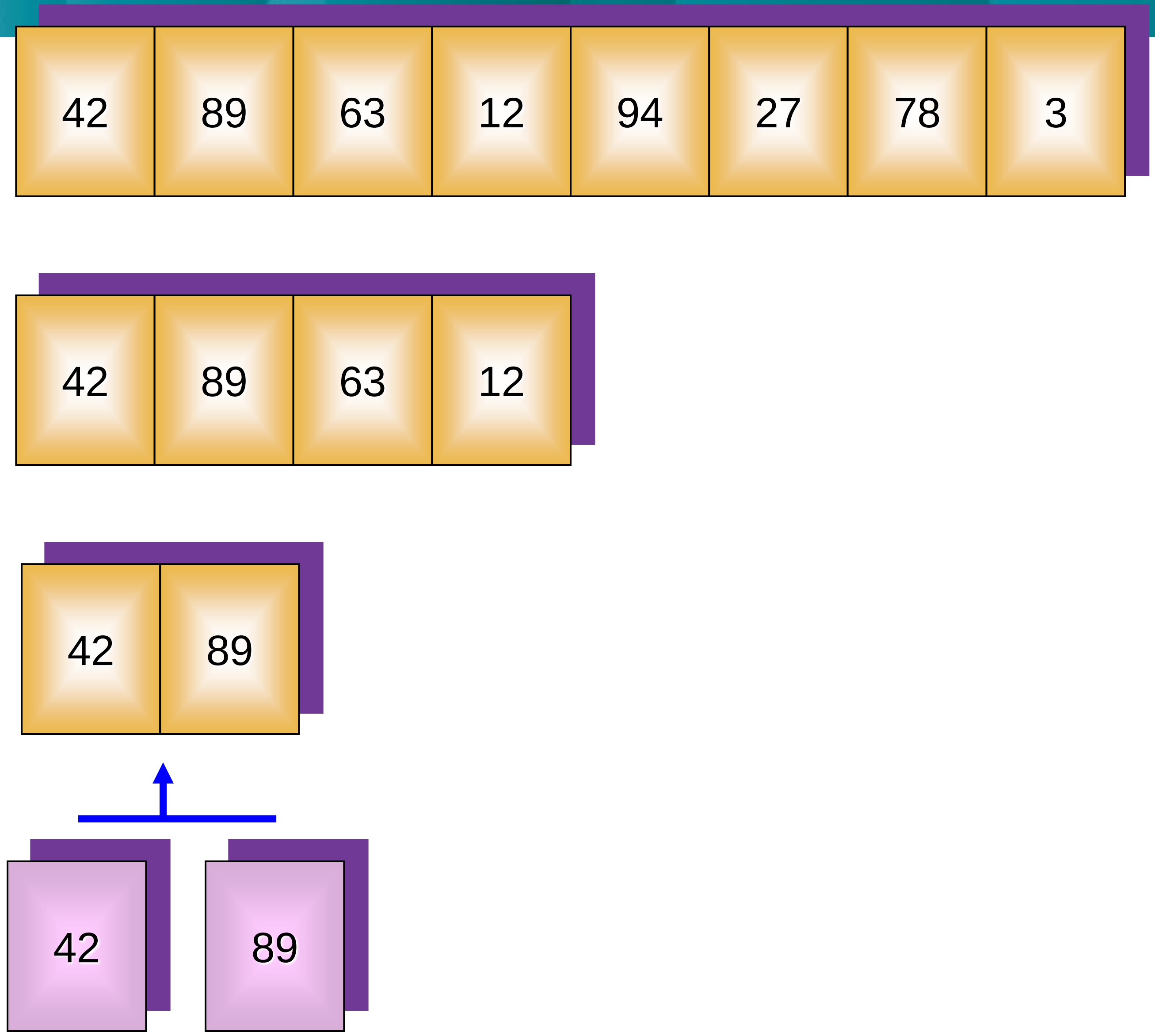

42 89 63 12 94 27 78 3
42 89 63 12
42 89
42

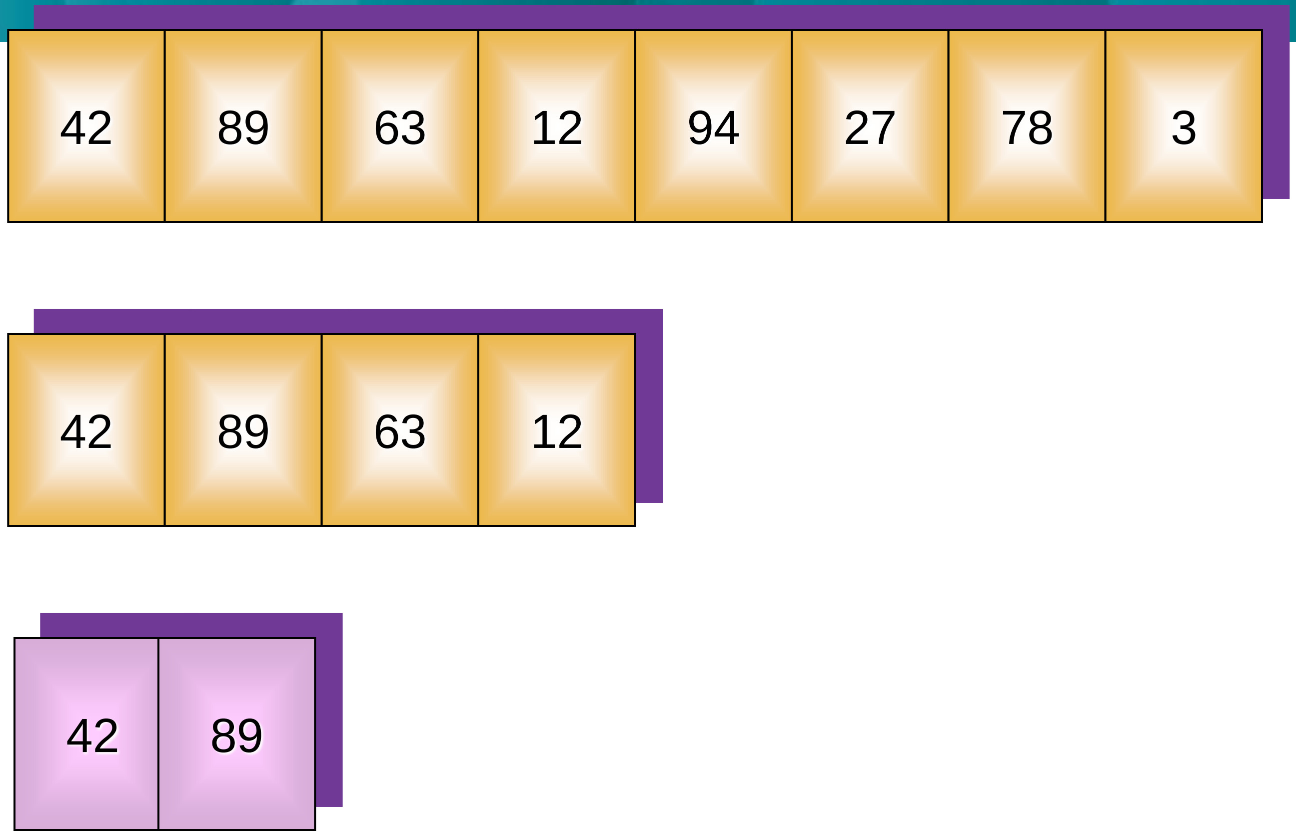

42 89 63 12 94 27 78 3
42 89 63 12
42 89
42
89

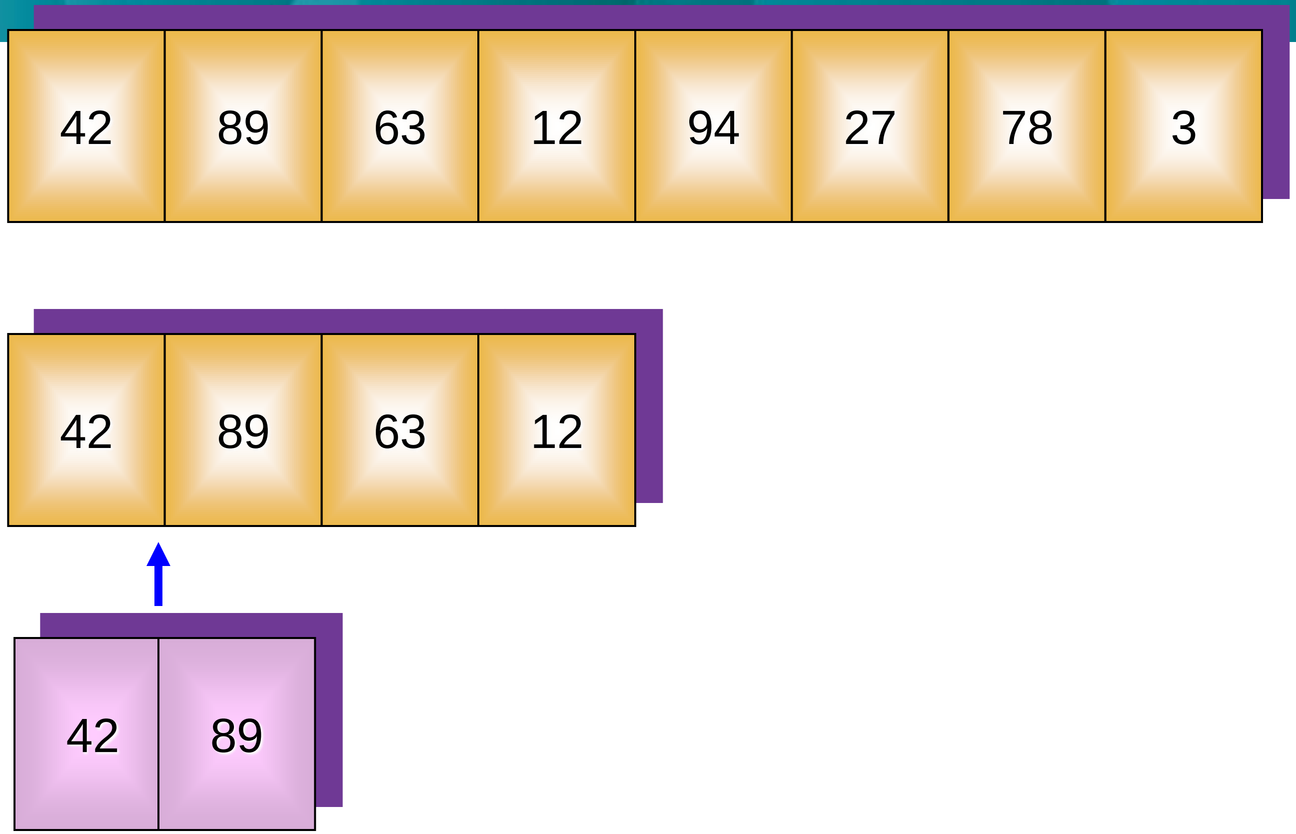

42
89
63
12
94
27
78
3
42
89
63
12
42
89
42
89

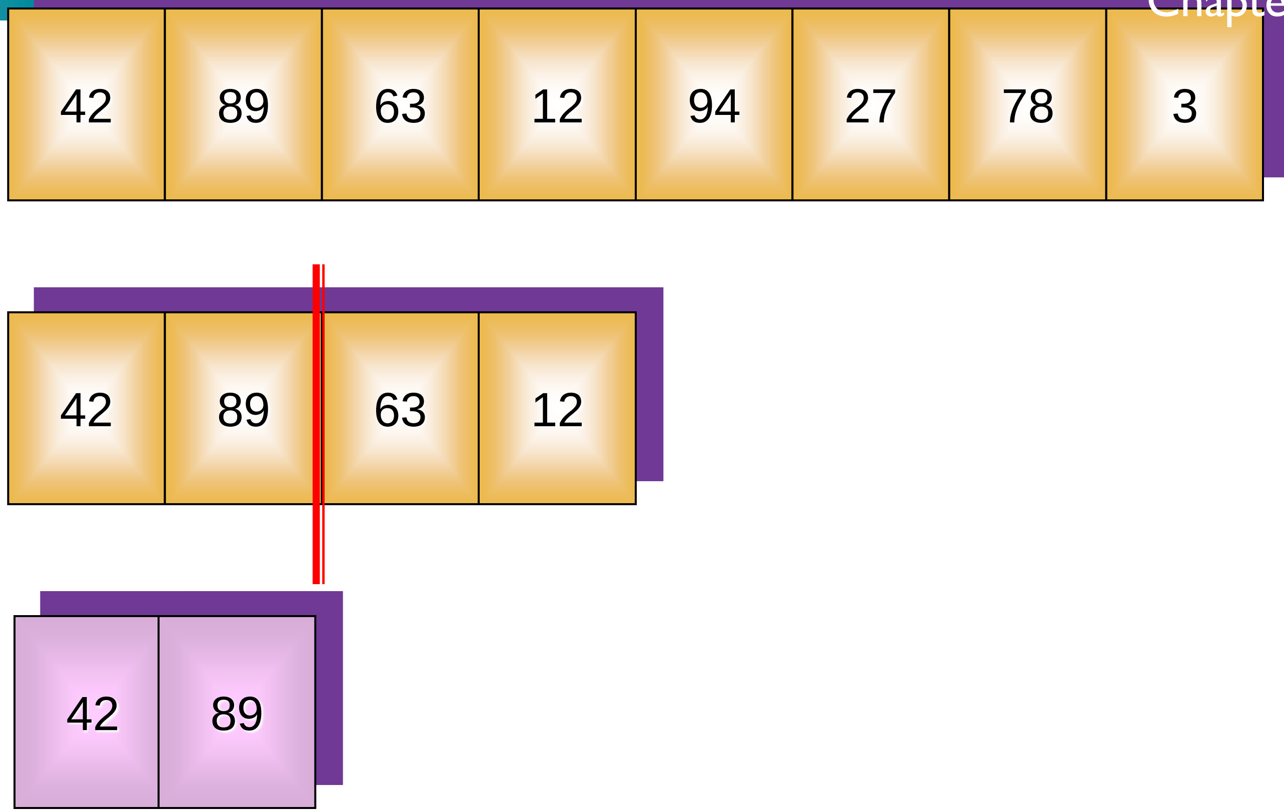

42 89 63 12 94 27 78 3
42 89 63 12
42 89
42
89

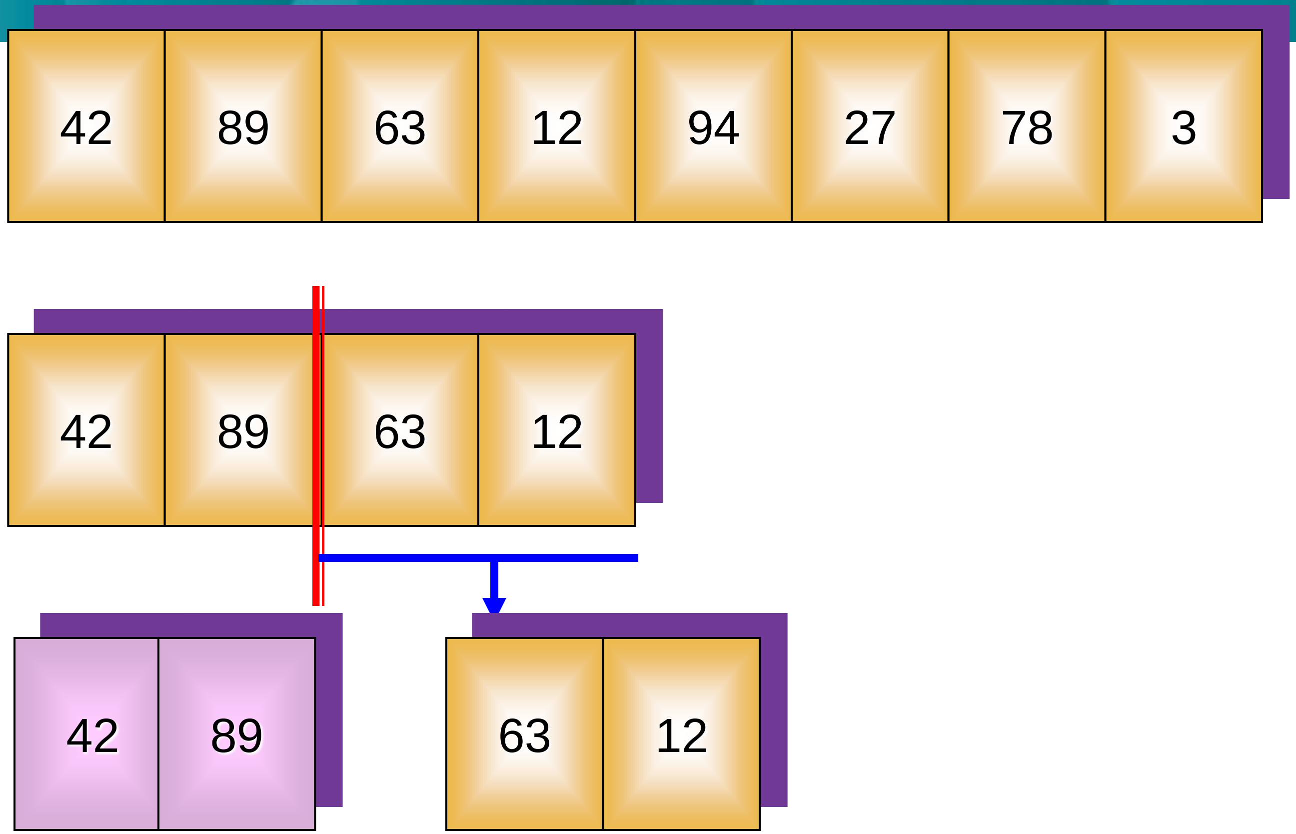

42 89 63 12 94 27 78 3
42 89 63 12
42 89
42
89

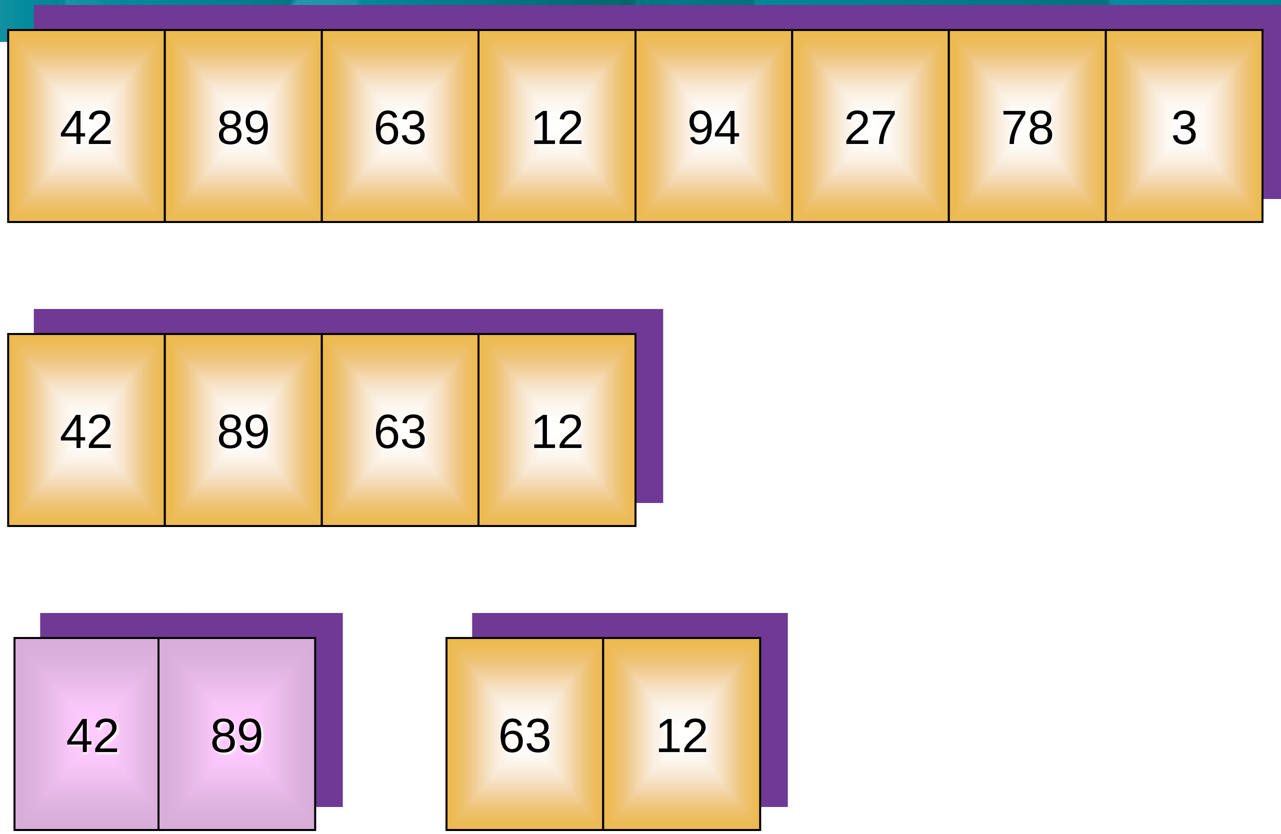

42
89
63
12
94
27
78
3
42
89
63
12
42
89
42
89

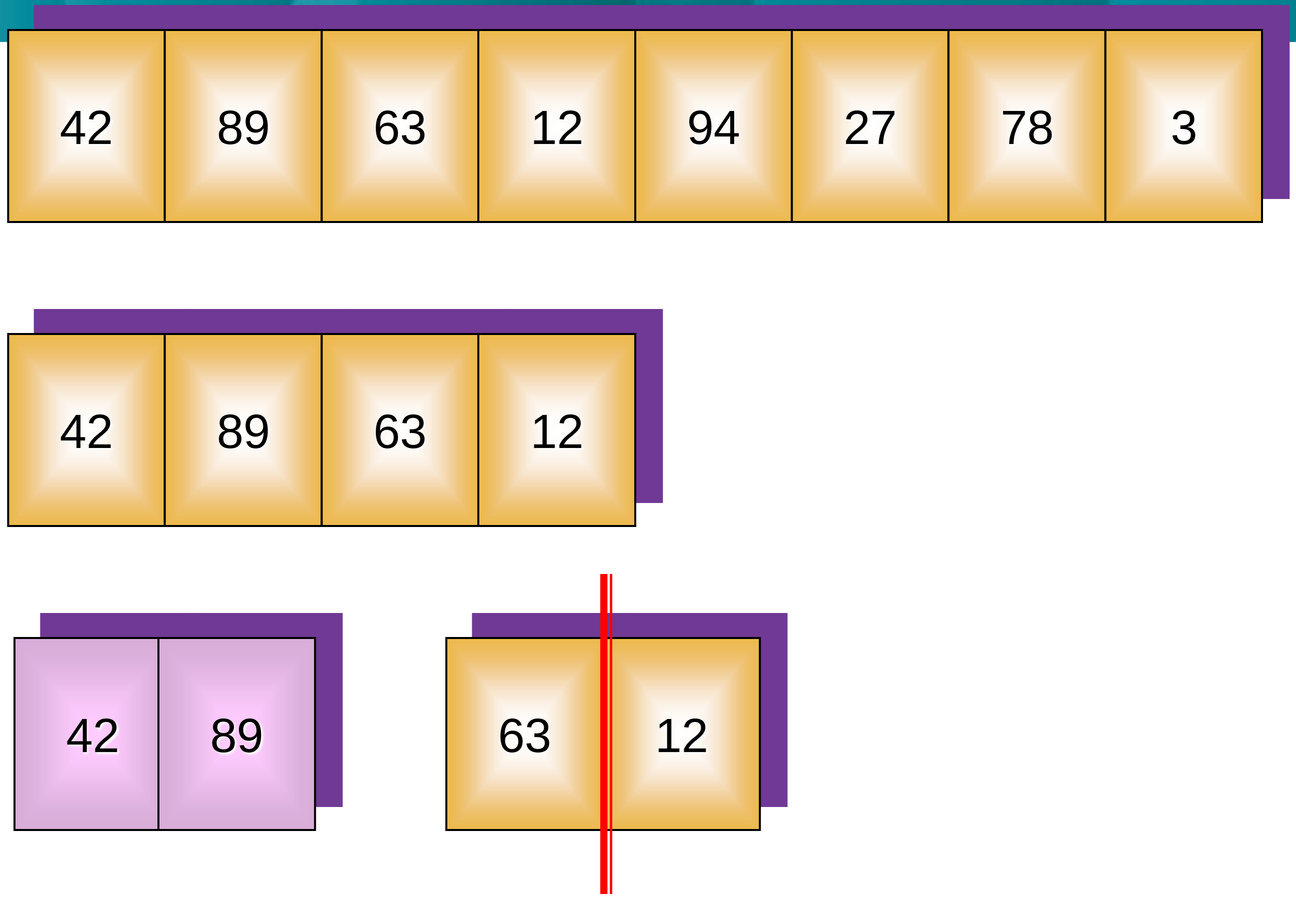

42 89 63 12 94 27 78 3
42 89 63 12
42 89

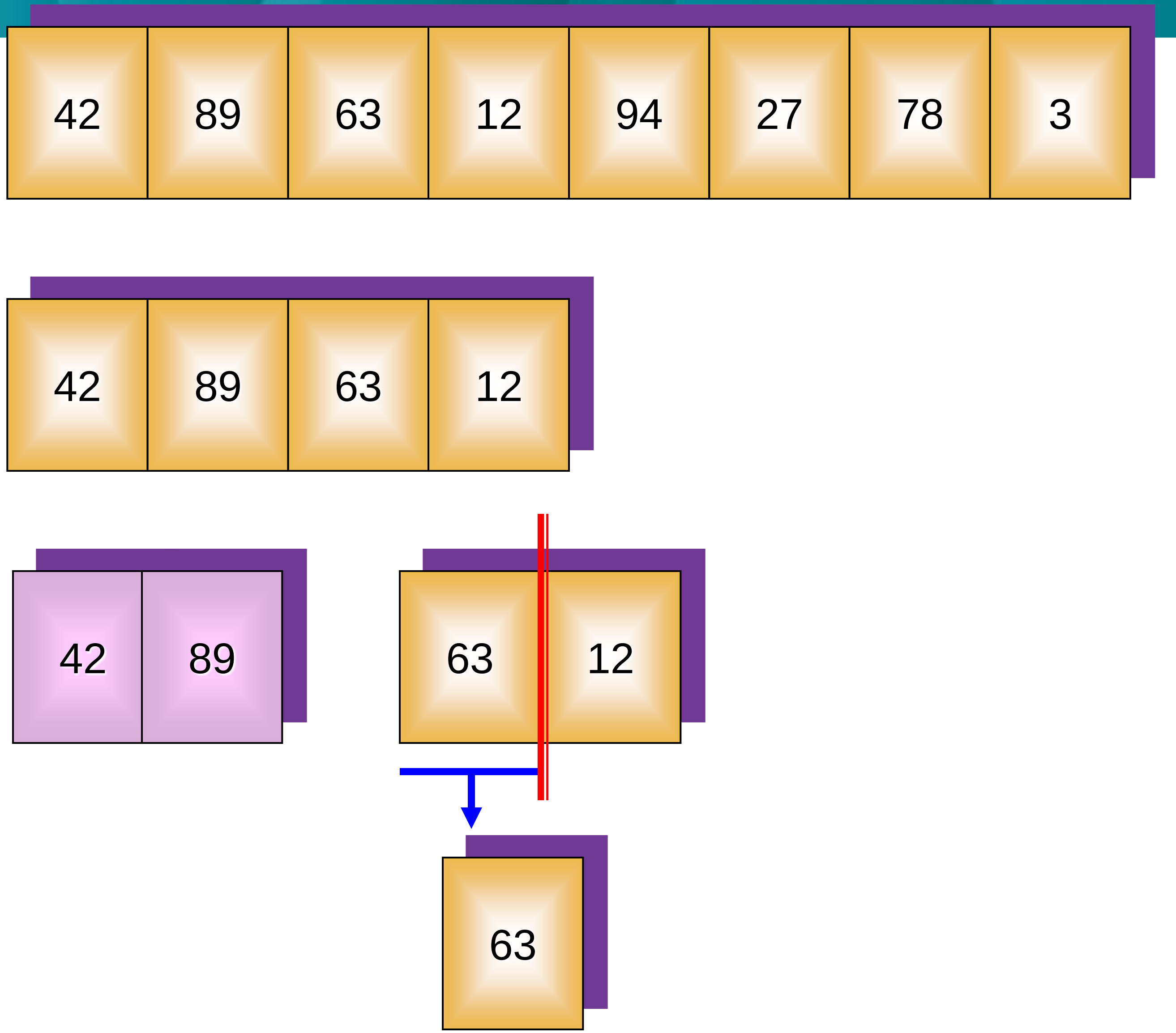

42 89 63 12 94 27 78 3
42 89 63 12
42 89

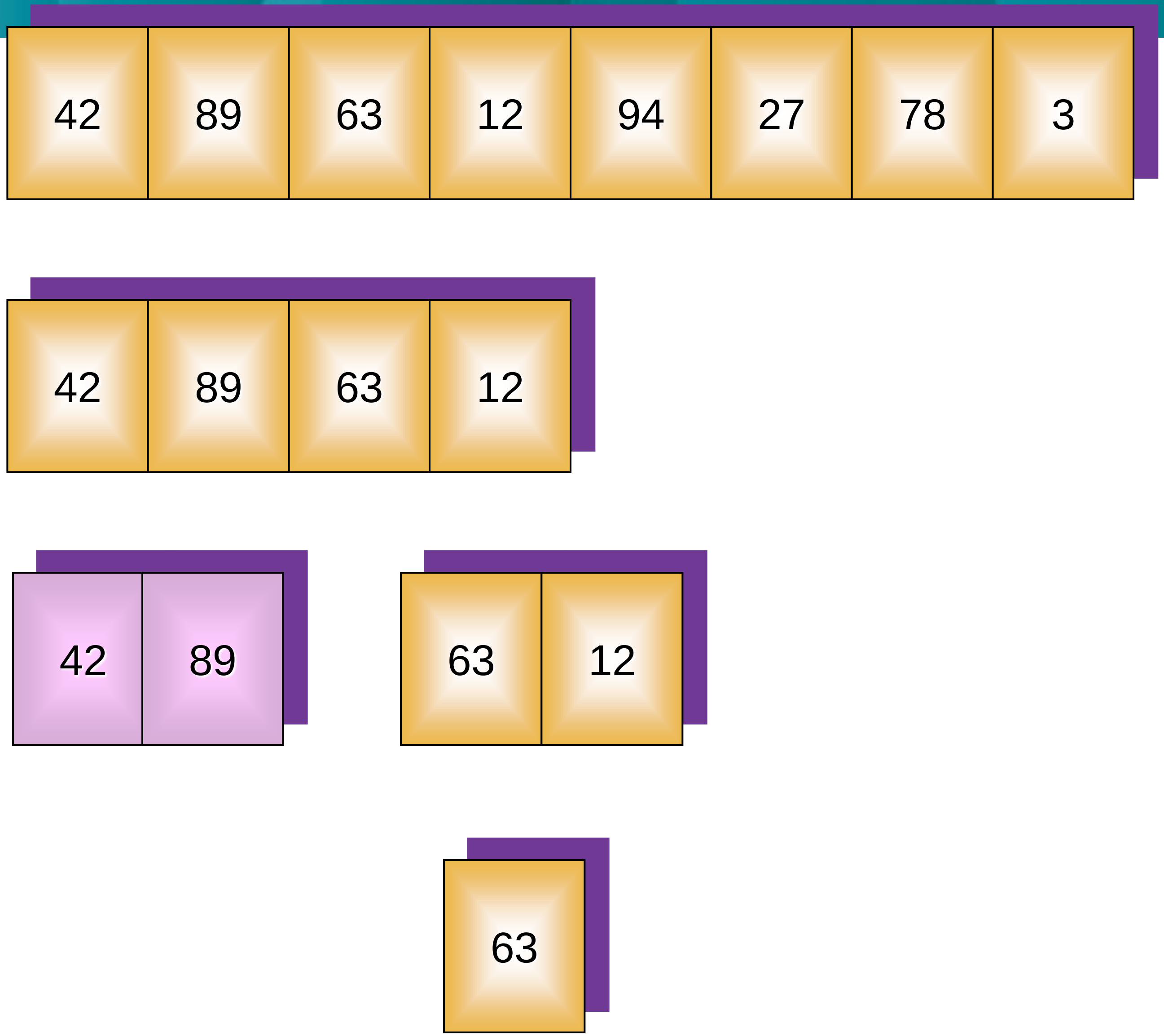

42 89 63 12 94 27 78 3
42 89 63 12
42 89

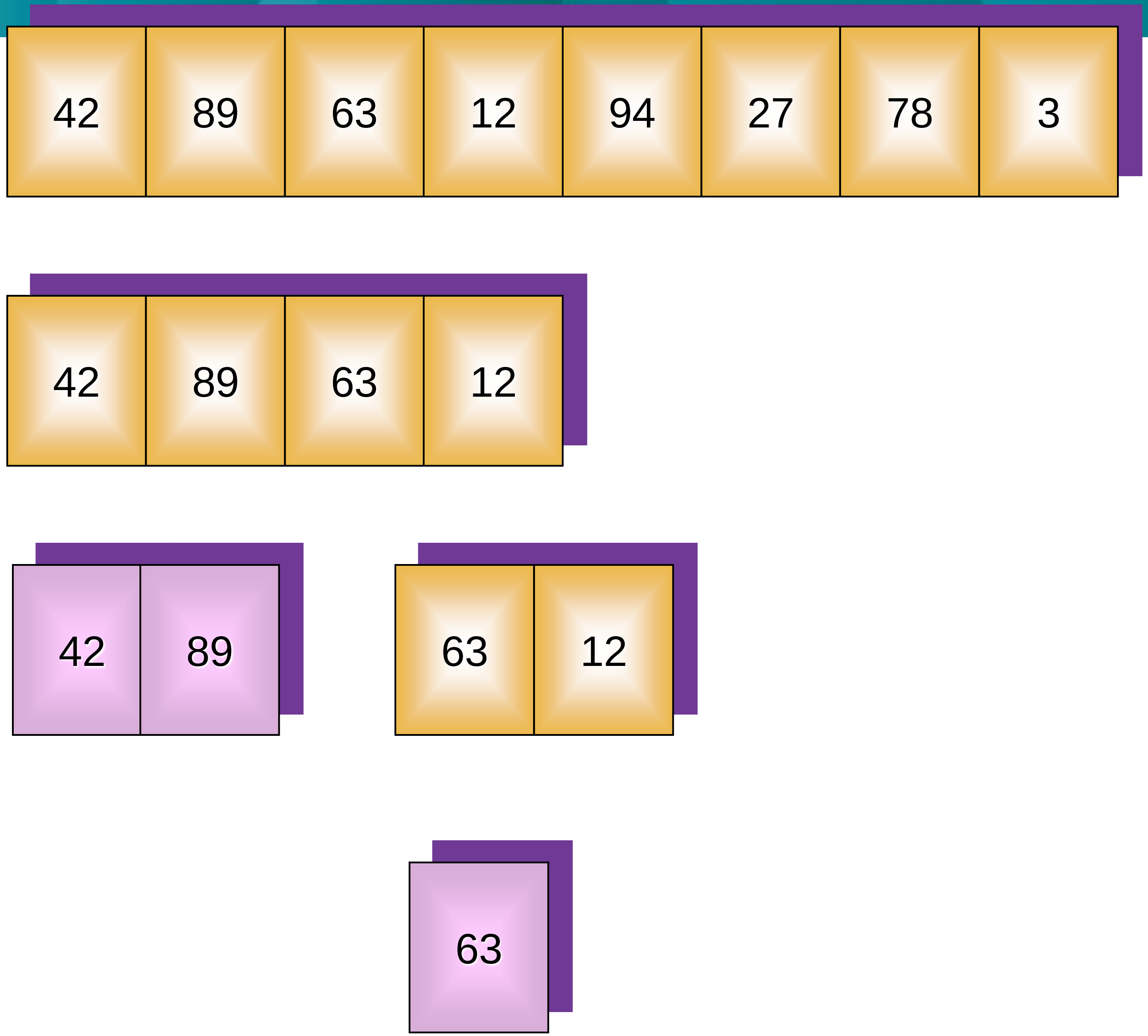

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12

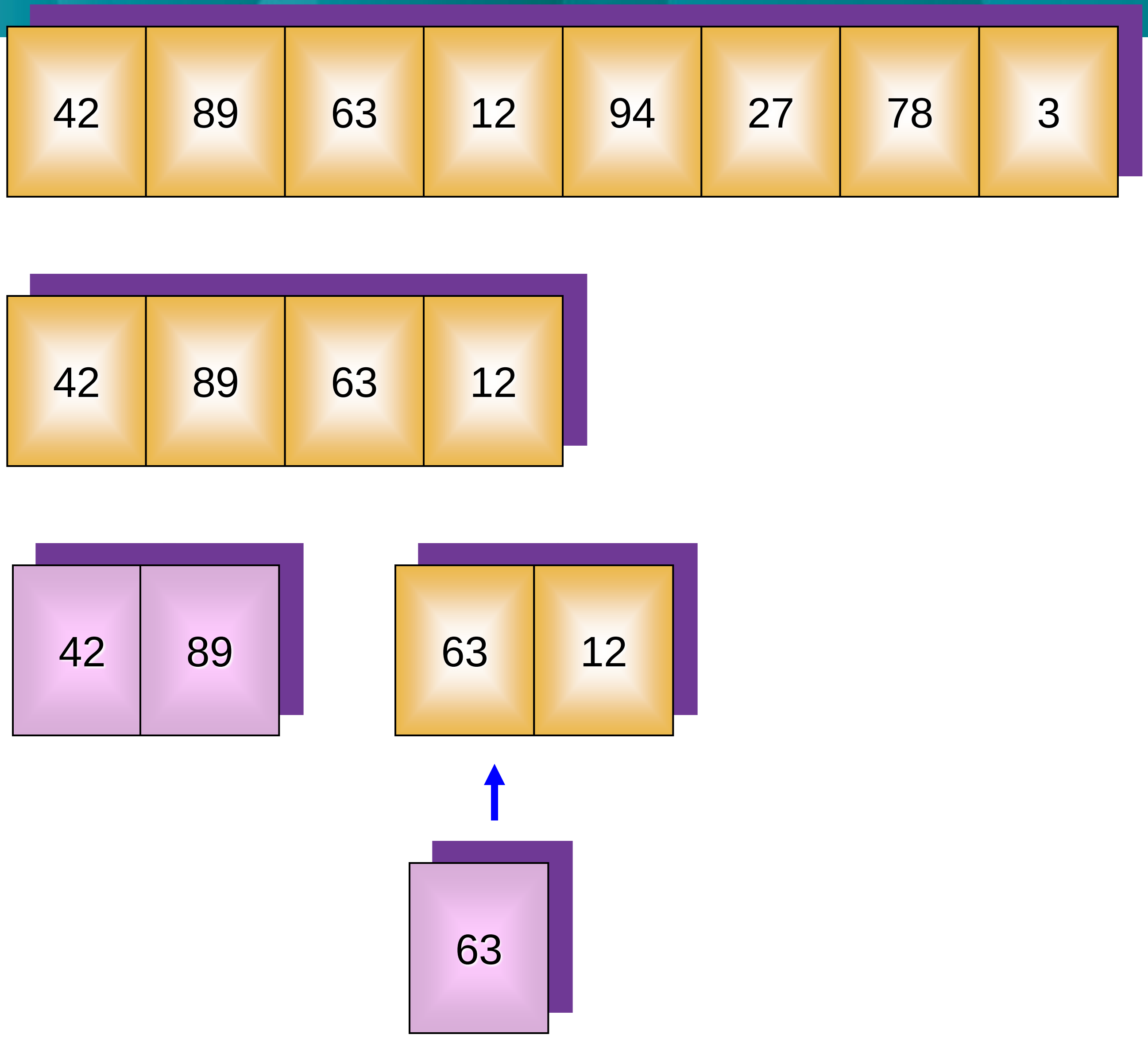

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12

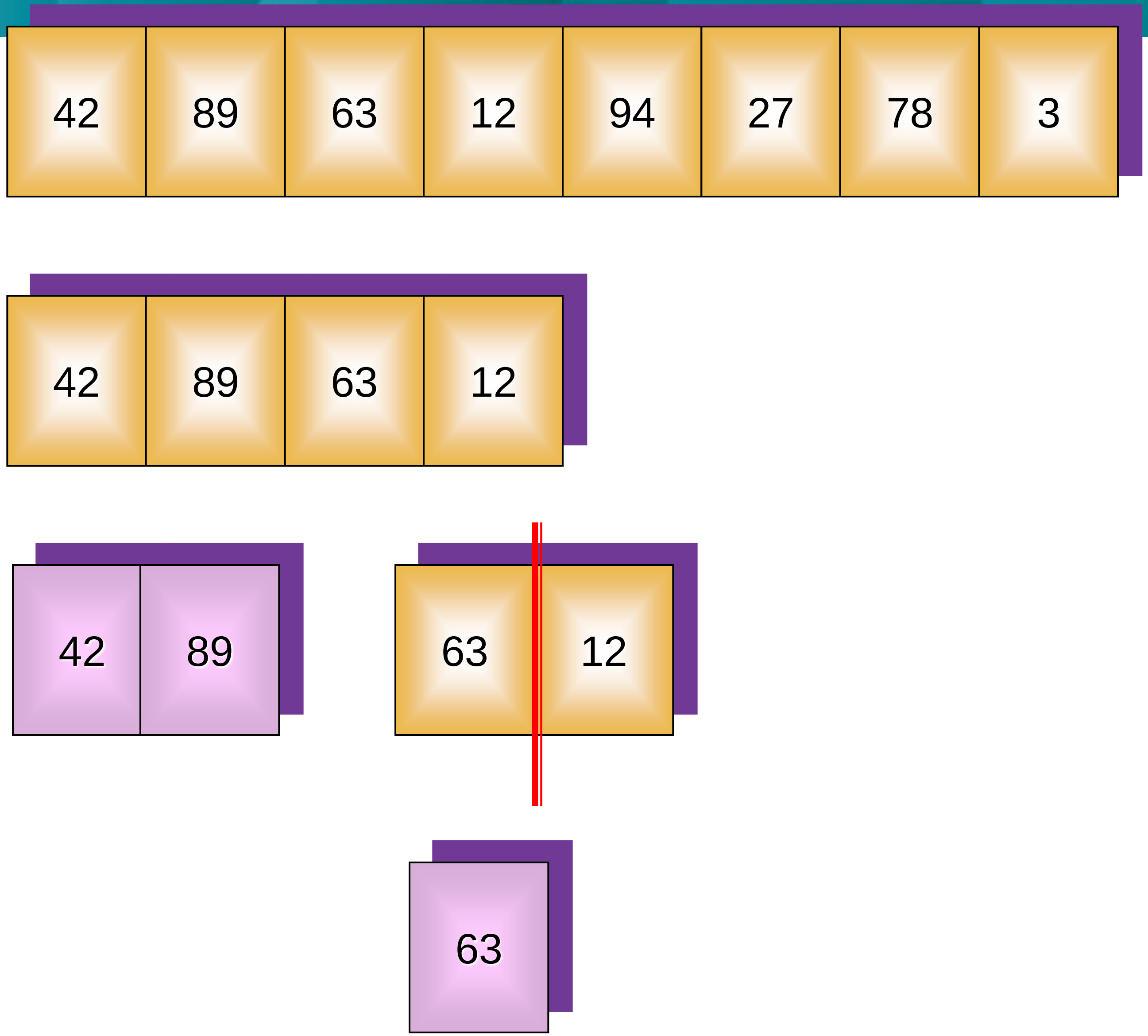

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12

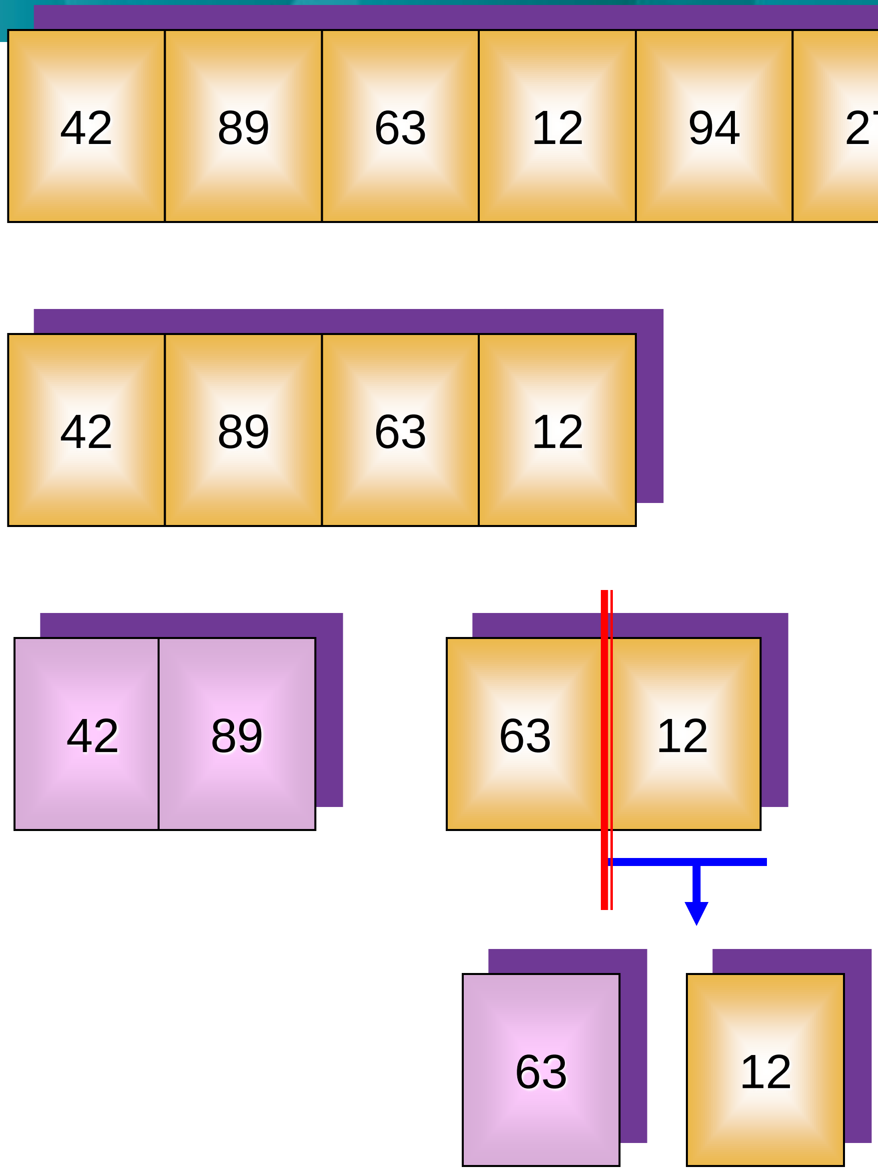

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63

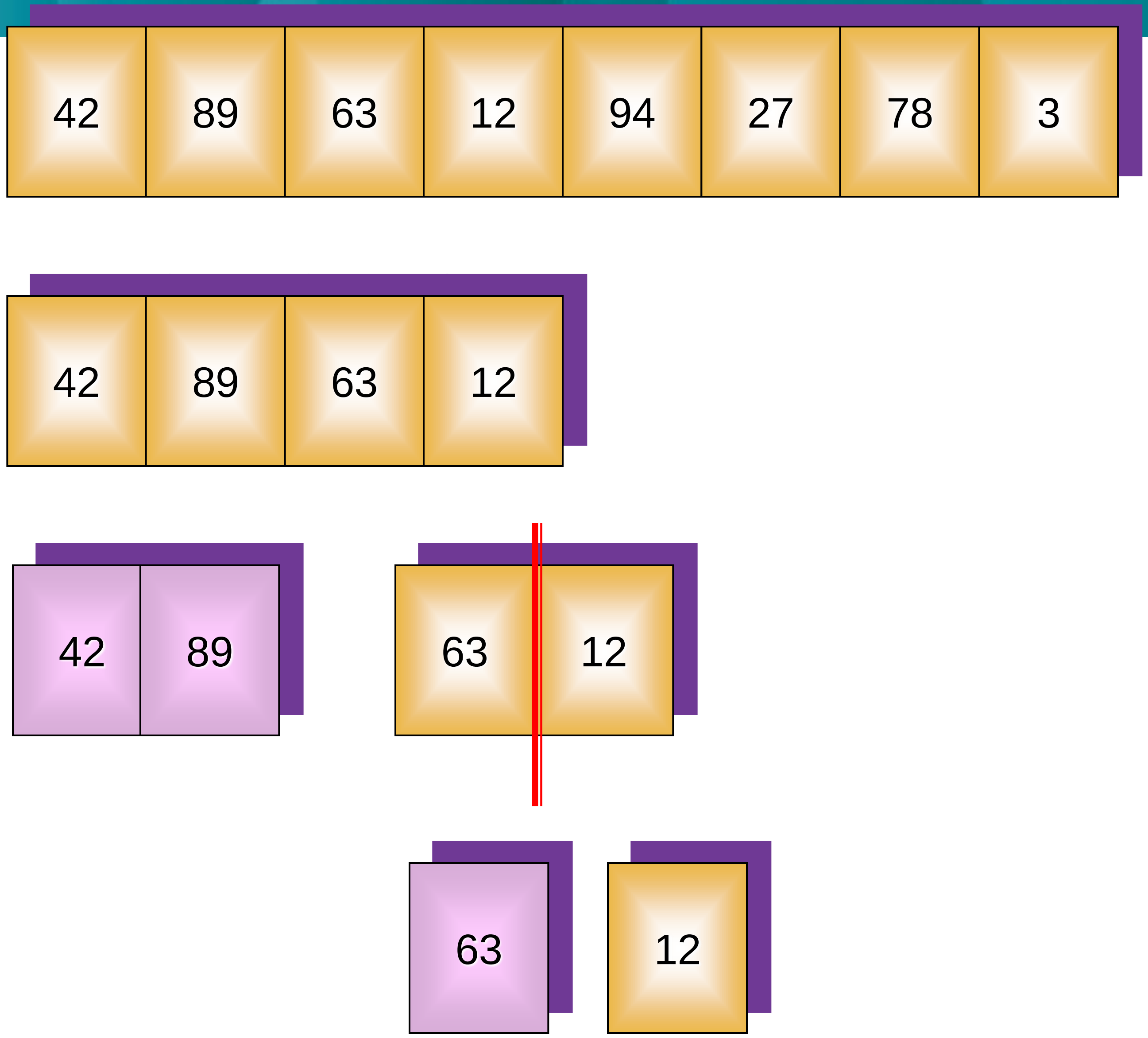

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63

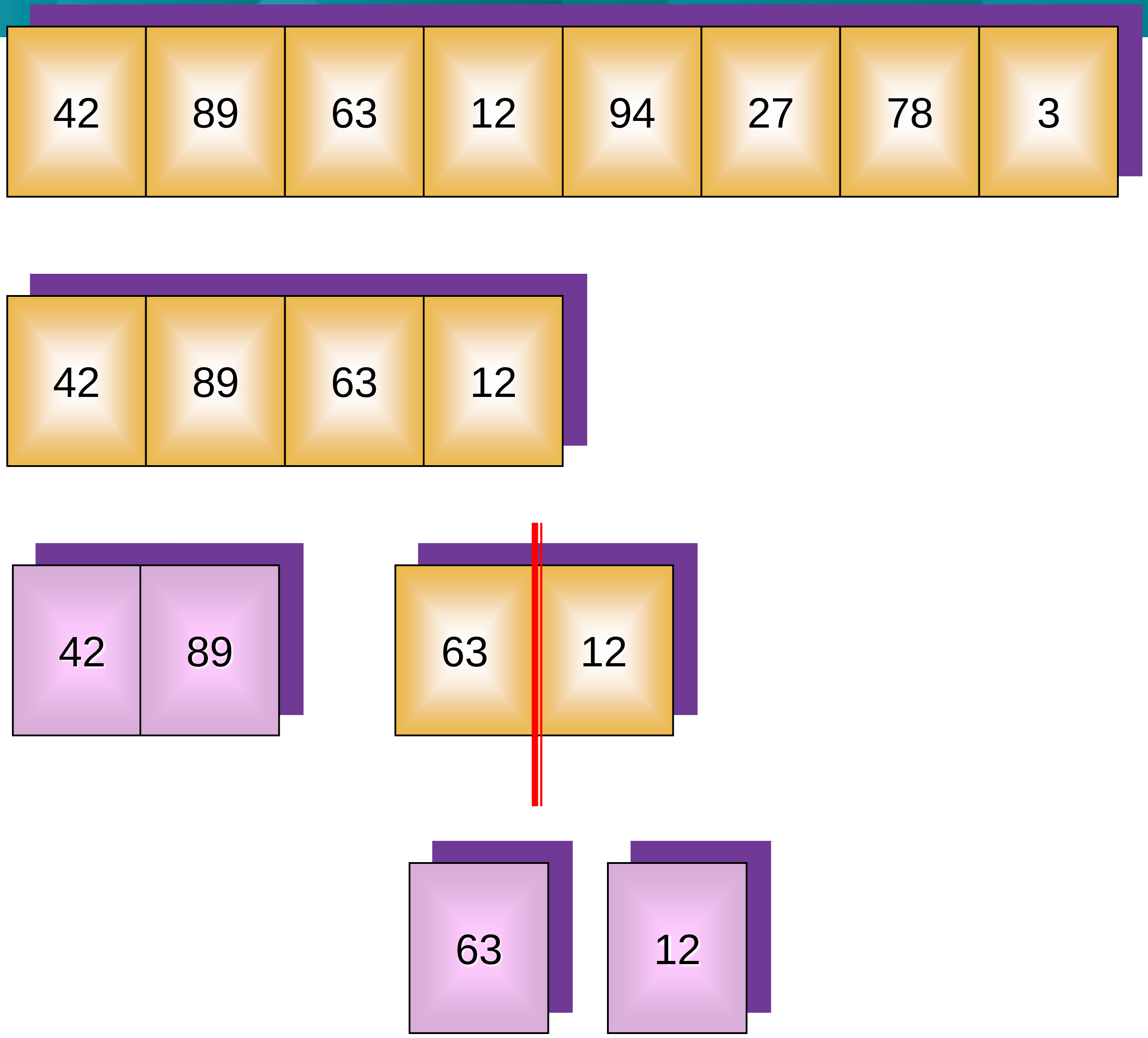

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63


42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63

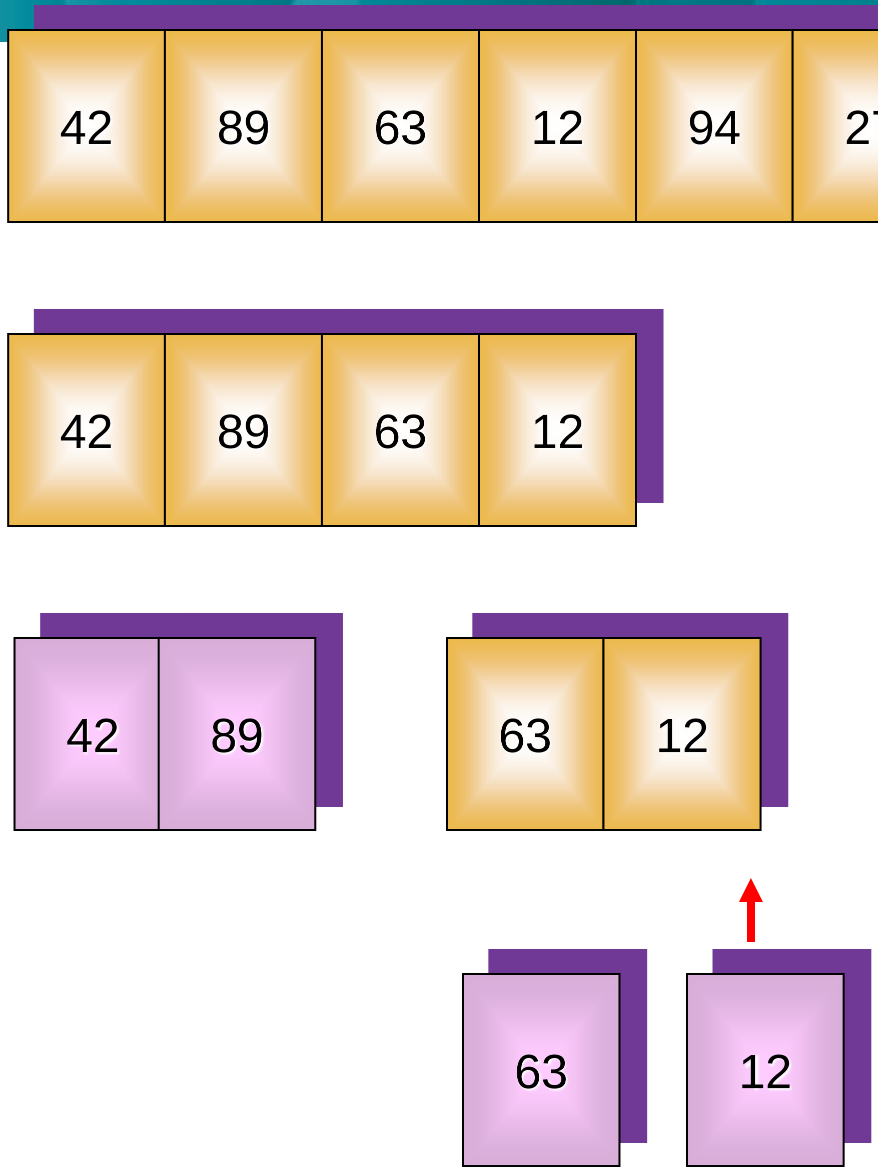

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63

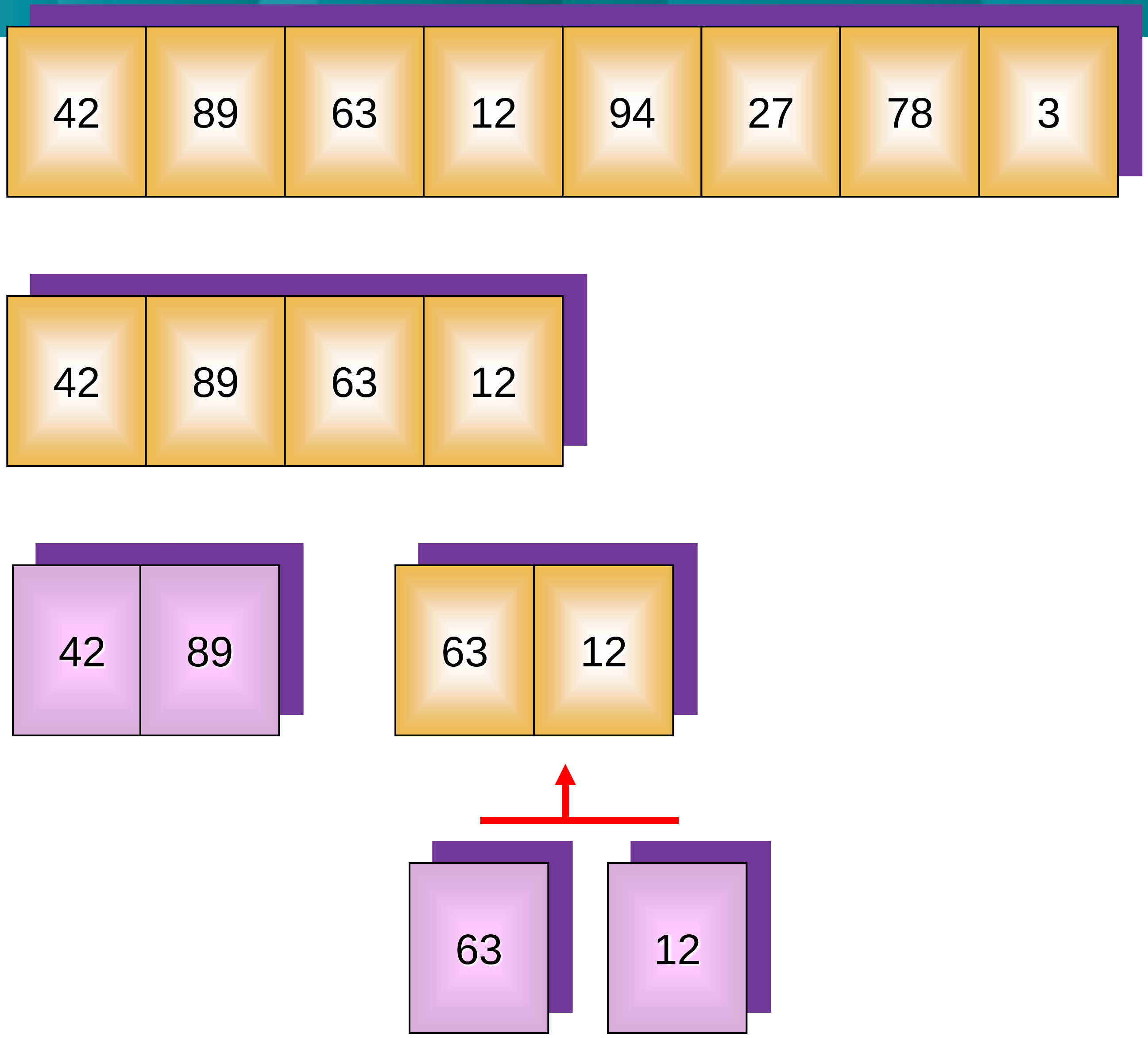

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63
12

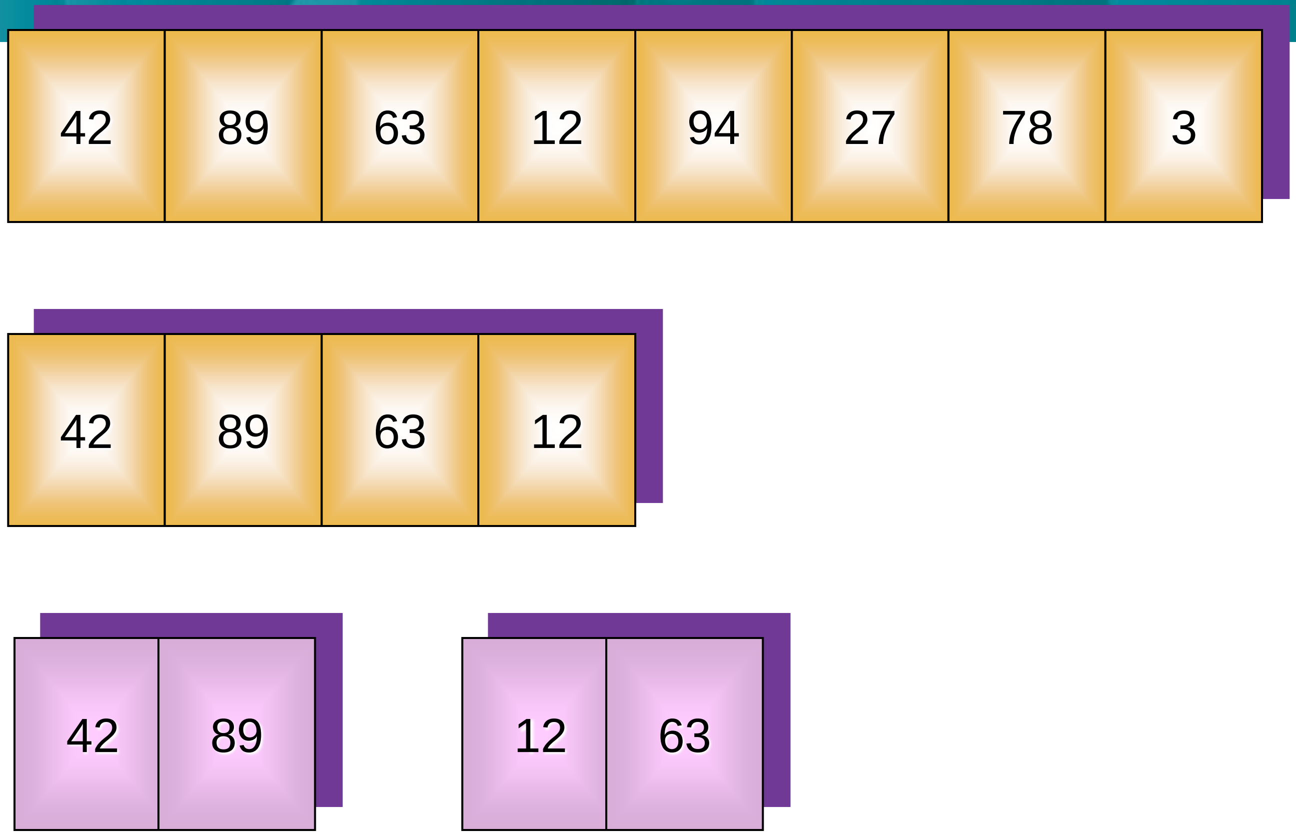

42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63
12

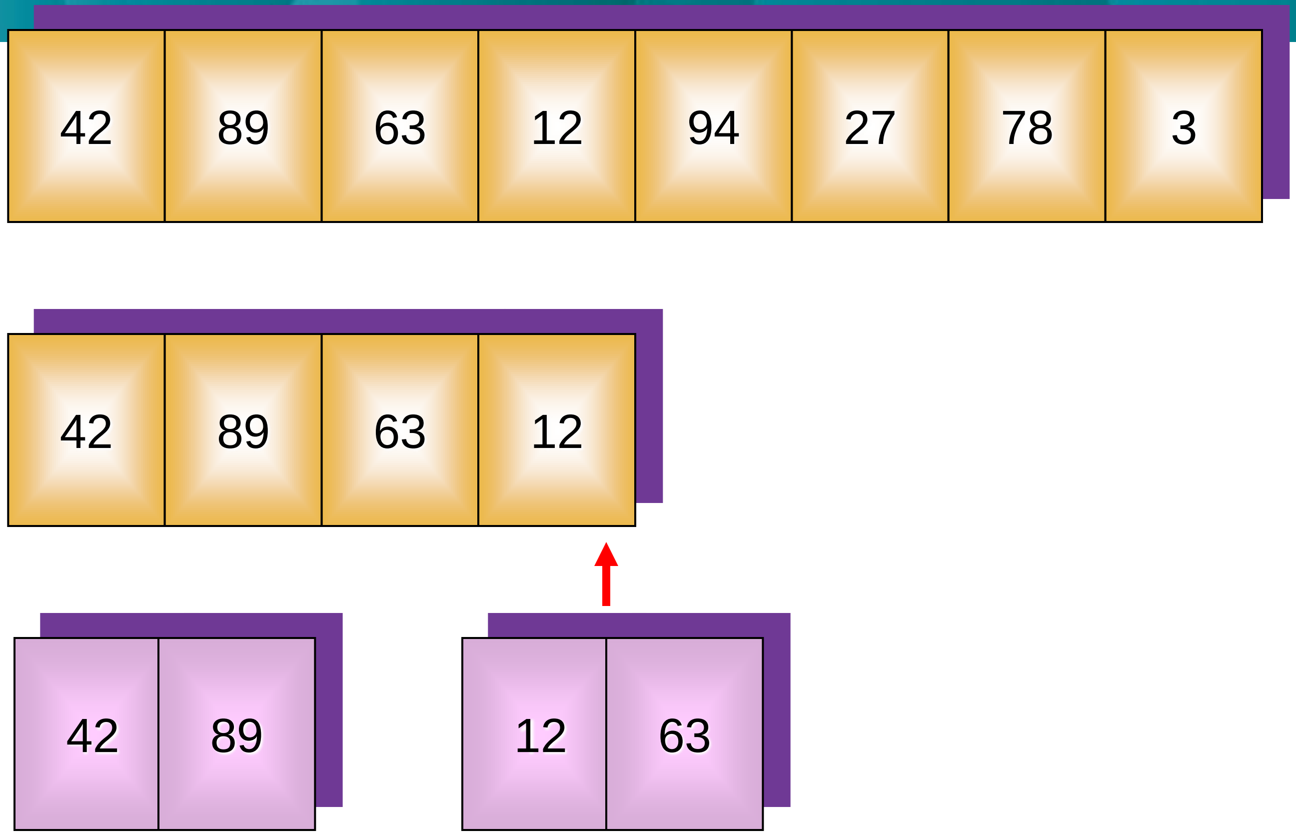


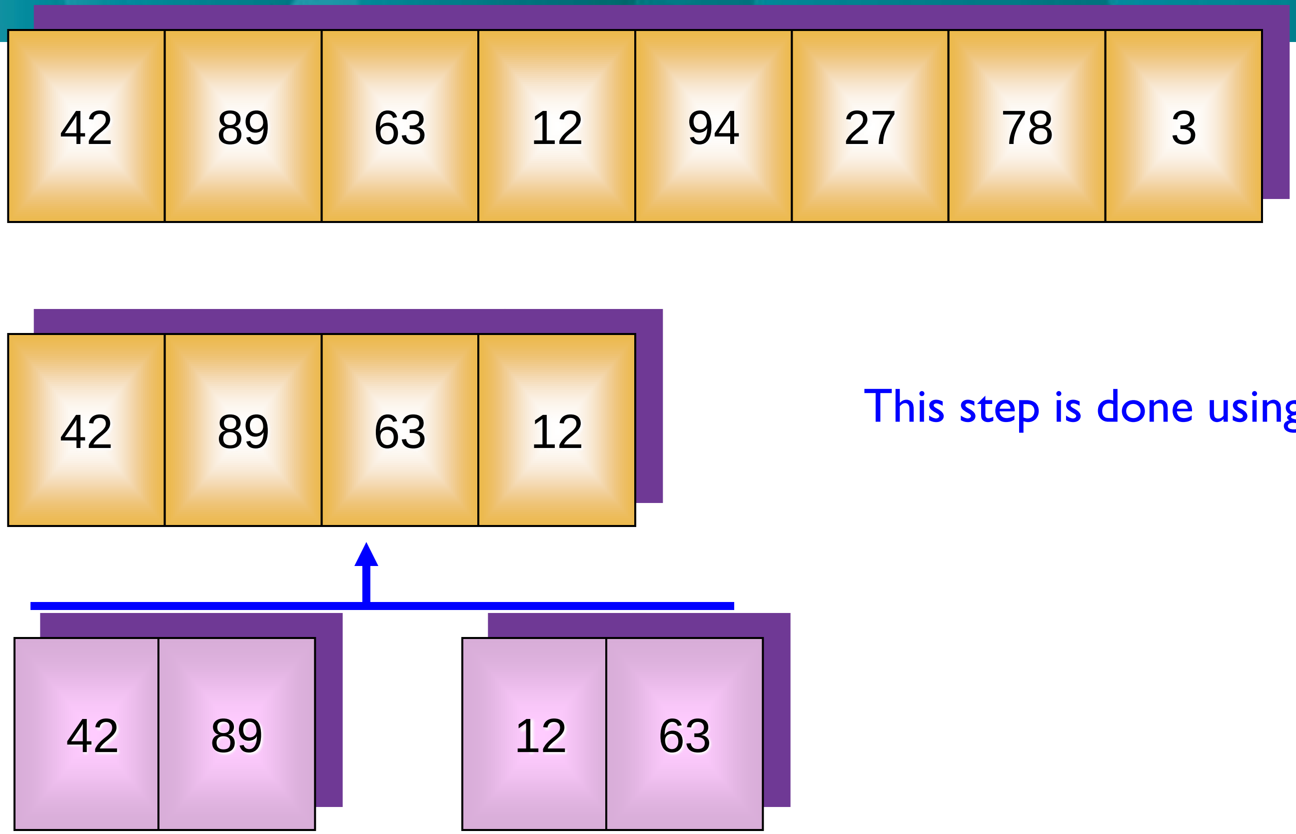
42
89
63
12
94
27
78
3
42
89
63
12
42
89
63
12
63
12


42
89
63
12
94
27
78
3
42
89
63
12
42
89
63
12
63
12


42 89 63 12 94 27 78 3
42 89 63 12
42 89
63 12
63
12


42 89 63 12 94 27 78 3
42 89 63 12
42 89
12 63


42 89 63 12 94 27 78 3
42 89 63 12
42 89
12 63

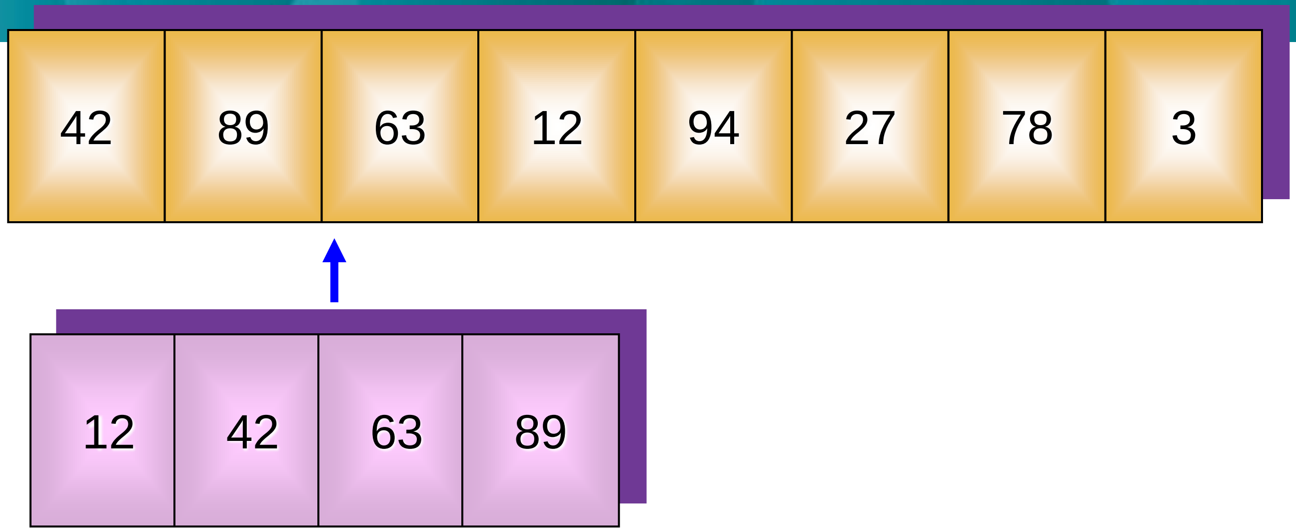


This step is done using MERGE described later

42	89	63	12	94	27	78	3

12	42	63	89

This step is done using MERGE described later


42 89 63 12 94 27 78 3
12 42 63 89

Global A[low..high] is a global array containing high-low+1 values the element to be sorted.

```
procedure MERGESORT(low, high:integer);
var mid : integer;
begin
    if (low < high) then
    begin
        mid ← (low+high) div 2;         //find the split position//
        MERGESORT(low,mid);             //sort first subset//
        MERGESORT(mid+1,high);          //sort another subset//
        MERGE(low,mid,high);            //combine the result//
    end
end.
```

Global A[low..high] is a global array containing two sorted subsets in A[low..mid] and in A[mid+1..high]

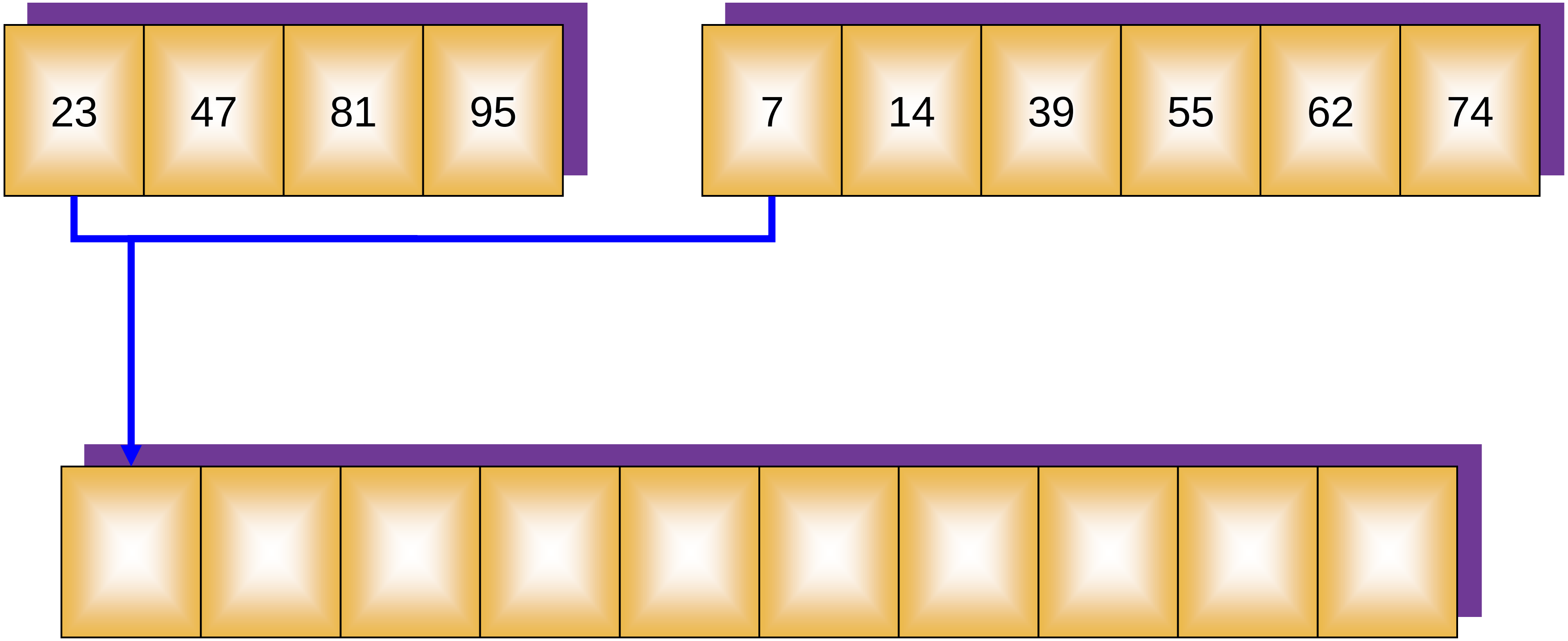
procedure MERGE(low, mid, high:integer);

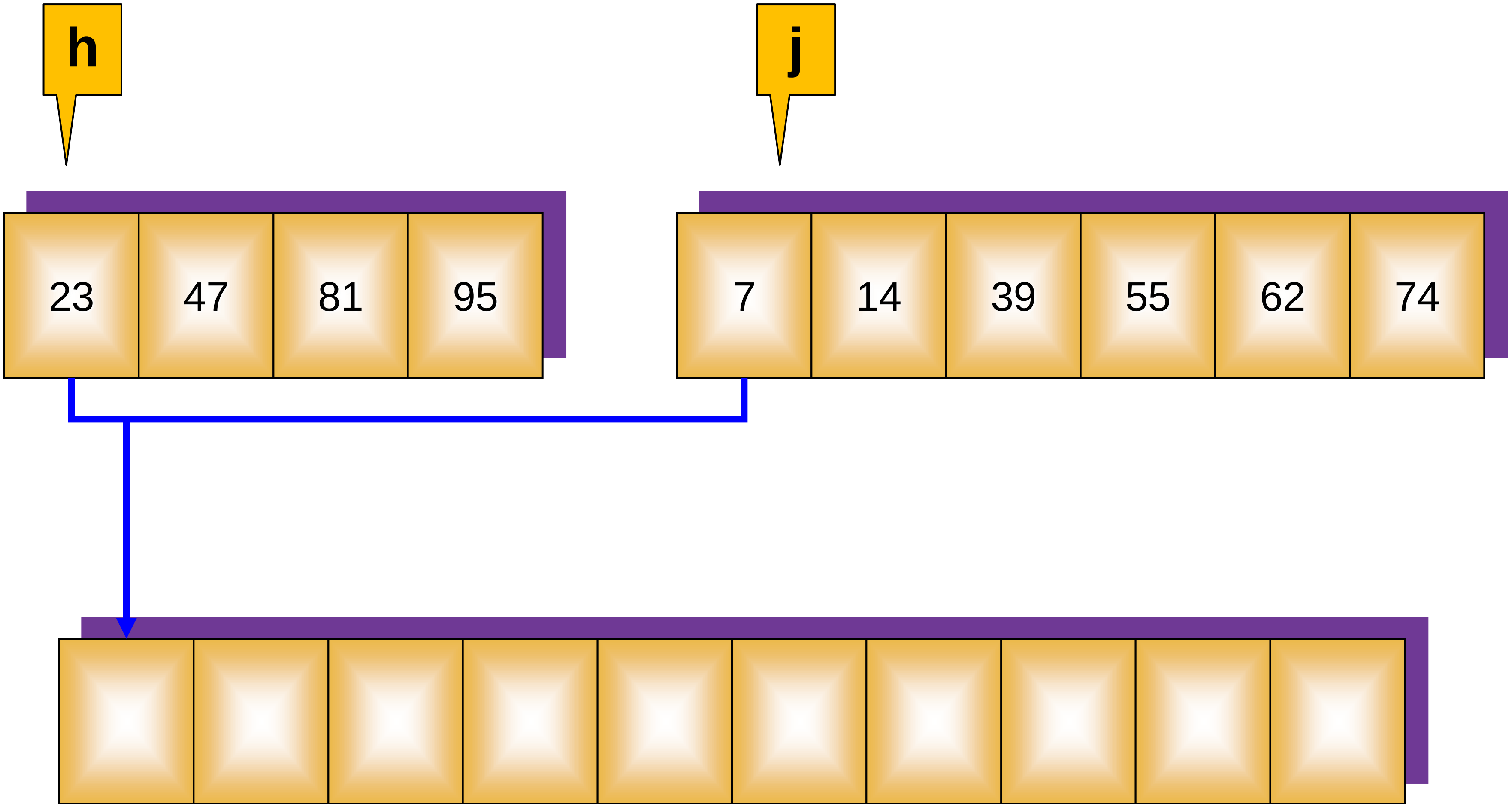
```
var       B: array[low..high] of items;
          h, i, j, k:integer;
begin
    h← low; j← mid+1; i← low;
    while ((h ≤ mid) and (j ≤ high)) do
    begin
        if ( A[h] ≤ A[j] ) then
        begin
            B[i] ← A[h] ; h ← h+1;
        end
```

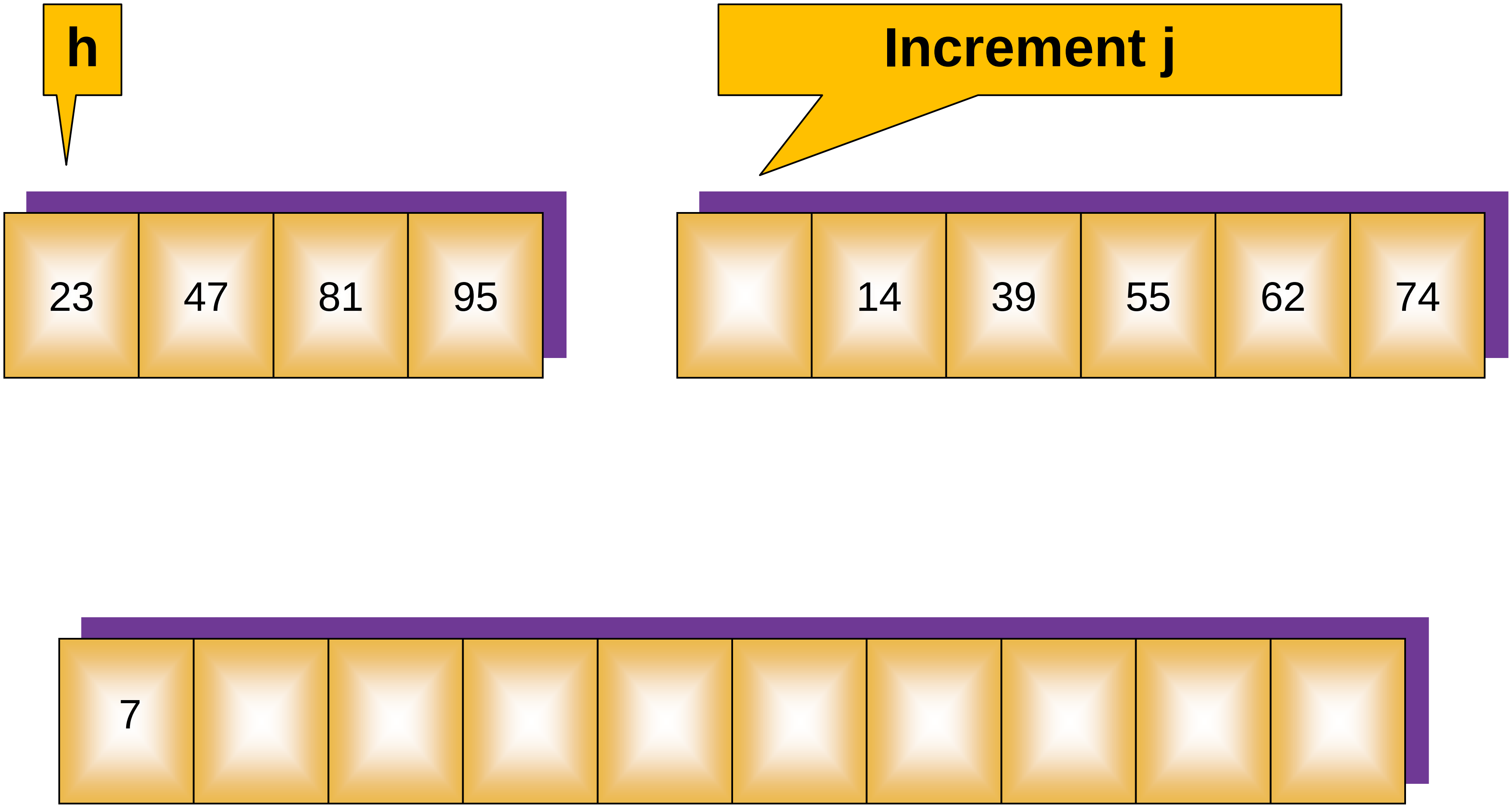
```
        else
        begin
                B[i] ← A[j] ; j ← j+1;
        end
end;
if ( h > mid ) then                    //remaining elements//
for k ← j to high do
begin
        B[i] ← A[k]; i ← i+1;
end
else
```

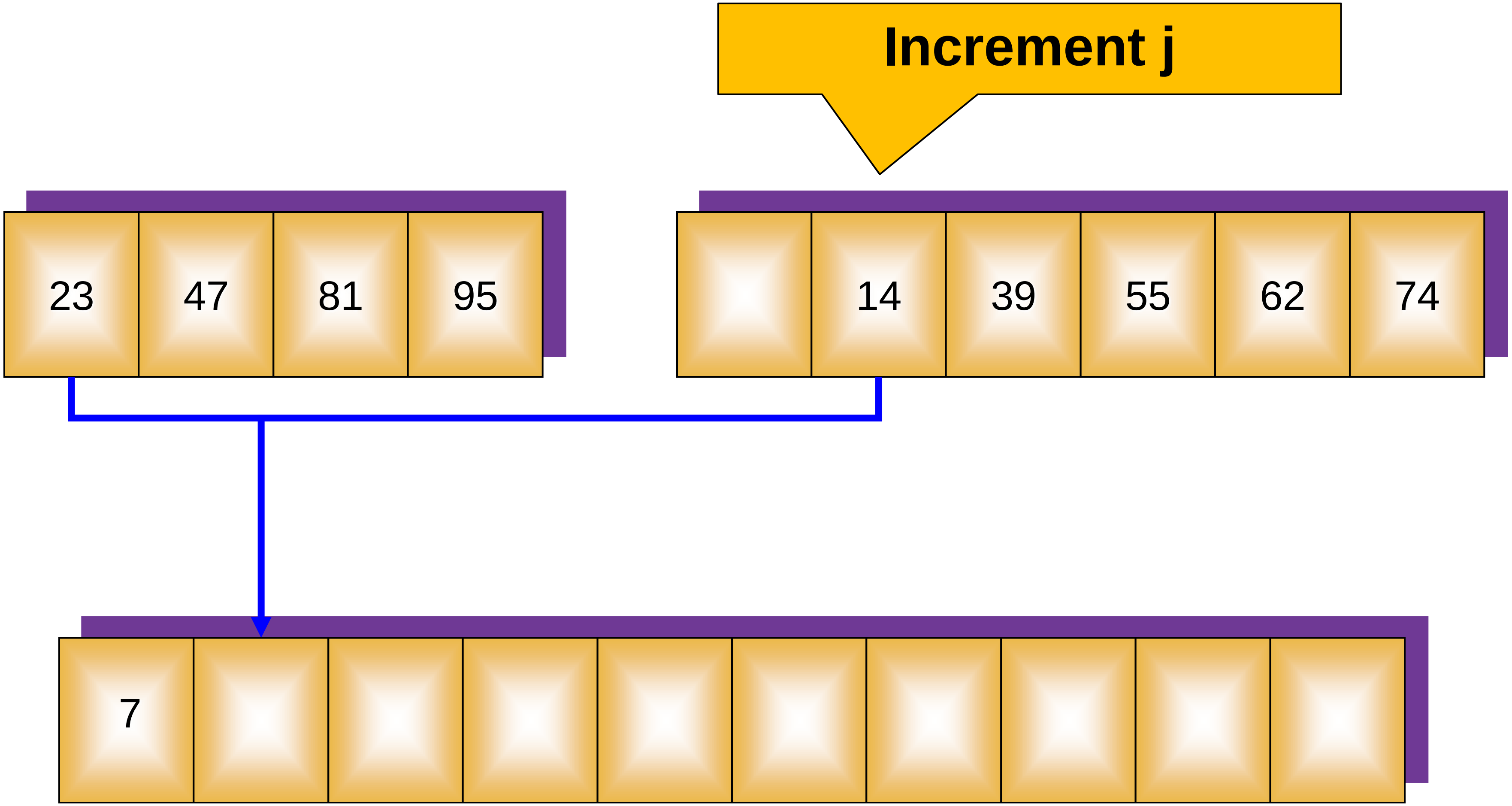
```
        for k ←h to mid do
        begin
                B[i] ← A[k]; i ← i+1;
        end
        for k ← low to high do
                A[k] ← B[k];
end.
```

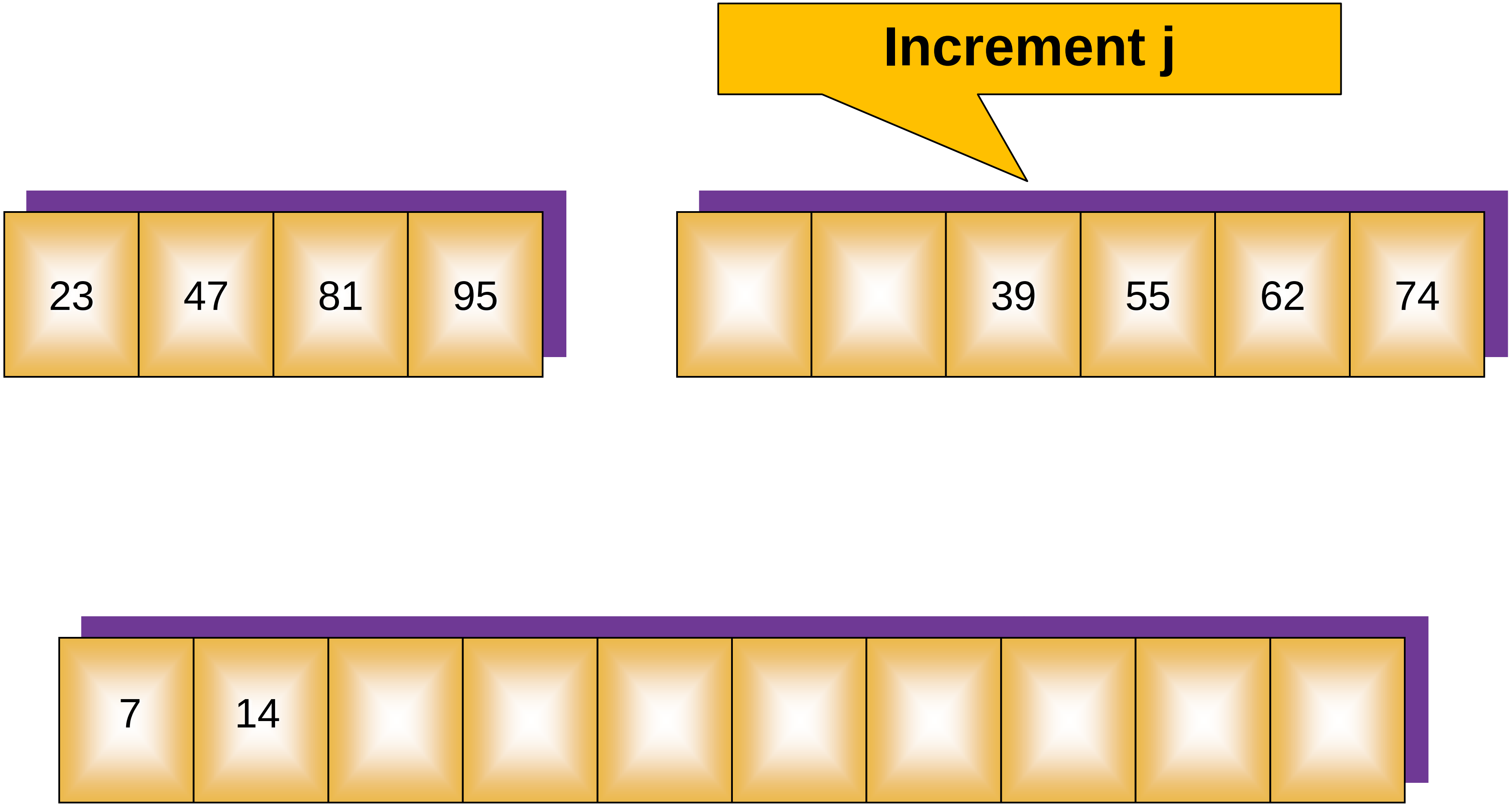
Merging two sorted arrays

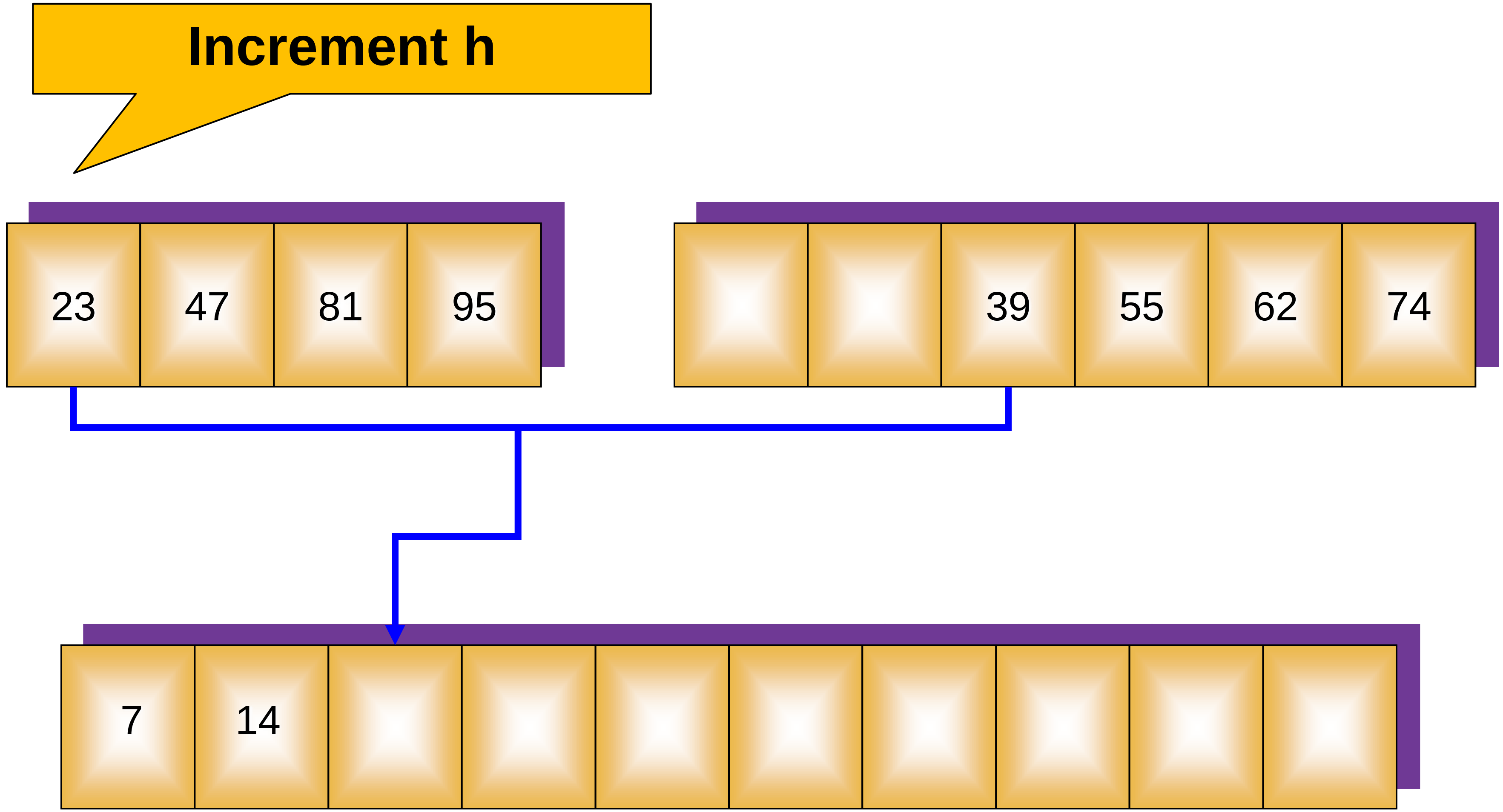


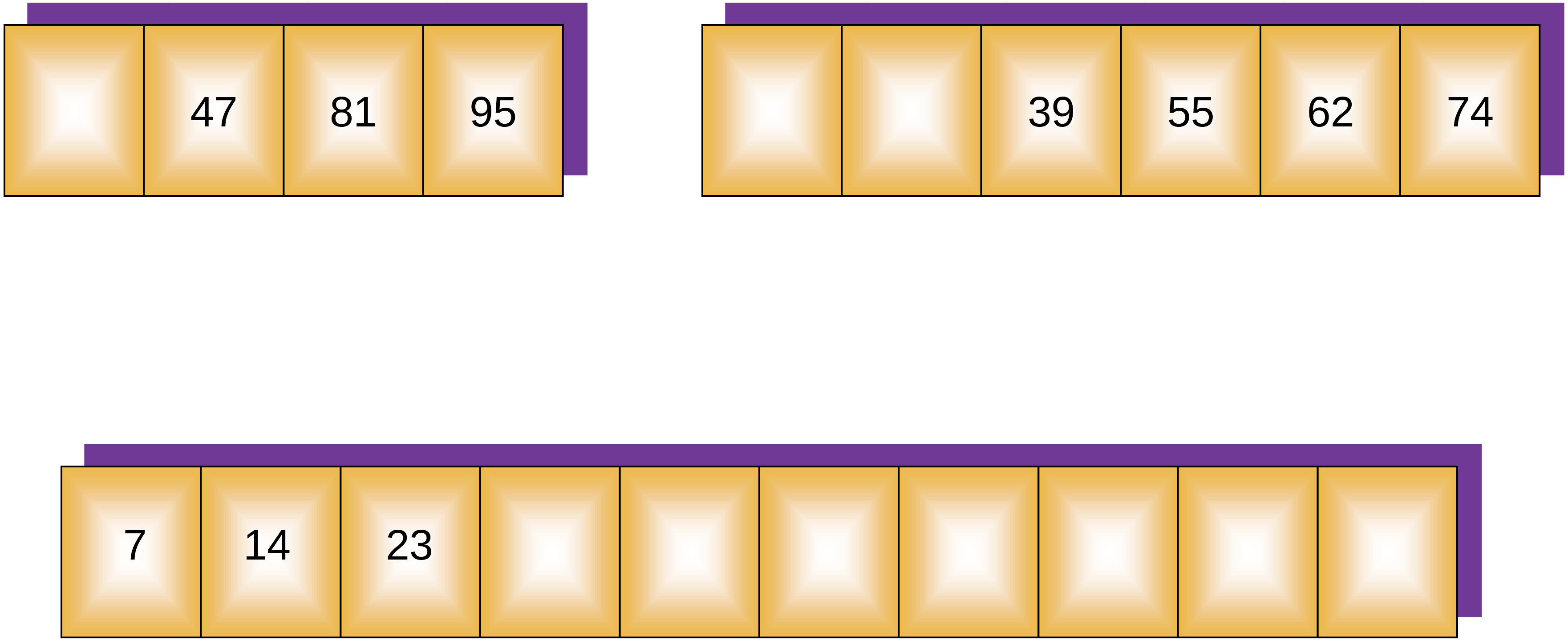

h
j
23
47
81
95
7
14
39
55
62
74

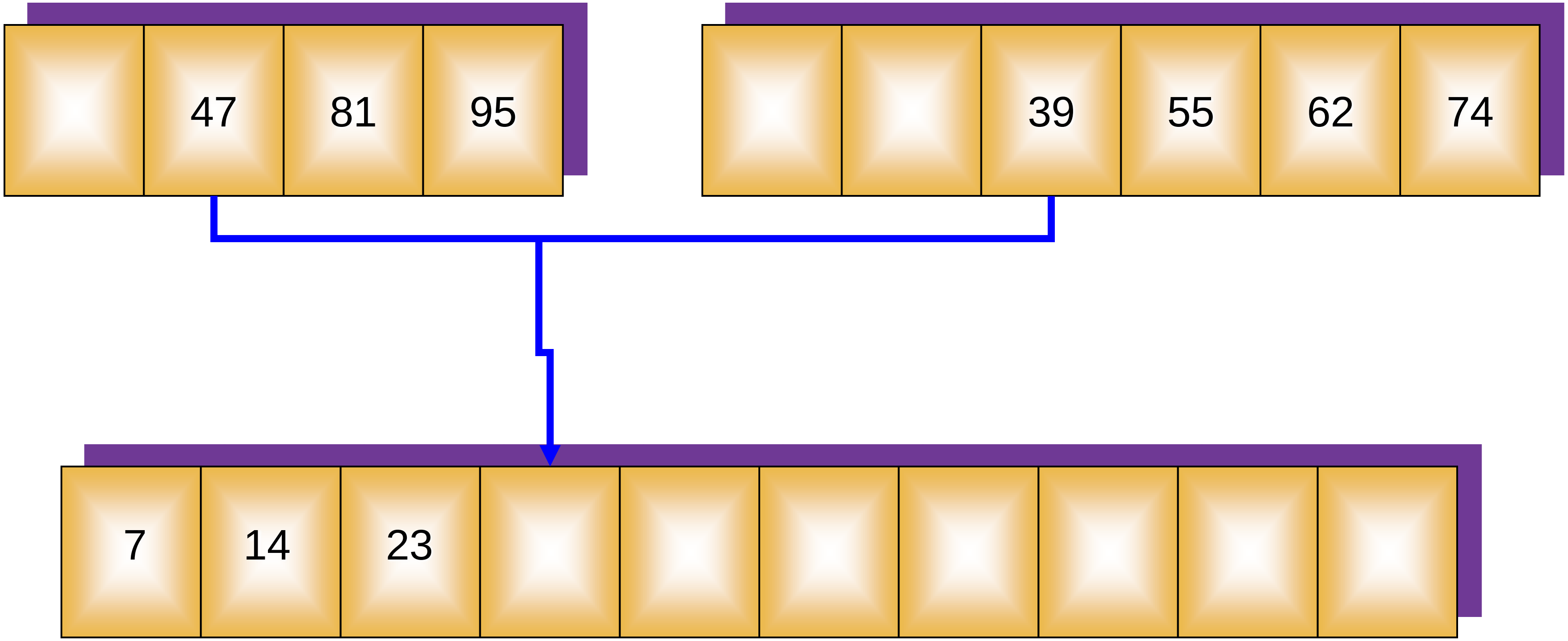

h
Increment j
23
47
81
95
14
39
55
62
74
7

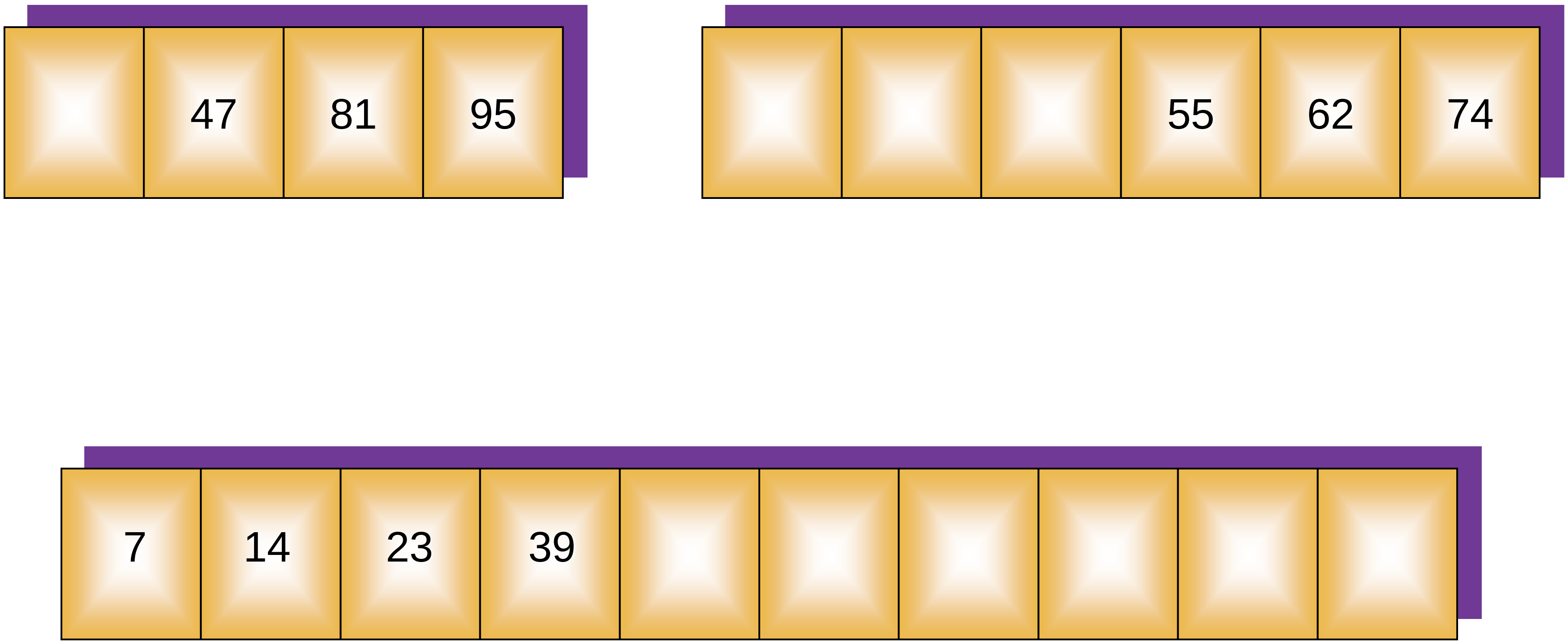

Increment j
23
47
81
95
14
39
55
62
74
7

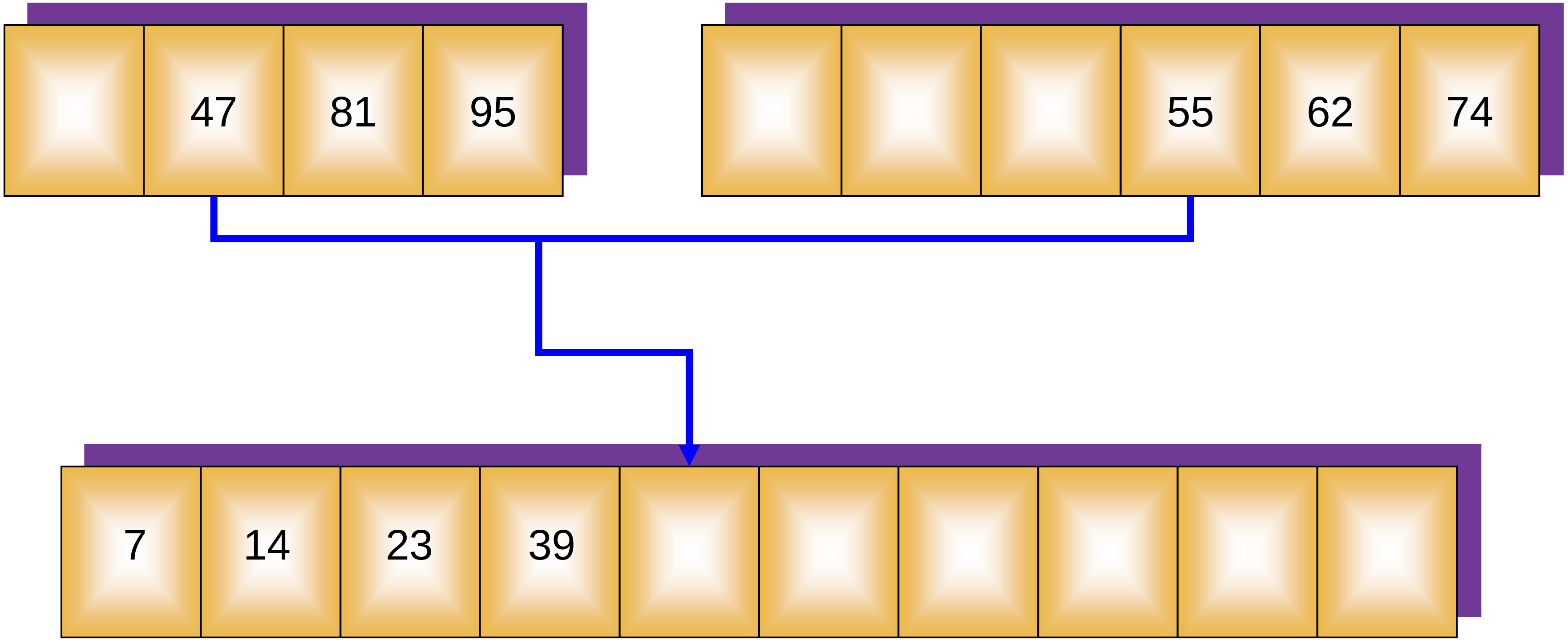

Increment j
23
47
81
95
39
55
62
74
7
14

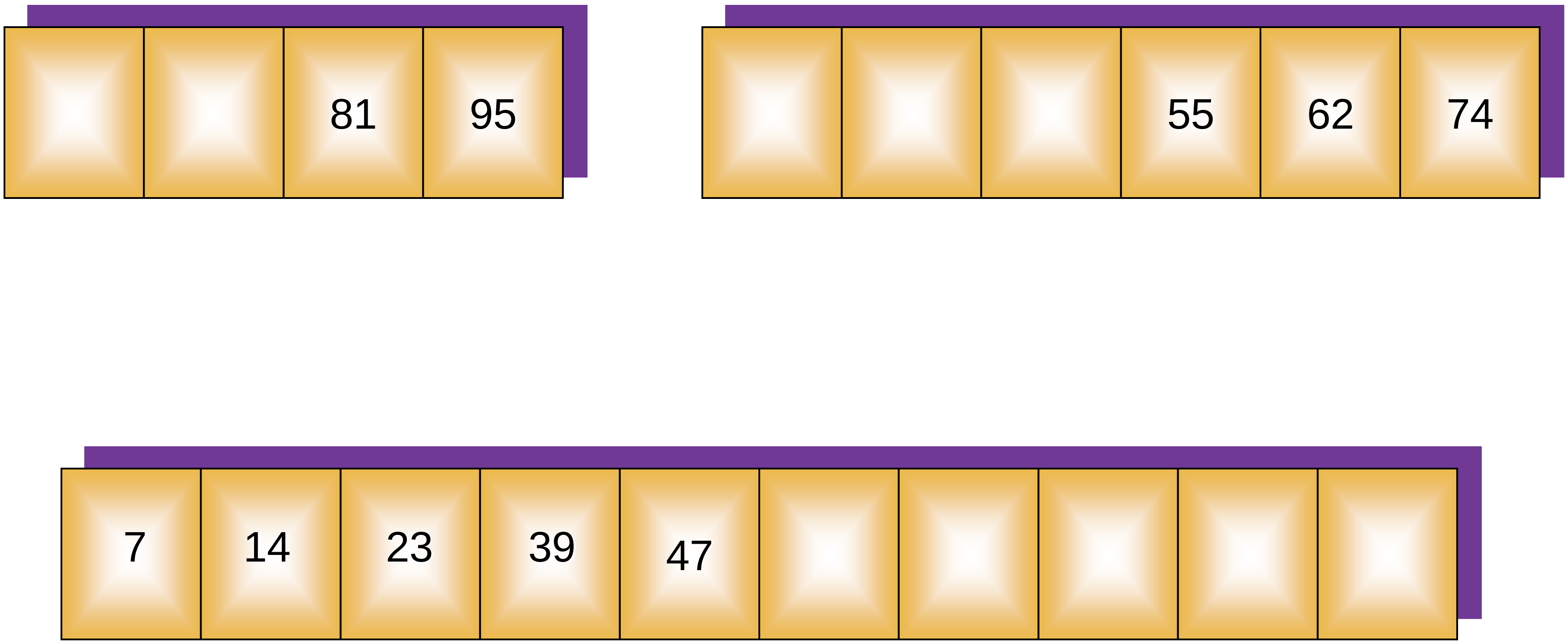

Increment h
23
47
81
95
39
55
62
74
7
14

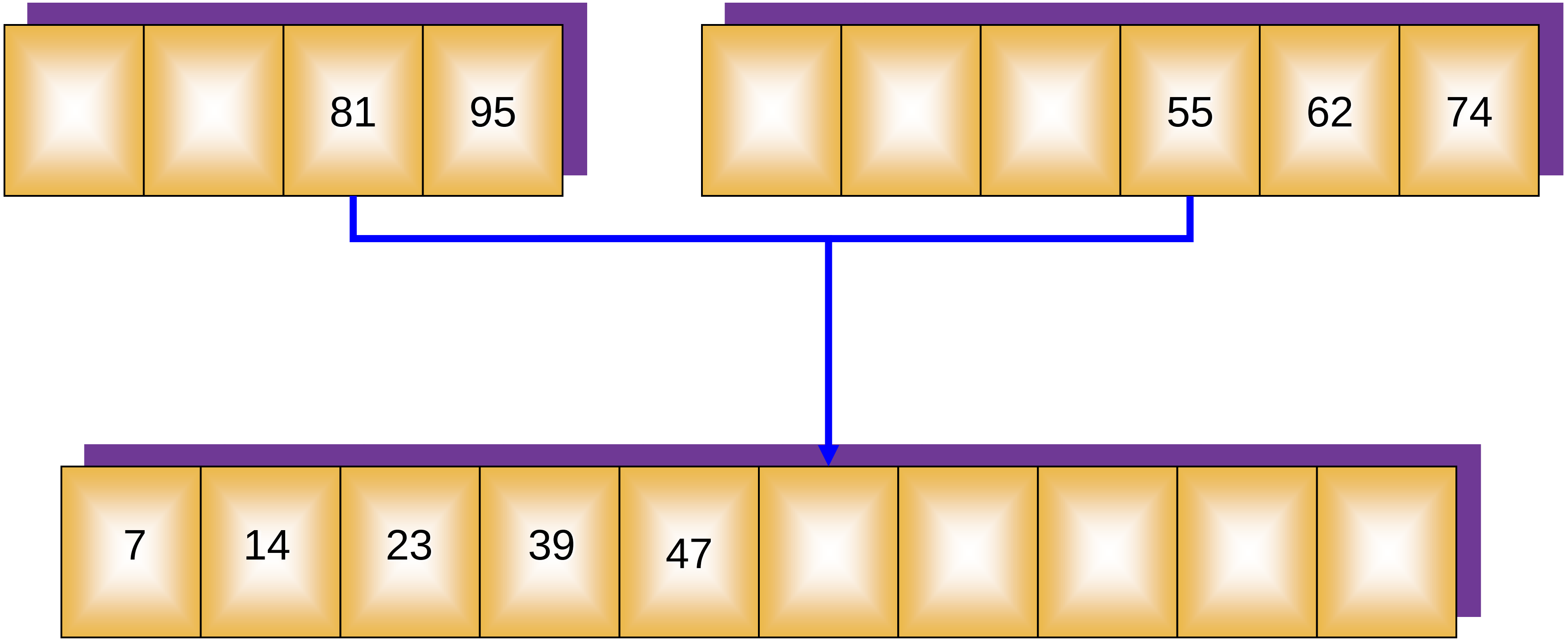

47
81
95
39
55
62
74
7
14
23

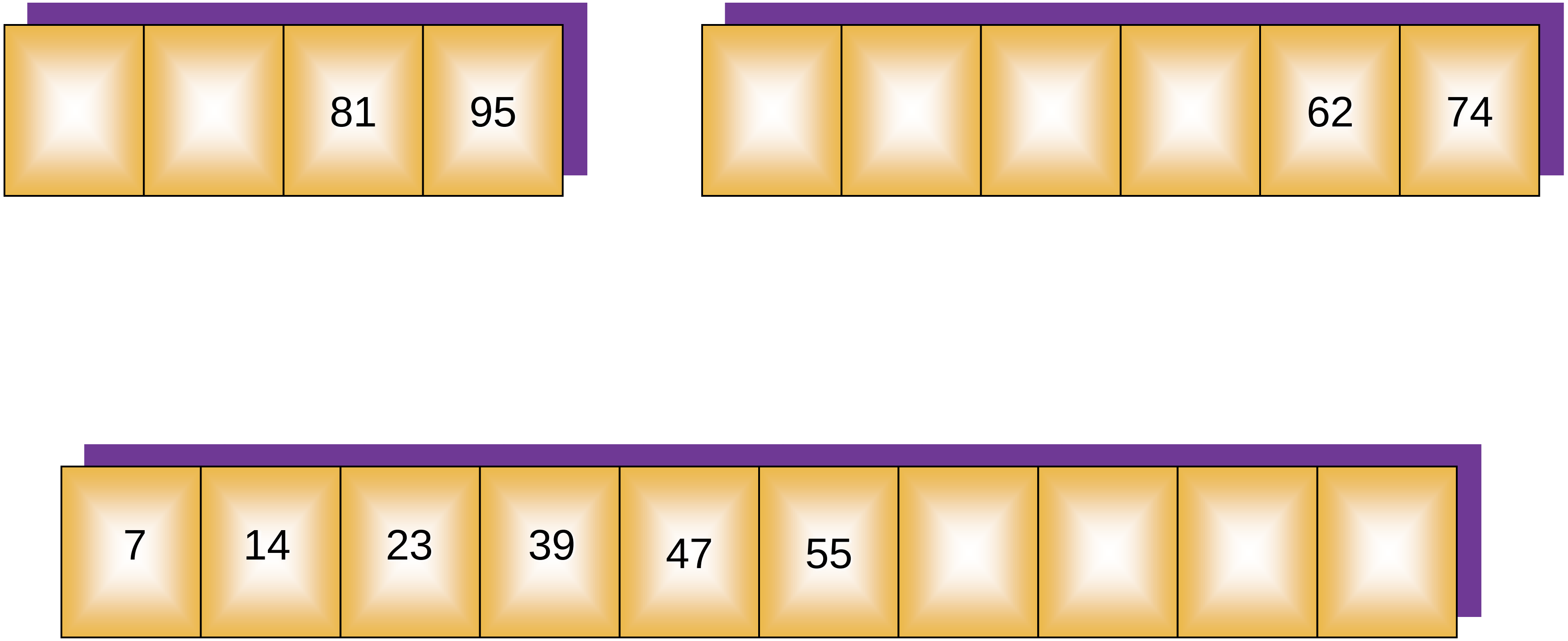

47
81
95
39
55
62
74
7
14
23

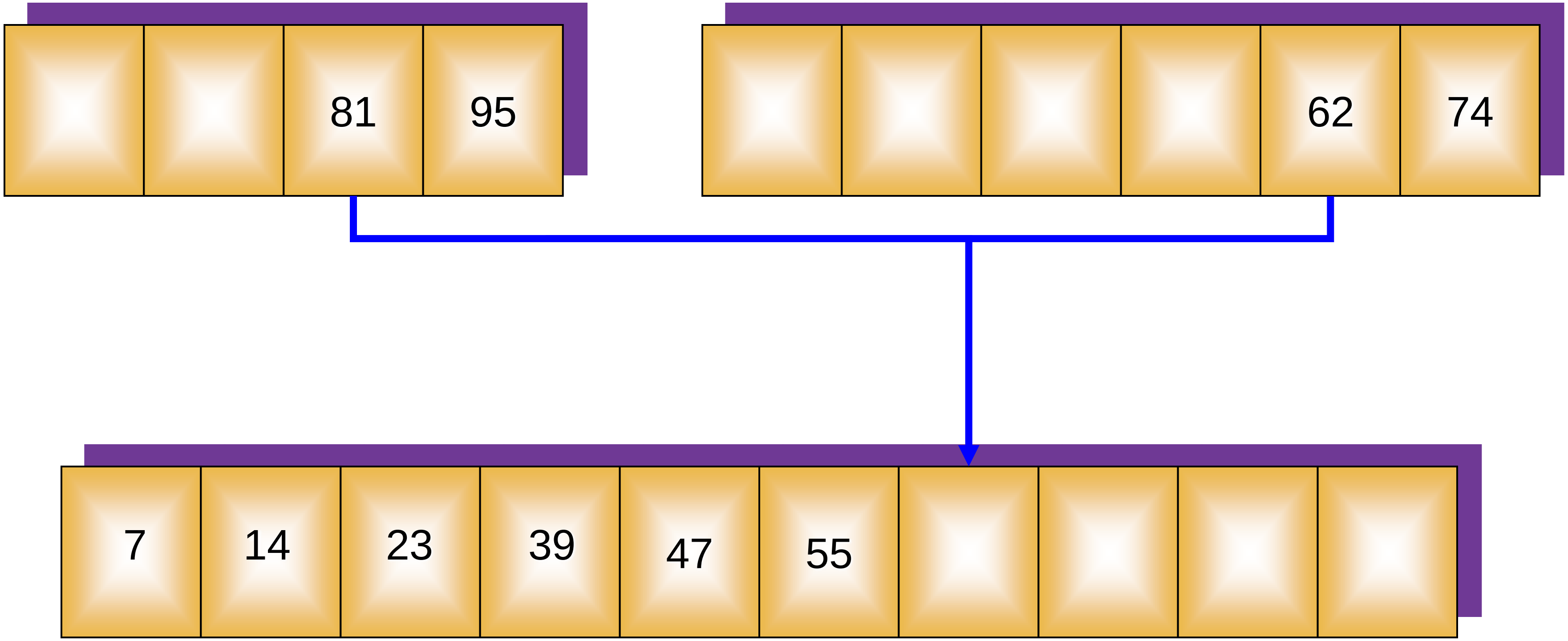

47
81
95
55
62
74
7
14
23
39

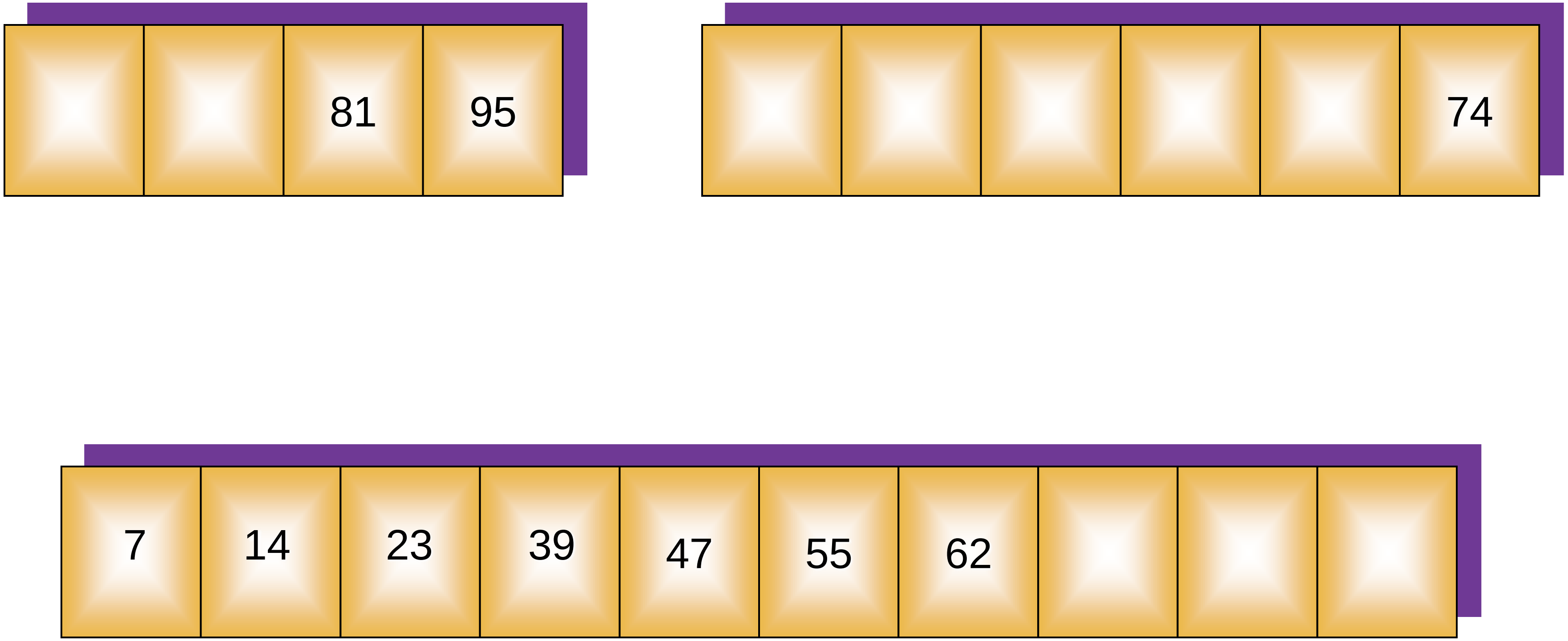

47
81
95
55
62
74
7
14
23
39

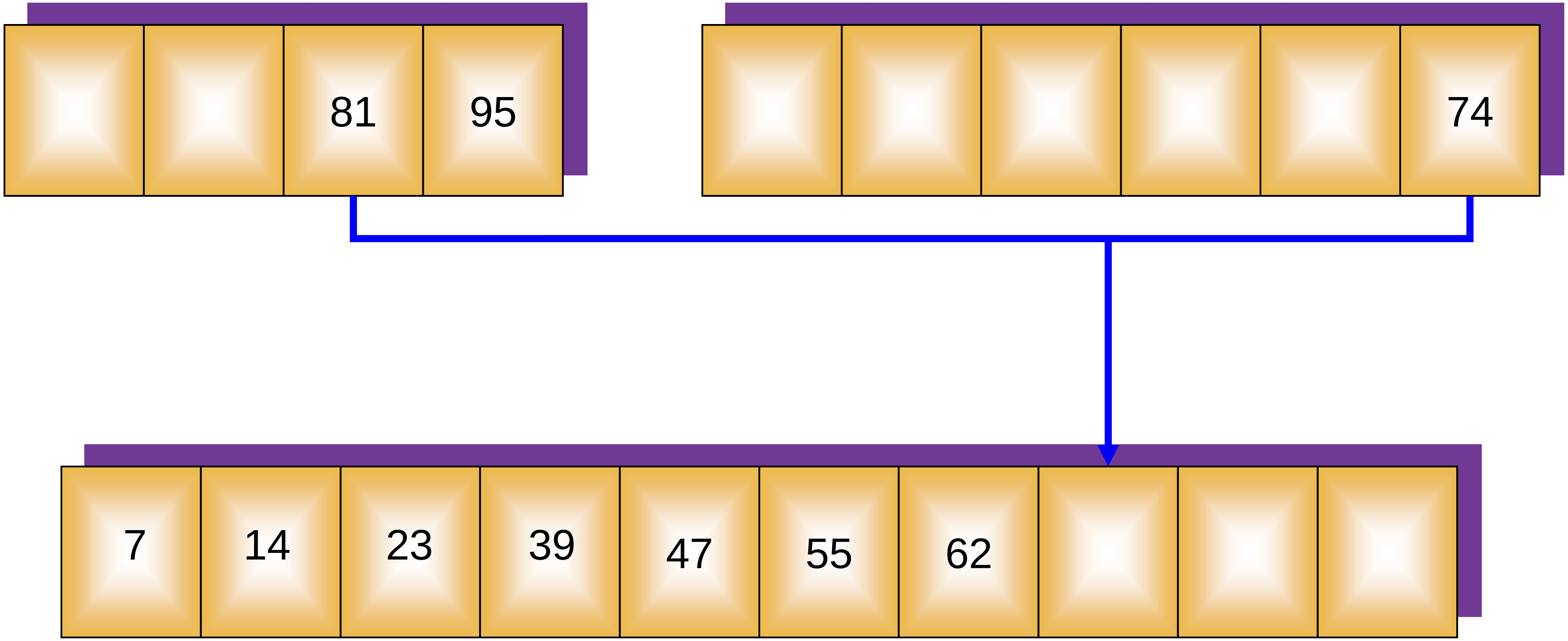

81
95
55
62
74
7
14
23
39
47

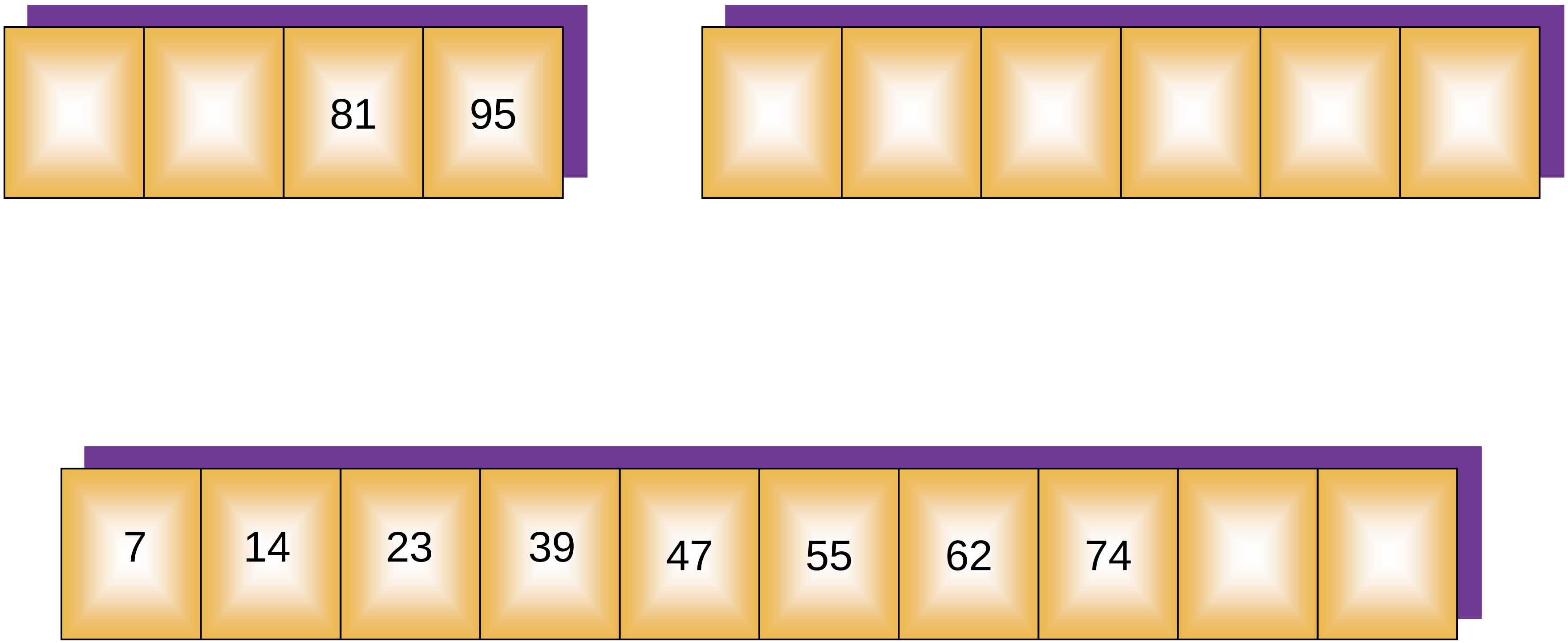

81
95
55
62
74
7
14
23
39
47


81
95
62
74
7
14
23
39
47
55


81
95
62
74
7
14
23
39
47
55


81
95
74
7
14
23
39
47
55
62


81
95
74
7
14
23
39
47
55
62

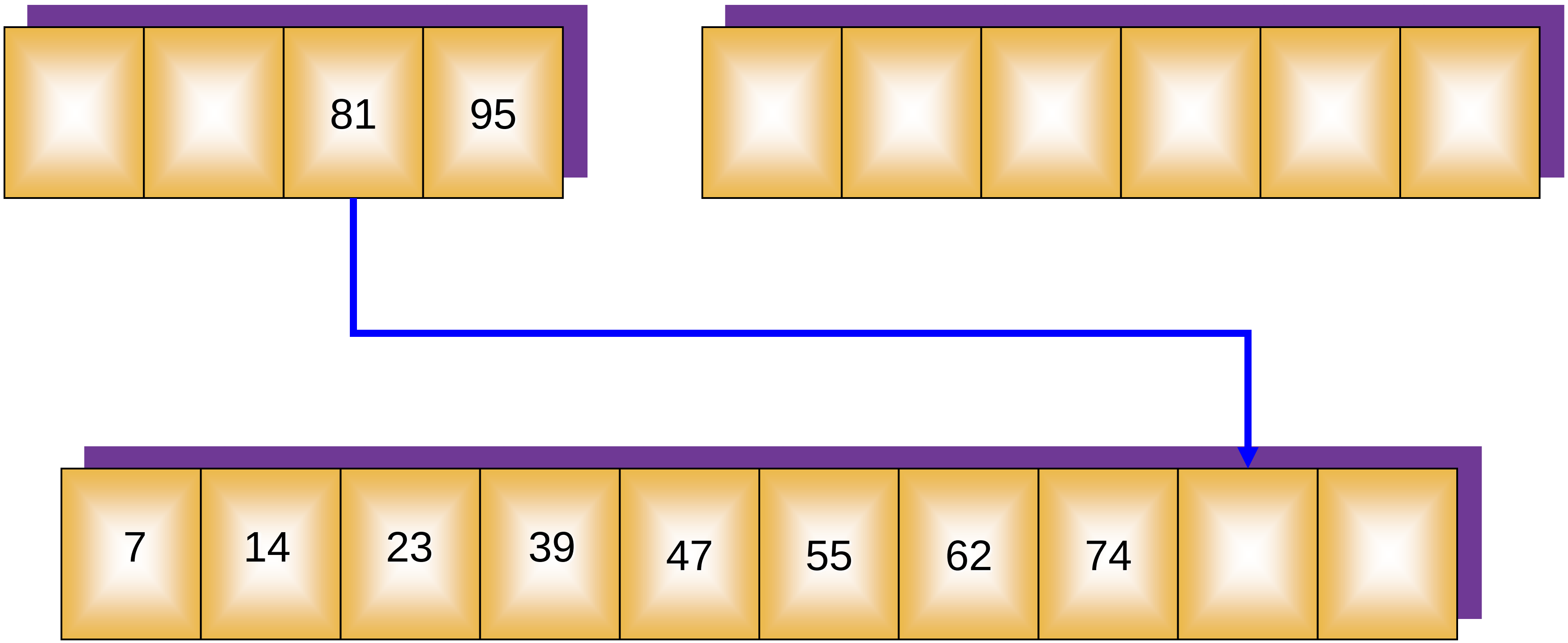

81
95
7
14
23
39
47
55
62
74

for k ←h to mid do

begin

        B[i] ← A[k]; i ← i+1;

end

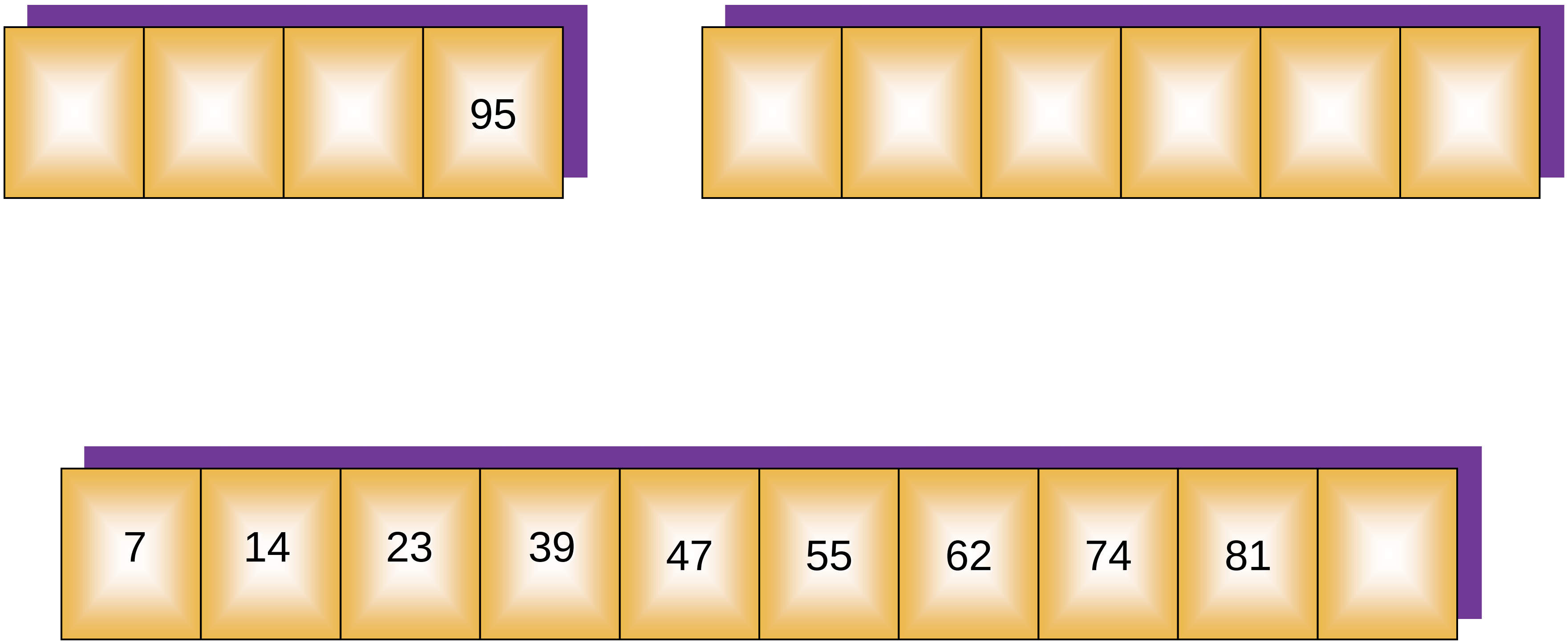


for k ←h to mid do

begin

    B[i] ← A[k]; i ← i+1;

end

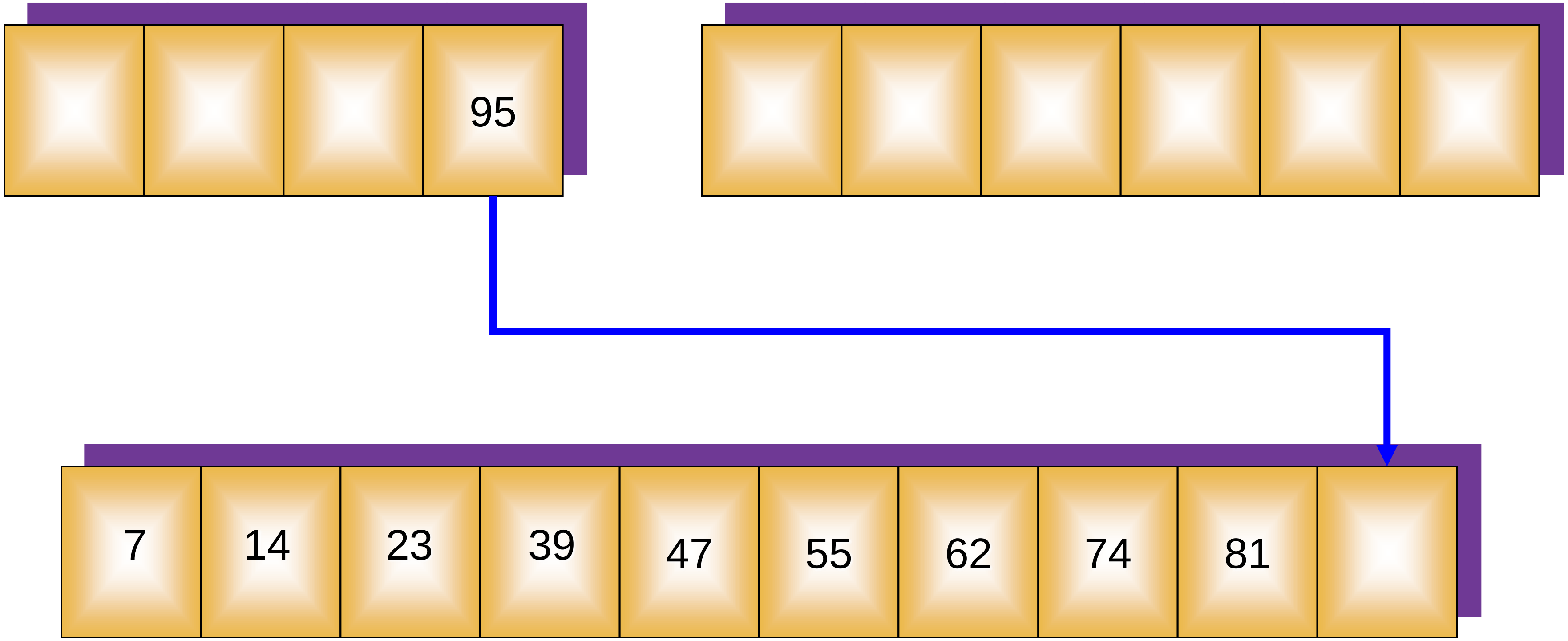


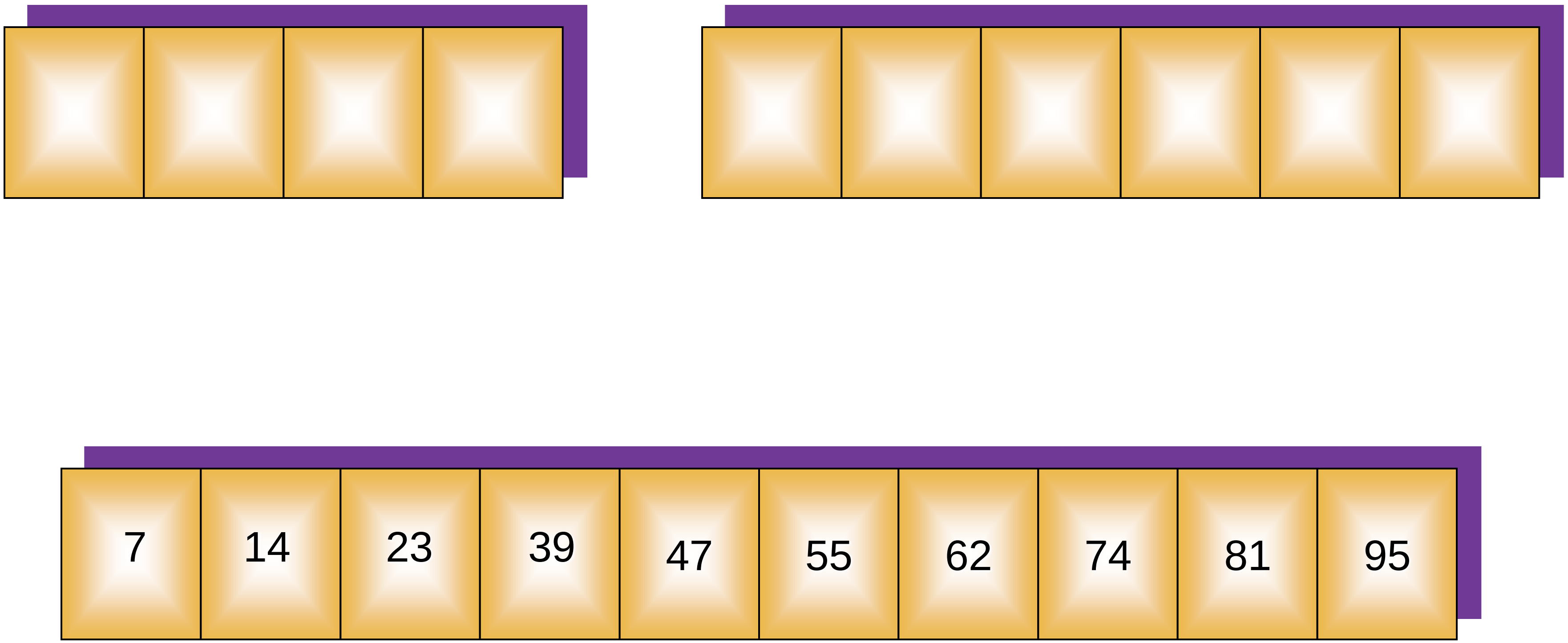
for k ←h to mid do

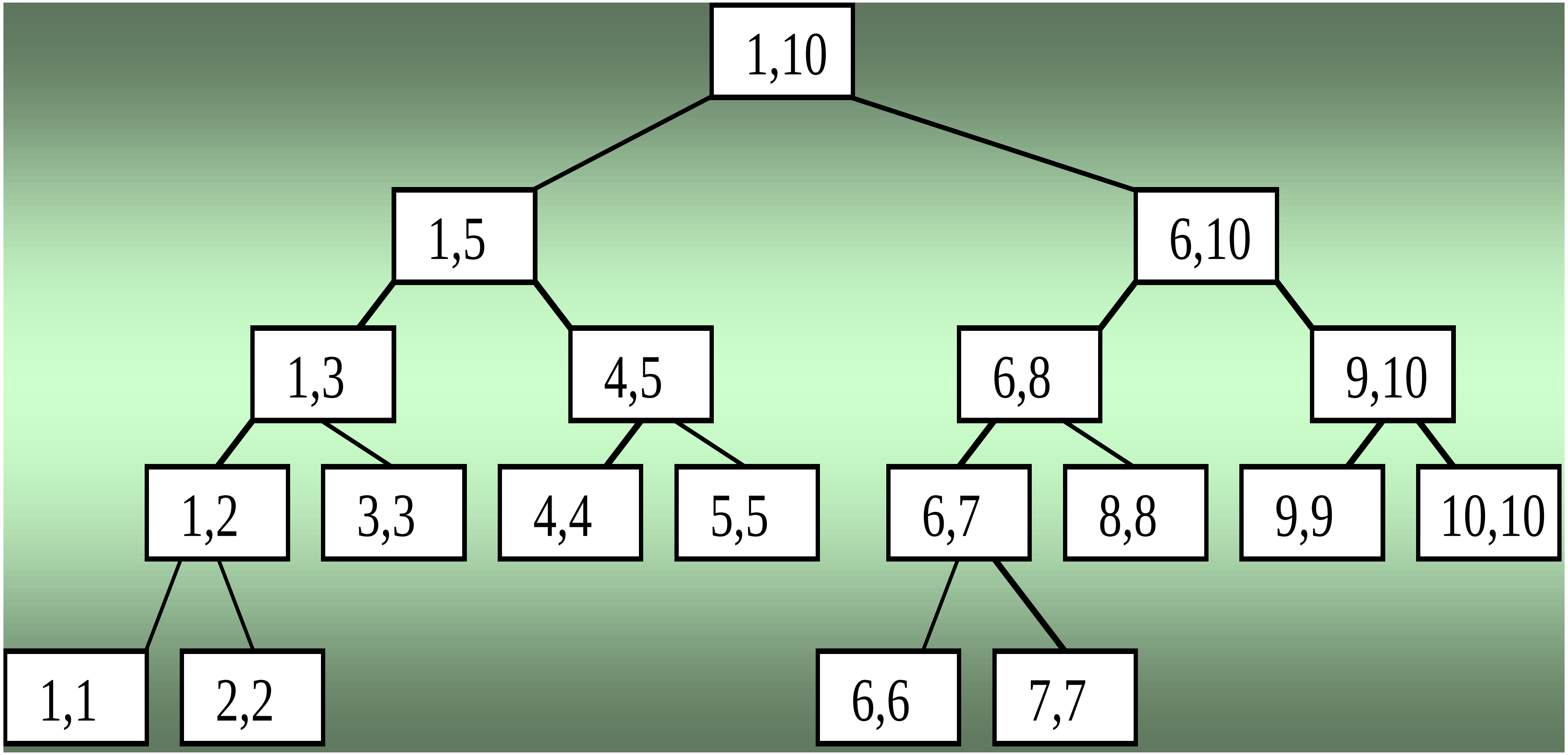
begin

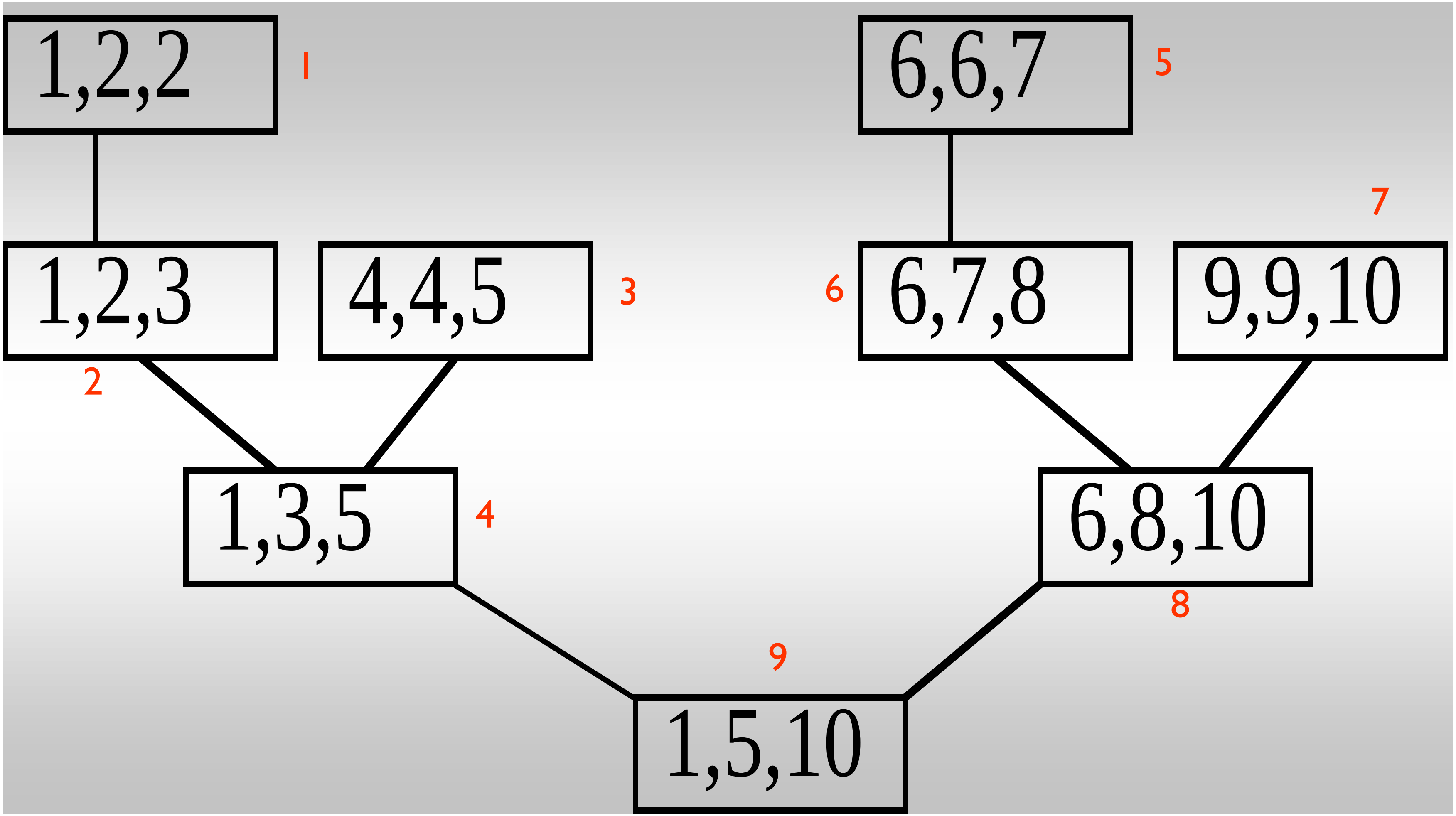
    B[i] ← A[k]; i ← i+1;

end




7
14
23
39
47
55
62
74
81
95


1,10
1,5
6,10
1,3
4,5
6,8
9,10
1,2
3,3
4,4
5,5
6,7
8,8
9,9
10,10
1,1
2,2
6,6
7,7


1,2,2
1
6,6,7
5
7
1,2,3
4,4,5
3
6
6,7,8
9,9,10
2
1,3,5
4
6,8,10
8
9
1,5,10

If the time for the merging operation is proportional to n then computing time for "MERGESORT" is given by recurrence relation-

$$T(n) = \begin{cases} a, \quad n = 1, \quad a \quad is \quad const. \\ 2T(n/2) + cn, \quad n > 1, \quad c \quad is \quad const. \end{cases}$$

when n is power of 2 then we can solve this equation by successive substitutions, namely-

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

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

$$T(n) = 8T(n/8) + 3.c.n$$

.

.

$$= 2^k T(1) + k.c.n$$

$$= n.a + \log n.c.n$$

$$= a.n + c.n.\log n$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$ therefore

$$T(n) = O(n \log n)$$