Structuri de Date și Algoritmi
Colecții și mulțimi

Mihai Nan

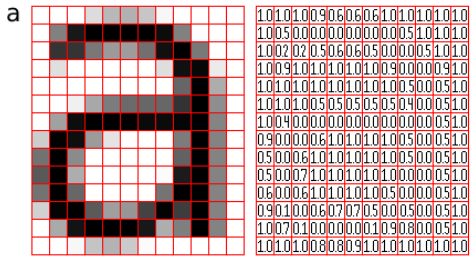
Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București

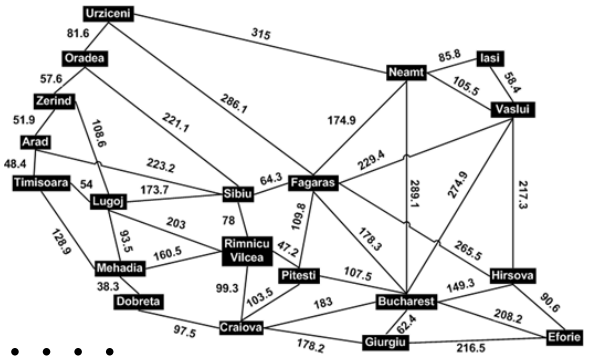
Anul Universitar 2022–2023

Conținutul cursului

Introducere în Structuri de date

> *"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*
>
> **Linus Torvalds, 2006**

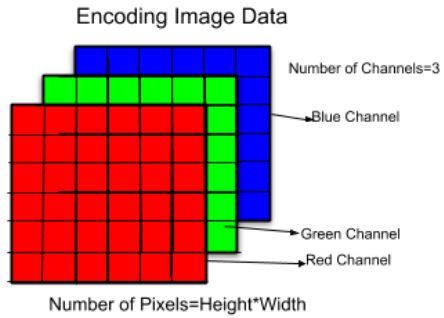




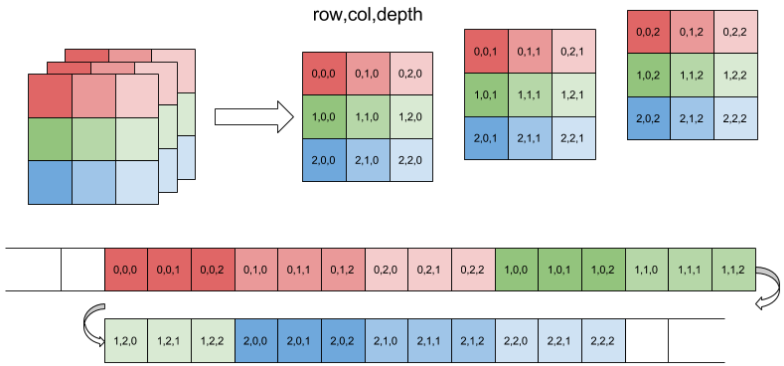
00101010010101010100101010000010010010100....

Introducere în Structuri de date

- În cadrul programelor, trebuie să găsim o modalitate de reprezentare a datelor digitale care să asigure:
  1. **Encodarea datelor**



  2. **Stocarea** – Stocat în mod ordonat în memorie/disc



  3. **Accesarea** – determinarea unor valori, ștergerea / inserarea
  4. **Procesarea** – Aplicarea unor algoritmi

Introducere în Struncturi de date

Structură de date

O structură de date este o colecție de elemente ce are asociată o modalitate de organizare a datelor în memorie ce asigură accesul eficient la date.

Exemple

variabile, vectori, matrice, liste, arbori, grafuri, tabele de dispersie

```
1  int vector[10];
2  double matrice[100][100];
3  struct punct {int x, y;} p;
4  struct node {
5      int element;
6      struct node *urm;
7  };
```

```
8  struct pereche {
9      char *cuvant;
10     int aparitii;
11 };
```

Tip abstract de date

Un tip de date precizează o mulțime **D** finită și ordonată de valori (constantele tipului) și o mulțime de operații aplicate valorilor, de forma $\mathbf{F} : \mathbf{D} \times \mathbf{D} \to \mathbf{D}$ pentru operații binare sau $\mathbf{F} : \mathbf{D} \to \mathbf{D}$ pentru operații unare.

Exemplu: Pentru tipul standard `int` avem următoarele:

- **D** – submulțimea numerelor întregi cuprinse între -32768 și 32767;
- $\mathbf{F} = \{+, -, *, /, \%\}$

Un **Tip Abstract de Date** este o specificare a unui set de date de un anumit tip, împreună cu un set de operații care pot fi executate cu aceste date. Cu alte cuvinte, un **TAD** este o *entitate matematică abstractă* definită printr-un tuplu $(\mathbf{D}, \mathbf{F}, \mathbf{A})$ unde:

- **D** – o mulțime de domenii (sau tipuri);
- **F** – o mulțime de funcții;
- **A** – o mulțime de axiome care precizează proprietățile funcțiilor și care trebuie respectate la implementare.

Tip abstract de date

Mulțimea numerelor naturale

- **D** $= \{\{0, 1, 2, 3, \dots\}, \{true, false\}\}$
- **F** $= \{zero, isZero, succ, add\}$
- **A** – mulțimea de axiome

$$isZero(0) = true$$
$$isZero(succ(x)) = false$$

$$add(0, x) = x$$
$$add(succ(x), y) = succ(add(x, y))$$

Observație

Un **TAD** permite abstractizarea operațiilor fără a ține cont de detalii (de implementare) cât și încapsularea datelor.

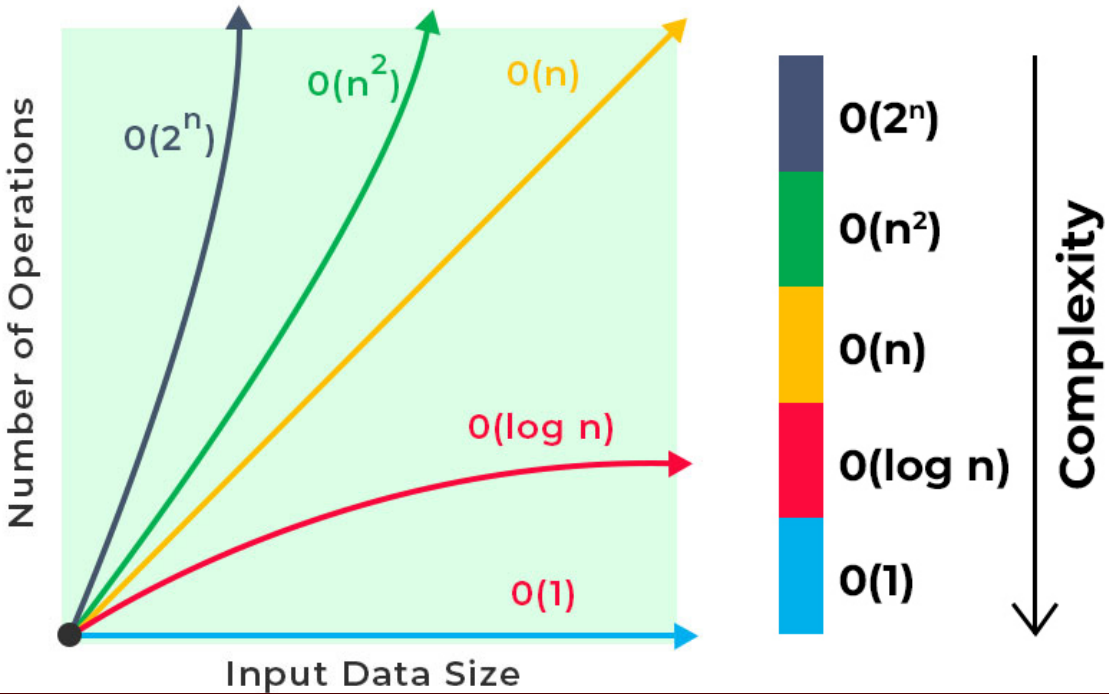
Noțiuni elementare de Analiza Algoritmilor

- **Dimensiunea unei probleme** desemnează o măsură compusă a acelor date care influențează major performanțele dinamice ale algoritmului.
- Fie $g : \mathbb{N} \to \mathbb{R}_+$ o funcție reprezentativă de complexitate. Parametrul unei asemenea funcții corespunde dimensiunii problemei, iar valoarea reprezintă consumul de resurse.
- **Eficiența** unui algoritm caracterizează resursele consumate de algoritm la execuție: timp de execuție (eficiența timpului) și memorie consumată (eficiența spațiului).
- **Analiza algoritmică** (numită și analiză asimptotică) caracterizează comportarea la execuție a algoritmului independent de platformă, compilator sau limbaj de programare.
- Din acest punct de vedere, exprimarea numărului de operații elementare în funcție de dimensiunea problemei (volumul datelor de intrare) caracterizează cel mai bine complexitatea algoritmului.

Noțiuni elementare de Analiza Algoritmilor

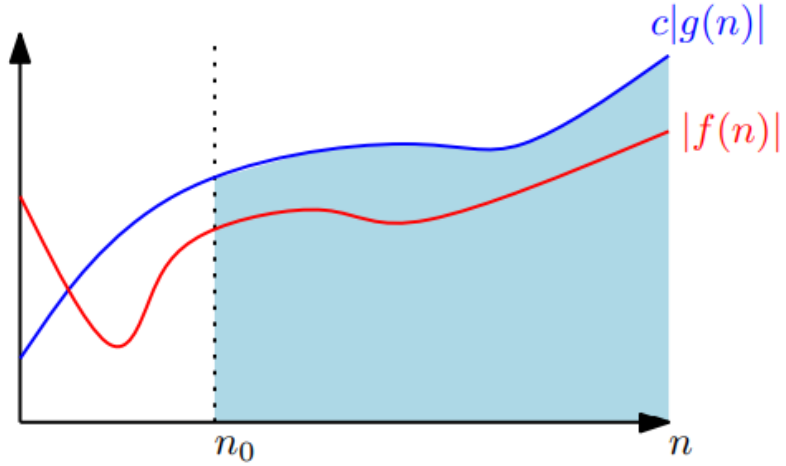
Considerăm o funcție T ce reprezintă complexitatea unui algoritm. Funcția T poate fi:

- constantă: $T(n) = c_0$
- liniară: $T(n) = c_0 \cdot N + c_1$
- pătratică: $T(n) = c_0 \cdot N^2 + c_1 \cdot N + c_2$
- polinomială: $T(n) = c_0 \cdot N^p + c_1 \cdot N^{p-1} + \cdots + c_p, p > 2$
- exponențială: $T(n) = c_0 \cdot a^n, a > 1$



Clasa O

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}_+ | \exists c \in \mathbb{R}_+, c > 0 \text{ și } n_0 \in \mathbb{N} \text{ astfel încât } 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0\}$$

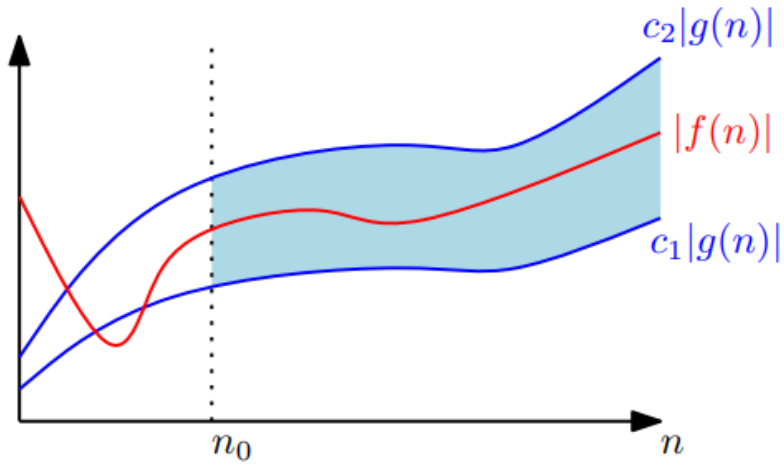


$$f(n) = O(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$$

- $f(n) = O(g(n))$ înseamnă că, pentru valori mai mari de dimensiunea problemei, $c \cdot g(n)$ este o limită superioară pentru $f(n)$ (alg. cu complexitatea $f(n)$ va fi mereu mai optim decât această limită pentru $n \geq n_o$)

Clasa Θ

$$\Theta(g(n)) = \{f : \mathbb{N} \to \mathbb{R}_+ | \exists c_1, c_2 \in \mathbb{R}_+, c_1 > 0, c_2 > 0 \text{ și } n_0 \in \mathbb{N} \text{ astfel încât } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0\}$$



$$f(n) = \Theta(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \text{ și}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$$

- $f(n) = \Theta(g(n))$ înseamnă că, pentru valori mai mari de dimensiunea problemei, algoritmul cu complexitatea $f(n)$ tinde să se comporte *cam* ca $g(n)$

1. $T(n) = 3n^2 + 4n + 5 \in O(n^2)$
2. $T(n) = 3n^2 + 4n + 5 \in O(n^3)$
3. $T(n) = 3n^2 + 4n + 5 \in O(n^4)$
4. $T(n) = 3n^2 + 4n + 5 \notin O(n)$
5. $T(n) = 3n^2 + 4n + 5 \in O(n!)$
6. $T(n) = 3n^2 + 4n + 5 \in \Theta(n^2)$
7. $T(n) = 3n^2 + 4n + 5 \notin \Theta(n^3)$
8. $T(n) = 3n^2 + 4n + 5 \notin \Theta(n^4)$
9. $T(n) = 3n^2 + 4n + 5 \notin \Theta(n)$
10. $T(n) = 3n^2 + 4n + 5 \notin \Theta(n!)$

Observație

Pentru analize de complexitate cât mai precise, preferăm notația Θ.

Exemplu pentru calculul complexității

Cerință: Implementați o funcție pentru calcularea sumei:

$$\sum_{i=1}^{nr} i = 1 + 2 + 3 + \cdots + nr$$

```
1  int suma1(int nr) {
2      int rezultat = (nr * (nr + 1)) / 2;
3      return rezultat;
4  }
```

Complexitatea temporală:

Exemplu pentru calculul complexității

Cerință: Implementați o funcție pentru calcularea sumei:

$$\sum_{i=1}^{nr} i = 1 + 2 + 3 + \cdots + nr$$

```
1  int suma1(int nr) {
2      int rezultat = (nr * (nr + 1)) / 2;
3      return rezultat;
4  }
```

Complexitatea temporală: $O(1)$ sau $\Theta(1)$

Exemplu pentru calculul complexității

Cerință: Implementați o funcție pentru calcularea sumei:

$$\sum_{i=1}^{nr} i = 1 + 2 + 3 + \cdots + nr$$

```
1  int suma1(int nr) {
2      int rezultat = (nr * (nr + 1)) / 2;
3      return rezultat;
4  }
```

Complexitatea temporală: $O(1)$ sau $\Theta(1)$

```
5  int suma2(int nr) {
6      int rezultat = 0, i;
7      for (i = 1; i <= nr; i++)
8          rezultat += i;
9      return rezultat;
10 }
```

Complexitatea temporală:

Exemplu pentru calculul complexității

Cerință: Implementați o funcție pentru calcularea sumei:

$$\sum_{i=1}^{nr} i = 1 + 2 + 3 + \cdots + nr$$

```
1  int suma1(int nr) {
2      int rezultat = (nr * (nr + 1)) / 2;
3      return rezultat;
4  }
```

Complexitatea temporală: $O(1)$ sau $\Theta(1)$

```
5  int suma2(int nr) {
6      int rezultat = 0, i;
7      for (i = 1; i <= nr; i++)
8          rezultat += i;
9      return rezultat;
10 }
```

Complexitatea temporală: $O(nr)$ sau $\Theta(nr)$

Exemplu pentru calculul complexității

Cerință: Implementați o funcție care calculează suma primelor elemente impare dintr-un vector.

```
1  int suma(int *vect, int n) {
2      int rezultat = 0, i;
3      for (i = 0; i < n; i++) {
4          if (vect[i] % 2 == 0)
5              break;
6          rezultat += vect[i];
7      }
8      return rezultat;
9  }
10 int vect1[] = {1, 5, 3, 7, 8}, vect2[] ={1, 5, 2, 3, 7},
   ↪  vect3[] = {2, 1, 5, 3, 7};
11 printf("%d %d %d\n", suma(vect1, 5), suma(vect2, 5),
   ↪  suma(vect3, 5));
```

Complexitatea temporală: ??

Colecții și Mulțimi

- O colecție este un grup de elemente de același tip în care pot exista duplicate.
- O mulțime este o colecție sortată ce nu conține duplicate.
- O colecție (respectiv mulțime) este un TAD care poate fi descris prin operațiile de bază care se pot executa asupra acestei structuri.
- Operațiile de bază asupra unei colecții pot fi descrise abstract, independent de tipul elementelor componente și de modul de reprezentare utilizat.
- Pentru implementarea acestor operații, avem nevoie de reprezentarea datelor în memorie, deci de o **structură de date**.

Definirea TAD-ului pentru o colecție

Constructori

- Aceștia au ca rezultat o colecție nouă cu elemente de tip T.
- Considerăm ca nume pentru TAD *TCollection*.
  1. Inițializarea colecției – $init :\rightarrow TCollection$
  2. Adăugarea unui element – $add : TCollection \times T \rightarrow TCollection$
  3. Eliminarea unui element – $delete : TCollection \times T \rightarrow TCollection$
  4. Reuniunea – $union : TCollection \times TCollection \rightarrow TCollection$
  5. Intersecția – $intersection : TCollection \times TCollection \rightarrow TCollection$
  6. Diferență – $difference : TCollection \times TCollection \rightarrow TCollection$

Funcții

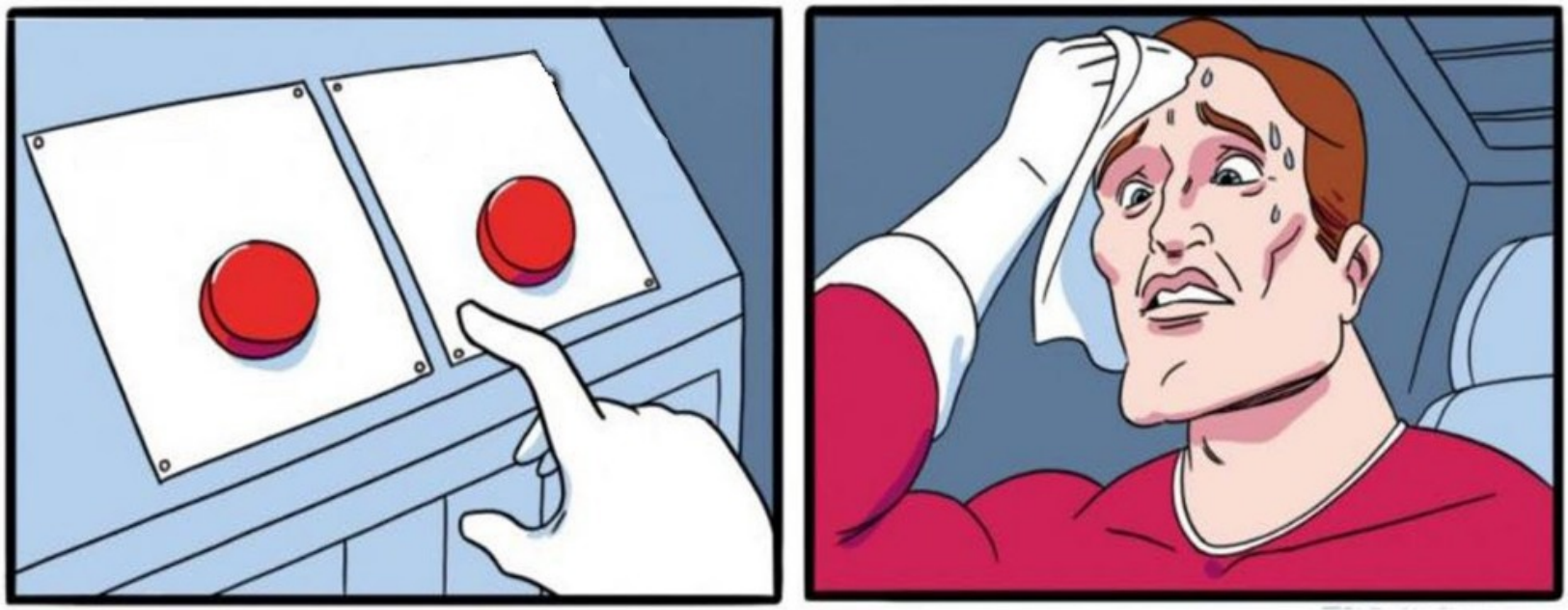
- Operații care furnizează informații despre o colecție.
  1. $empty : TCollection \rightarrow \{0, 1\}$
  2. $size : TCollection \rightarrow Int$
  3. $contains : TCollection \times T \rightarrow \{0, 1\}$

Semantica acestor operații abstracte se poate stabili prin scrierea pseudocodului asociat unei operații.

Implementarea TAD-ului pentru o colecție

Implementare TAD = Structură de date + Algoritm

- Avem nevoie de o structură de date pentru reținerea elementelor unei colecții.
- Ce alegem?



Implementarea TAD-ului pentru o colecție

Implementare TAD = Structură de date + Algoritm

- Avem nevoie de o structură de date pentru reținerea elementelor unei colecții.
- Ce alegem?
  1. Vector alocat static
  2. Vector alocat dinamic
  3. **Listă înlănțuită** *(în cursul următor)*
- Trebuie aleasă o reprezentare care să permită prelucrarea cât mai eficientă a elementelor colecției.
- Daca operațiile de adăugare / eliminare sunt relativ rare, atunci se pot utiliza vectori de elemente. De obicei se poate stabili o limită superioară a numărului de elemente.
- Vectorul elementelor din colecție poate fi alocat:
  1. **Static** – în cazul în care limita superioară este cunoscută din faza de compilare și este valabilă în marea majoritate a cazurilor;
  2. **Dinamic** – în restul situațiilor.

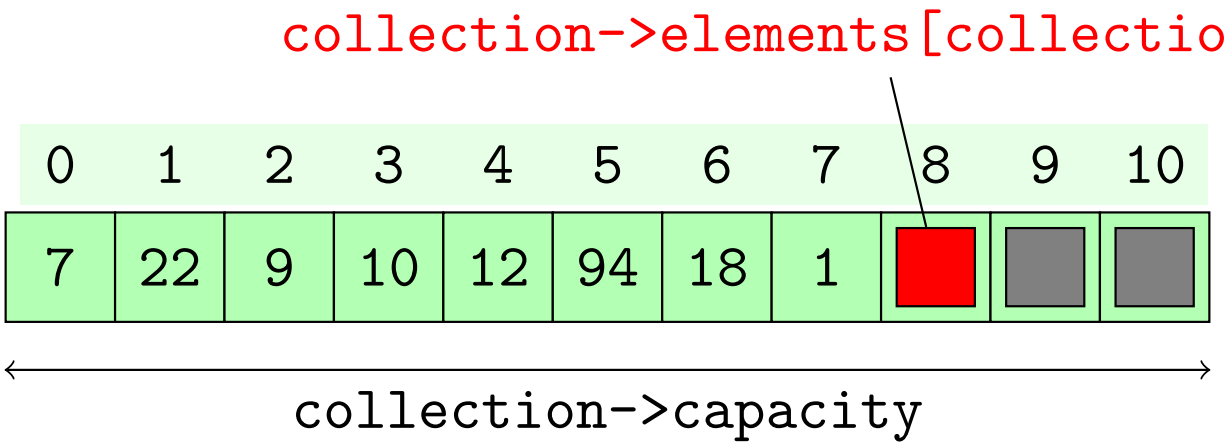
Implementare colecție cu vector alocat dinamic

```
1  typedef int T;
2
3  typedef struct collection {
4      T *elements;
5      long size;
6      long capacity;
7  } *TCollection;
8
9  TCollection init() {
10     TCollection collection = malloc(sizeof(struct
   ↪  collection));
11     collection->size = 0;
12     collection->capacity = 10;
13     collection->elements = malloc(10 * sizeof(T));
14     return collection;
15 }
```

Implementare colecție cu vector alocat dinamic

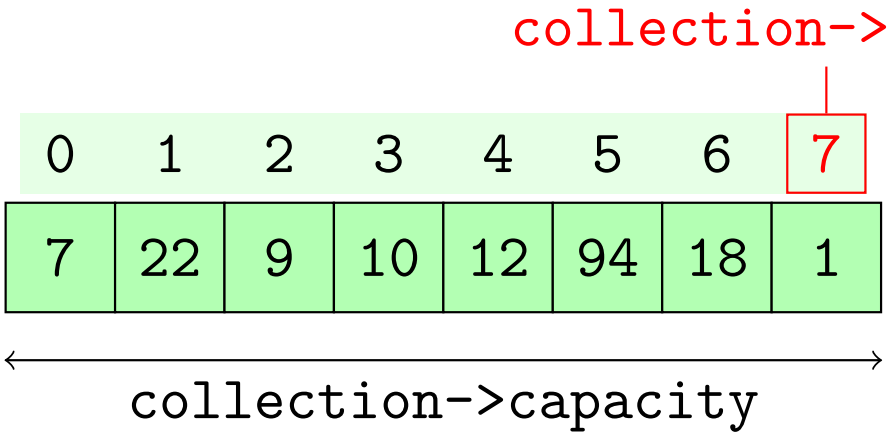
Adăugarea unui element la finalul vectorului

1. Dacă avem alocat spațiu



2. Dacă trebuie să realocăm spațiu

Condiție: `collection->capacity == collection->size`

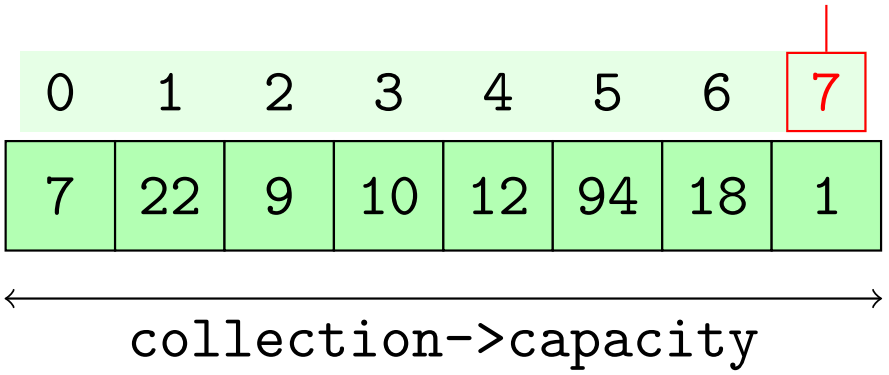


Implementare colecție cu vector alocat dinamic

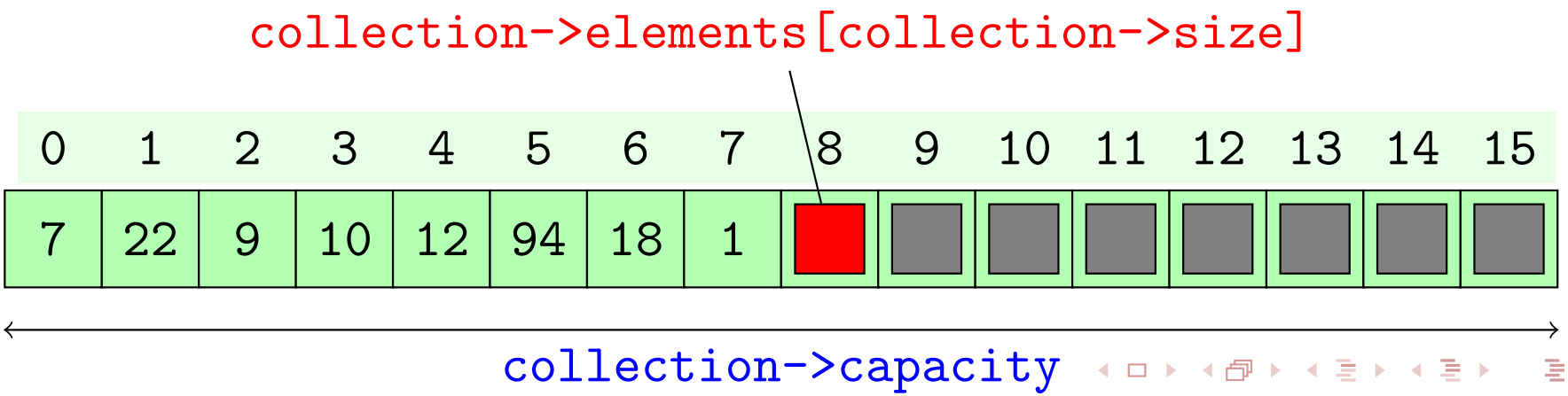
Adăugarea unui element la finalul vectorului

2. Dacă trebuie să realocăm spațiu

   Condiție: `collection->capacity == collection->size`



După realocarea memoriei pentru vector



Implementare colecție cu vector alocat dinamic

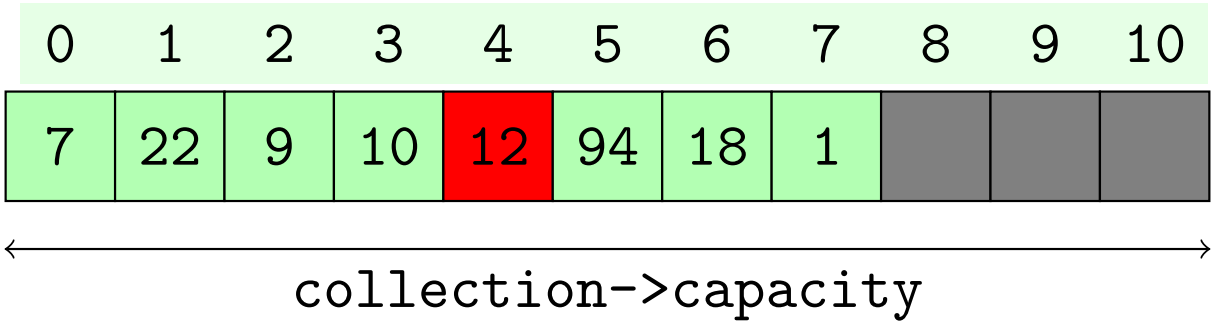
- Presupunem că pornim de la colecție nevidă!

```
16  TCollection add(TCollection collection, T element) {
17      T *vect;
18      if (collection->size == collection->capacity) {
19          collection->capacity *= 2;
20          vect = realloc(collection->elements,
    ↪   collection->capacity * sizeof(T));
21          if (vect == NULL)
22              return collection;
23          collection->elements = vect;
24      }
25      collection->elements[collection->size] = element;
26      collection->size++;
27      return collection;
28  }
```

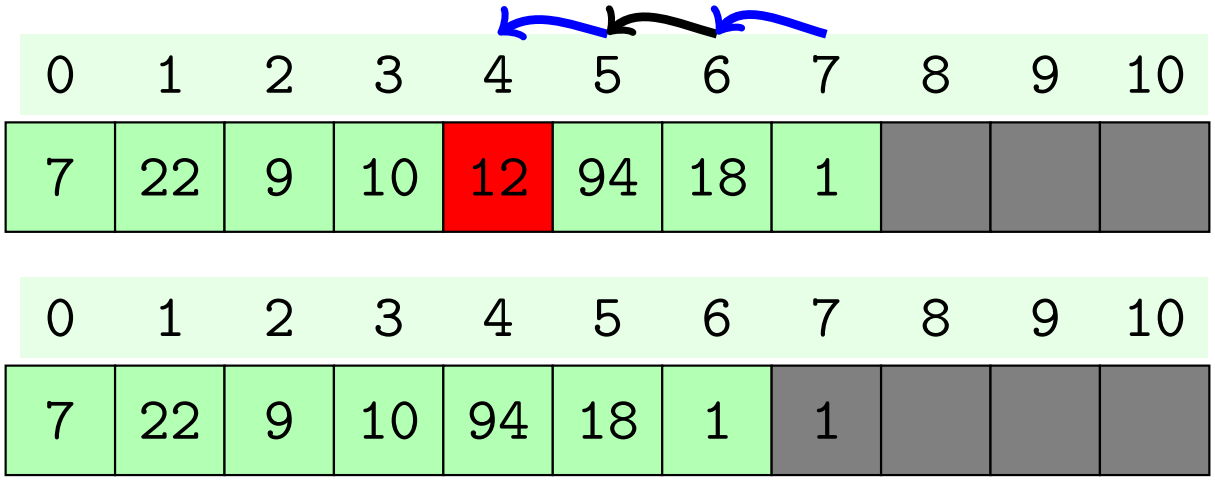
Implementare colecție cu vector alocat dinamic

Eliminarea unui element

1. Determinăm poziția elementului ce urmează să fie eliminat



2. Deplasăm elementele care sunt poziționate după cel care trebuie eliminat



Implementare colecție cu vector alocat dinamic

Eliminarea unui element

```
29  TCollection delete(TCollection collection, T element) {
30      int i, pos = -1;
31      for (i = 0; i < collection->size; i++) {
32          if (collection->elements[i] == element) {
33              pos = i;
34              break;
35          }
36      }
37      if (pos == -1)
38          return collection;
39      for (i = pos; i < collection->size - 1; i++)
40          collection->elements[i] = collection->elements[i+1];
41      collection->size--;
42      return collection;
43  }
```

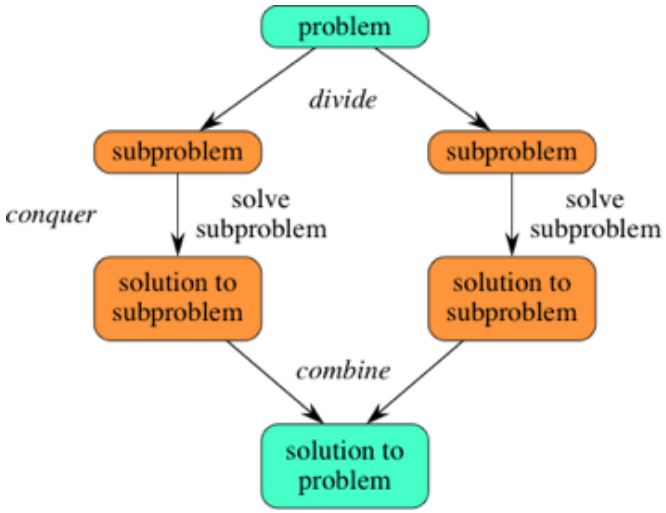
Implementare colecție cu vector alocat dinamic

Operații care furnizează informații despre o colecție

```
44  int empty(TCollection collection) {
45      return collection == NULL || collection->size == 0;
46  }
47  int size(TCollection collection) {
48      if (collection == NULL)
49          return 0;
50      return collection->size;
51  }
52  int contains(TCollection collection, T element) {
53      int i;
54      for (i = 0; i < collection->size; i++) {
55          if (collection->elements[i] == element)
56              return 1;
57      }
58      return 0;
59  }
```

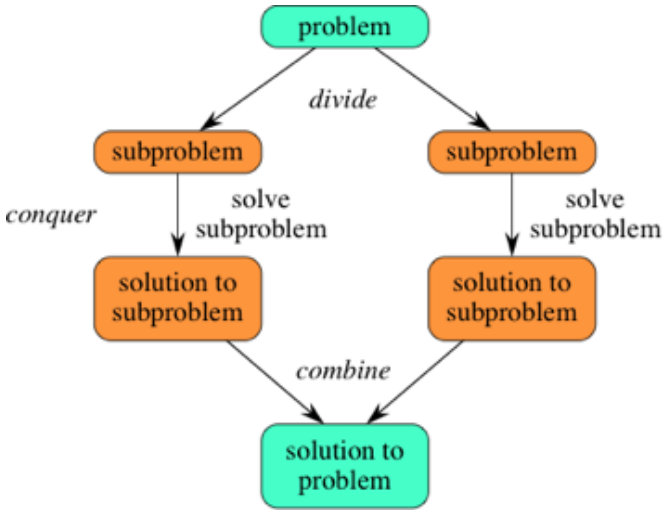
Tehnica Divide et Impera

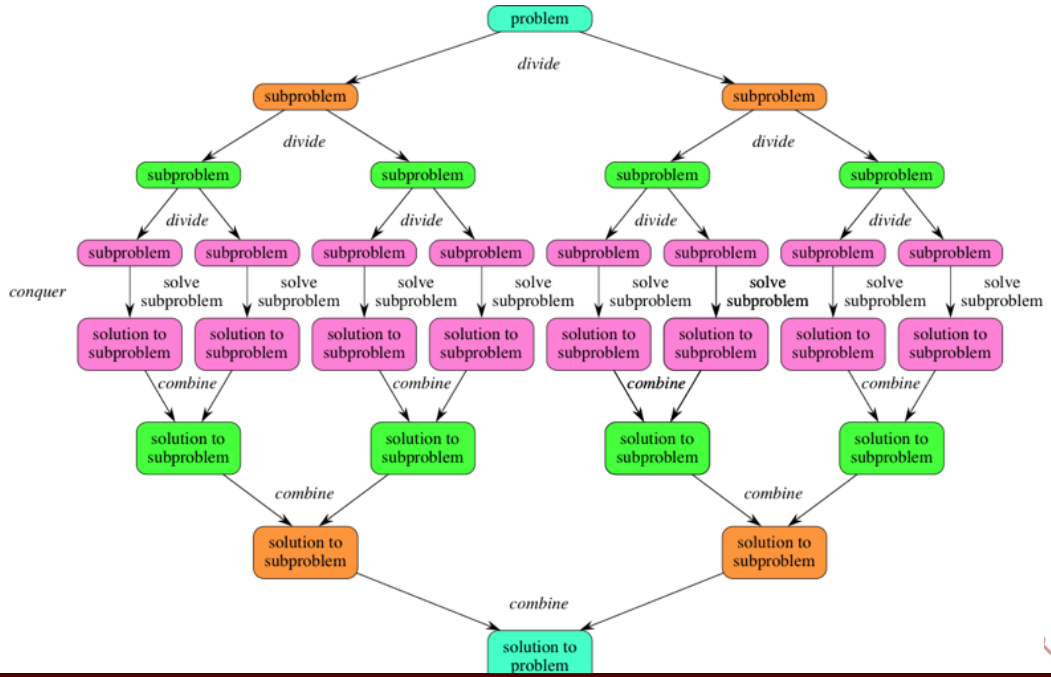
- Mulți algoritmi utili au o structură recursivă: pentru a rezolva o problemă dată, aceștia sunt apelați de către ei înșiși o dată sau de mai multe ori pentru a rezolva subprobleme apropiate.
- Acești algoritmi folosesc, de obicei, tehnica de programare numită **Divide et Impera**: ei împart problema în mai multe probleme similare problemei inițiale, dar de dimensiune mai mică, le rezolvă în mod recursiv și apoi le combină pentru a crea o soluție a problemei inițiale.



Tehnica Divide et Impera

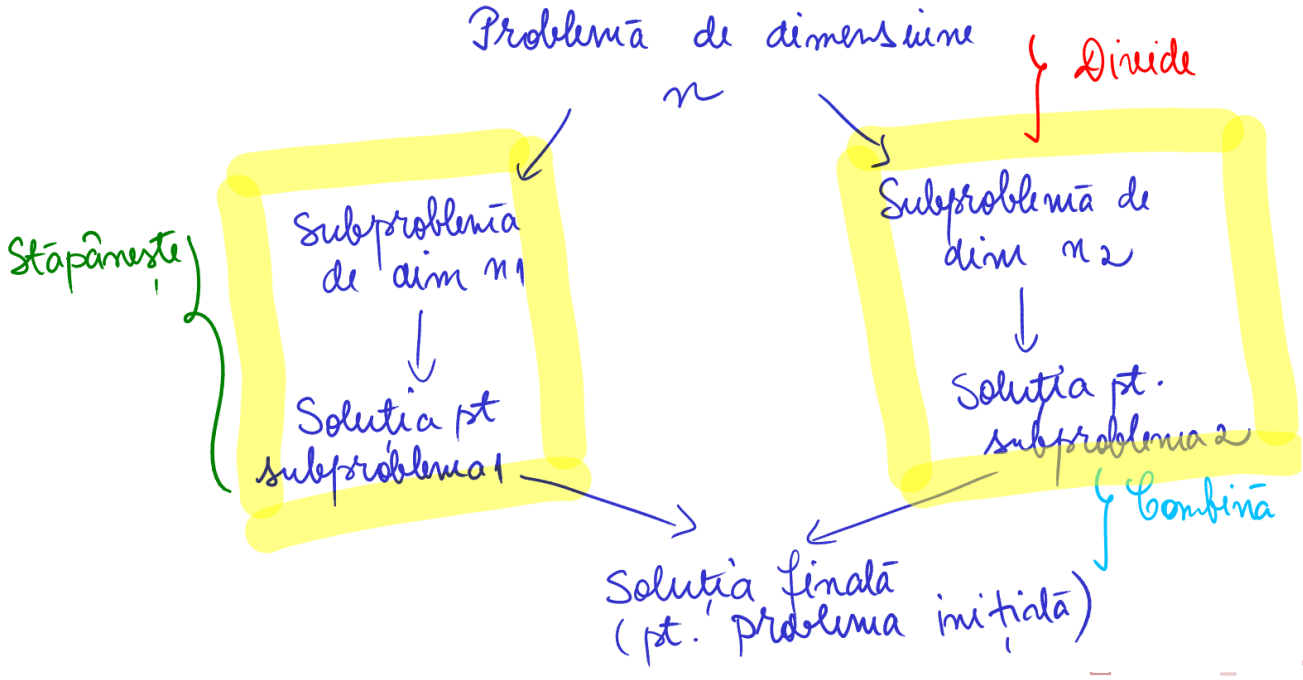
- Mulți algoritmi utili au o structură recursivă: pentru a rezolva o problemă dată, aceștia sunt apelați de către ei înșiși o dată sau de mai multe ori pentru a rezolva subprobleme apropiate.
- Acești algoritmi folosesc, de obicei, tehnica de programare numită **Divide et Impera**: ei împart problema în mai multe probleme similare problemei inițiale, dar de dimensiune mai mică, le rezolvă în mod recursiv și apoi le combină pentru a crea o soluție a problemei inițiale.





Tehnica Divide et Impera

- Tehnica **Divide et Impera** implică trei pași la fiecare nivel de recursivitate:
  1. **Divide** problema într-un număr de subprobleme.
  2. **Stăpânește** subproblemele prin rezolvarea acestora în mod recursiv. Dacă dimensiunile acestora sunt suficient de mici, rezolvă subproblemele în modul uzual, nerecursiv.
  3. **Combină** soluțiile tuturor subproblemelor în soluția finală pentru problema inițială.



Căutare binară

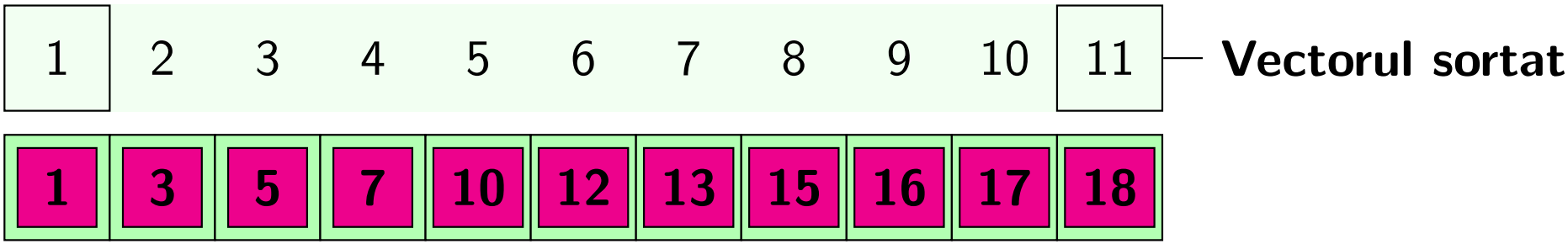
Problemă

Se dau un vector cu **nr** elemente **sortate crescător** și o valoare **x**. Să se verifice dacă valoarea **x** se află în vectorul dat sau nu.

1. Algoritmul funcționează pe baza tehnicii **Divide et Impera**.
2. Valoarea căutată este comparată cu cea a elementului din mijlocul vectorului.
   - A. Dacă este egală cu cea a acelui element, algoritmul se termină.
   - B. Dacă este mai mare decât acea valoare, algoritmul se reia de la mijlocul vectorului până la sfârșit.
   - C. Dacă este mai mică decât acea valoare, algoritmul se reia pentru elementele de la începutul vectorului până la mijloc.

Căutare binară

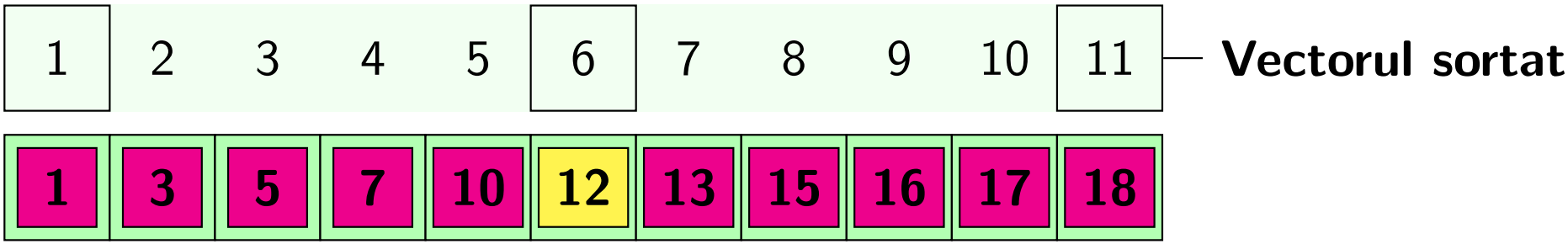
Exemplu – I



Valoarea căutată: 10

Căutare binară

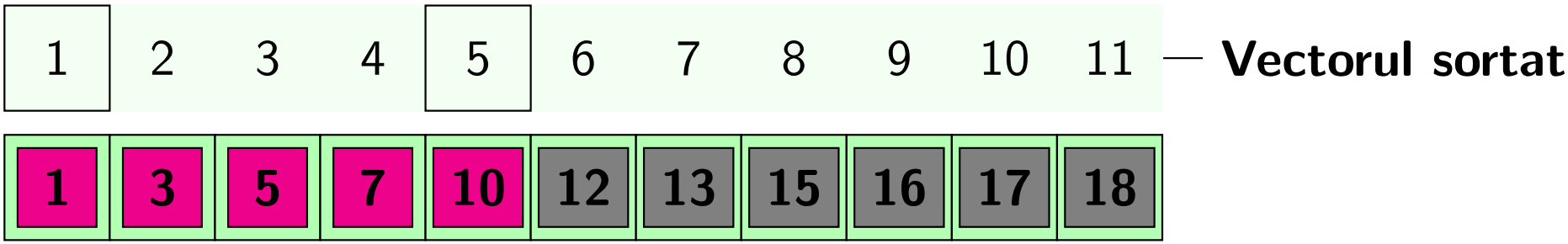
Exemplu – II



Valoarea căutată: 10

Căutare binară

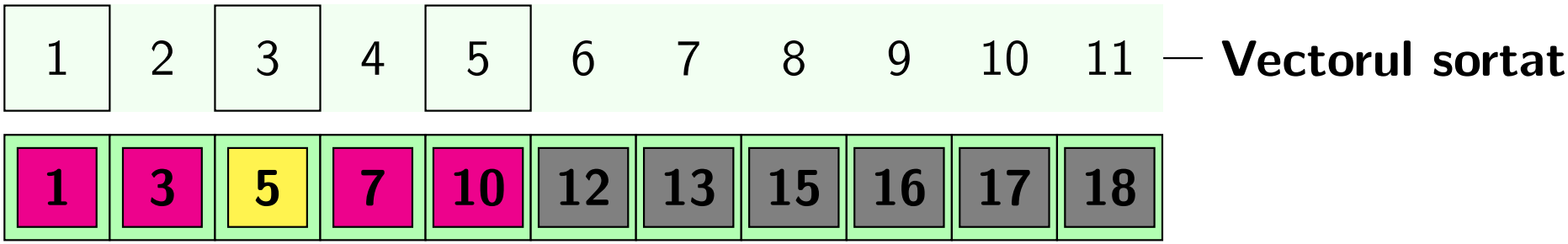
Exemplu – III



Valoarea căutată: 10

Căutare binară

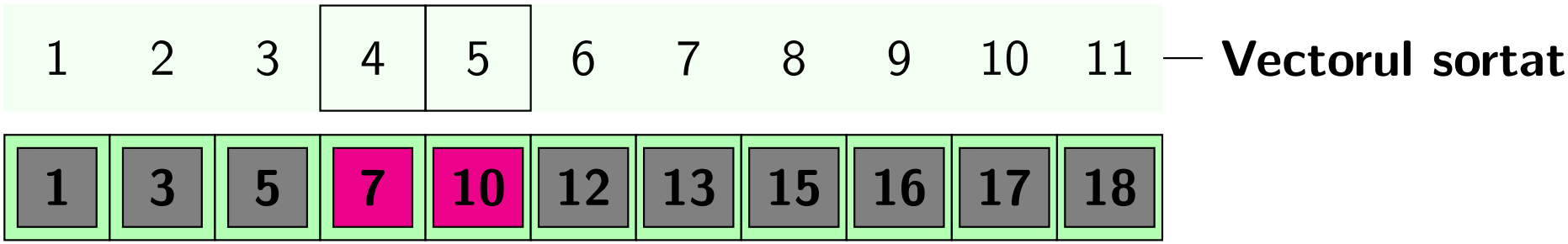
Exemplu – IV



Valoarea căutată: 10

Căutare binară

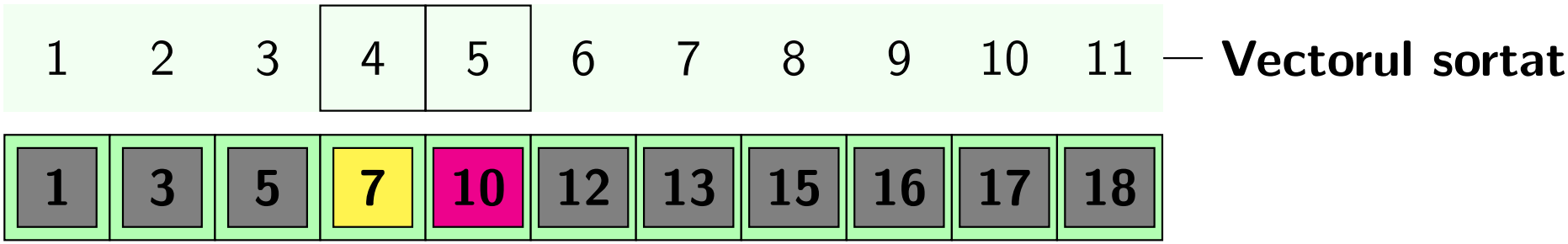
Exemplu – V



Valoarea căutată: 10

Căutare binară

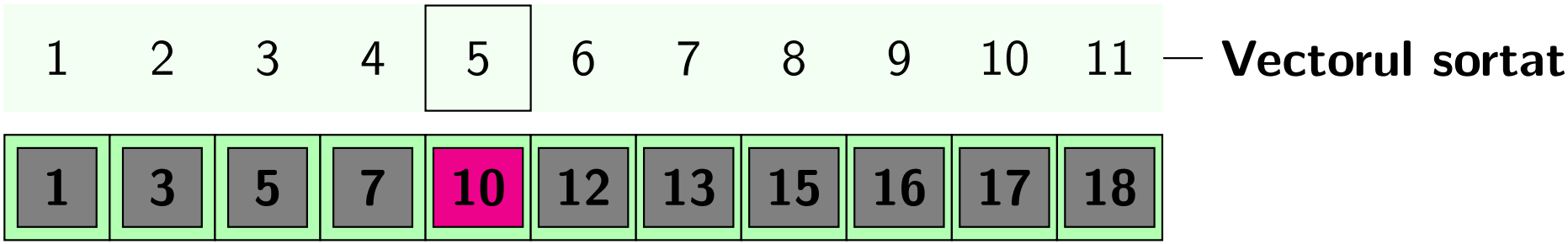
Exemplu – VI



Valoarea căutată: 10

Căutare binară

Exemplu – VII



Valoarea căutată: 10

Căutare binară

```
function BinarySearch(vect, start, end, item)
    if end ≥ start then
        mid ← start + (end − start)/2
        if vect[mid] == item then
            return mid
        end if
        if vect[mid] > item then
            return BinarySearch(vect, start, mid − 1, item)
        end if
        return BinarySearch(vect, mid + 1, end, item)
    end if
    return −1
end function
```

Căutare binară

```
function BinarySearch(vect, start, end, item)
    if end ≥ start then
        mid ← start + (end − start)/2
        if vect[mid] == item then
            return mid
        end if
        if vect[mid] > item then
            return BinarySearch(vect, start, mid − 1, item)
        end if
        return BinarySearch(vect, mid + 1, end, item)
    end if
    return −1
end function
```

Cum calculăm complexitatea?

$$T(n) = ??$$

Căutare binară

```
function BinarySearch(vect, start, end, item)
    if end ≥ start then
        mid ← start + (end − start)/2
        if vect[mid] == item then
            return mid
        end if
        if vect[mid] > item then
            return BinarySearch(vect, start, mid − 1, item)
        end if
        return BinarySearch(vect, mid + 1, end, item)
    end if
    return −1
end function
```

Cum calculăm complexitatea?

$$T(n) = T(n/2) + ??$$

Căutare binară

```
function BinarySearch(vect, start, end, item)
    if end ≥ start then
        mid ← start + (end − start)/2
        if vect[mid] == item then
            return mid
        end if
        if vect[mid] > item then
            return BinarySearch(vect, start, mid − 1, item)
        end if
        return BinarySearch(vect, mid + 1, end, item)
    end if
    return −1
end function
```

Cum calculăm complexitatea?

$$T(n) = T(n/2) + O(1) \text{ și } T(1) = O(1)$$

Căutare binară – Calculul complexității

$$T(n) = T\left(\frac{n}{2^1}\right) + O(1)$$
$$T\left(\frac{n}{2^1}\right) = T\left(\frac{n}{2^2}\right) + O(1)$$
$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + O(1)$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$T\left(\frac{n}{2^k}\right) = T\left(\frac{n}{2^{k+1}}\right) + O(1)$$

Ne oprim când ajungem la cazul de bază ($T(1) = O(1)$):
$\frac{n}{2^{k+1}} = 1 \Rightarrow n = 2^{k+1} \Rightarrow \log_2 n = k + 1$

Căutare binară – Calculul complexității

Ne oprim când ajungem la cazul de bază ($T(1) = O(1)$):
$\frac{n}{2^{k+1}} = 1 \Rightarrow n = 2^{k+1} \Rightarrow \log_2 n = k + 1$
Dacă însumăm obținem:

$$T(n) = T\left(\frac{n}{2^1}\right) + O(1)$$
$$T\left(\frac{n}{2^1}\right) = T\left(\frac{n}{2^2}\right) + O(1)$$
$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + O(1)$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$T\left(\frac{n}{2^k}\right) = T\left(\frac{n}{2^{k+1}}\right) + O(1)$$

$$T(n) = T\left(\frac{n}{2^{k+1}}\right) + \sum_{i=1}^{k+1} O(1) = T(1) + (k+1)O(1) =$$
$$= O(1) + O(k+1) = O(1) + O(\log_2 n) = O(\log_2 n)$$

Merge Sort

Ideea algoritmului

- Se împarte vectorul, în secvente din ce în ce mai mici, astfel încât fiecare secvență să fie ordonată la un moment dat și interclasata cu o altă secvență din vector.
- Practic, interclasarea va începe când se ajunge la o secvență formata din două elemente. Aceasta, odată ordonată, se va interclasa cu o alta corespunzătoare. Cele două secvențe vor alcătui un subșir ordonat din vector care, la rândul lui, se va interclasa cu subșirul corespunzător **șamd**.

Algoritmul Merge Sort I

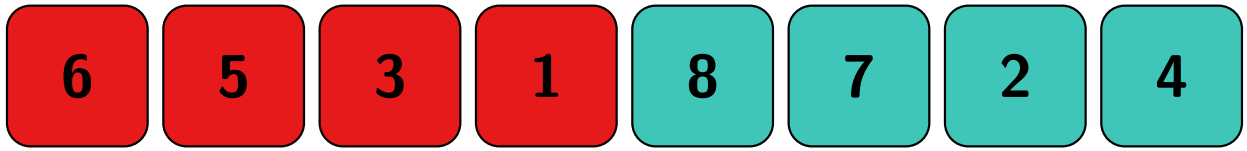
```
1:  function MERGE(V, start, pivot, end)
2:      B[start...end] ← 0
3:      i ← start
4:      k ← start
5:      j ← pivot + 1
6:      while i ≤ pivot && j ≤ end do
7:          if A[i] ≤ A[j] then
8:              B[k++] ← A[i++]
9:          else
10:             B[k++] ← A[j++]
11:         end if
12:     end while
13:     while i ≤ pivot do
14:         B[k++] ← A[i++]
15:     end while
```

Algoritmul Merge Sort II

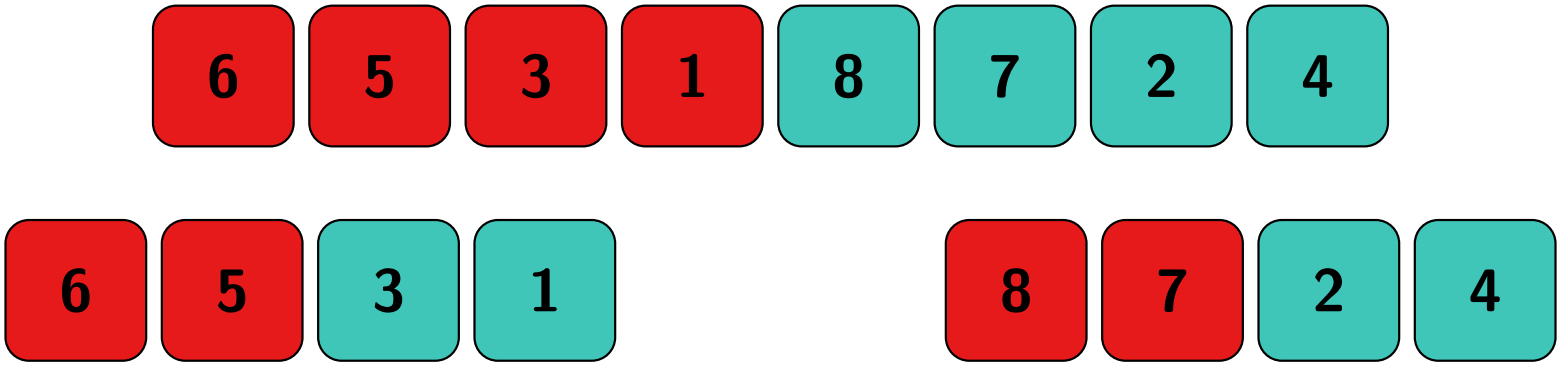
```
16:     while j ≤ end do
17:         B[k++] ← A[j++]
18:     end while
19:     for i ← start to end do
20:         A[i] ← B[i]
21:     end for
22: end function
```

```
1: function MergeSort(V, start, end)
2:     if start < end then
3:         pivot ← start + (end − start)/2
4:         mergeSort(V, start, pivot)
5:         mergeSort(V, pivot + 1, end)
6:         Merge(V, start, pivot, end)
7:     end if
8: end function
```

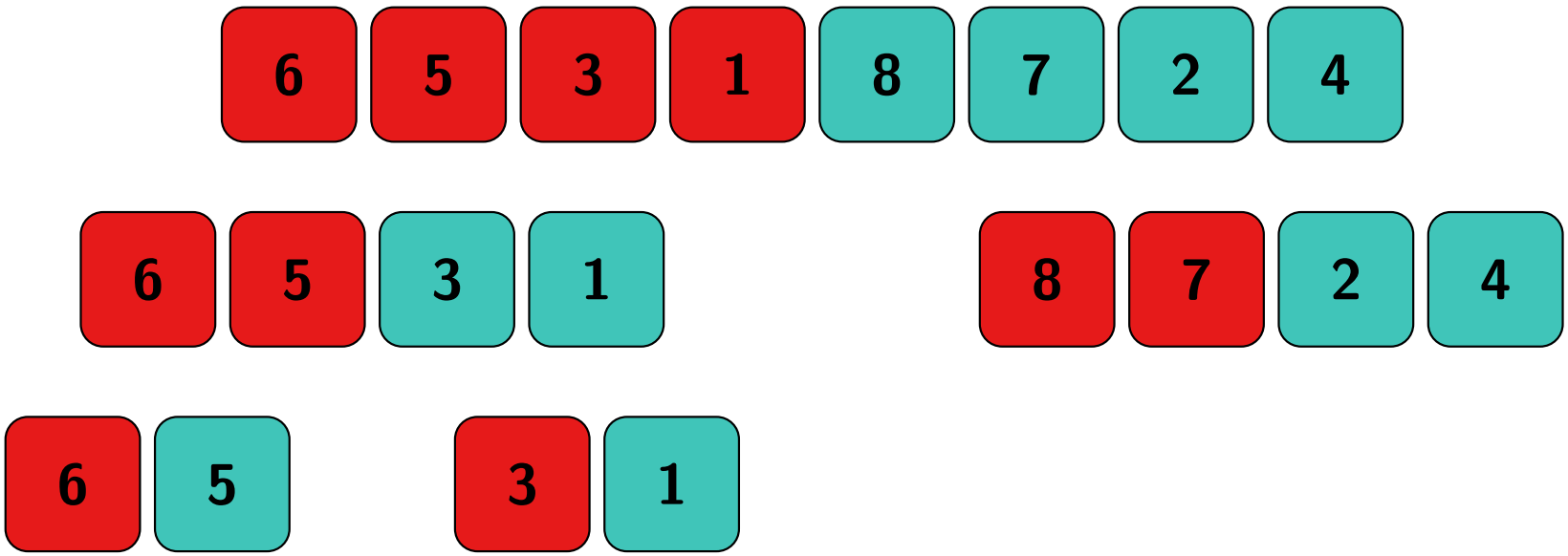
Merge Sort - Divide



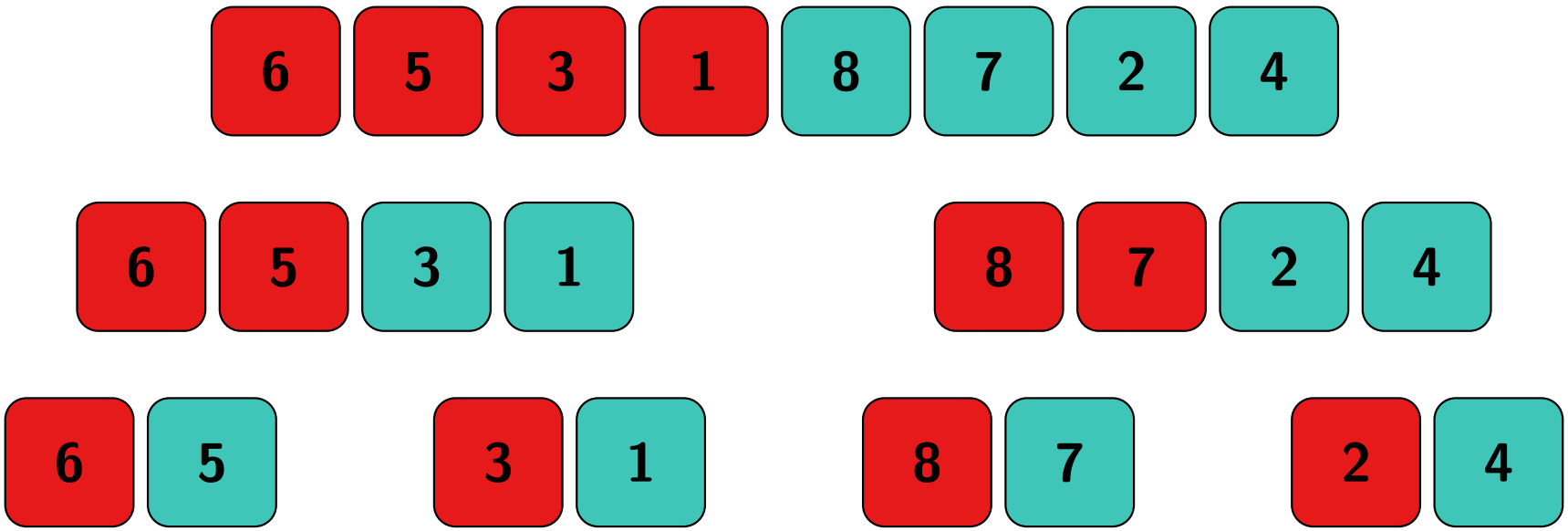
Merge Sort - Divide



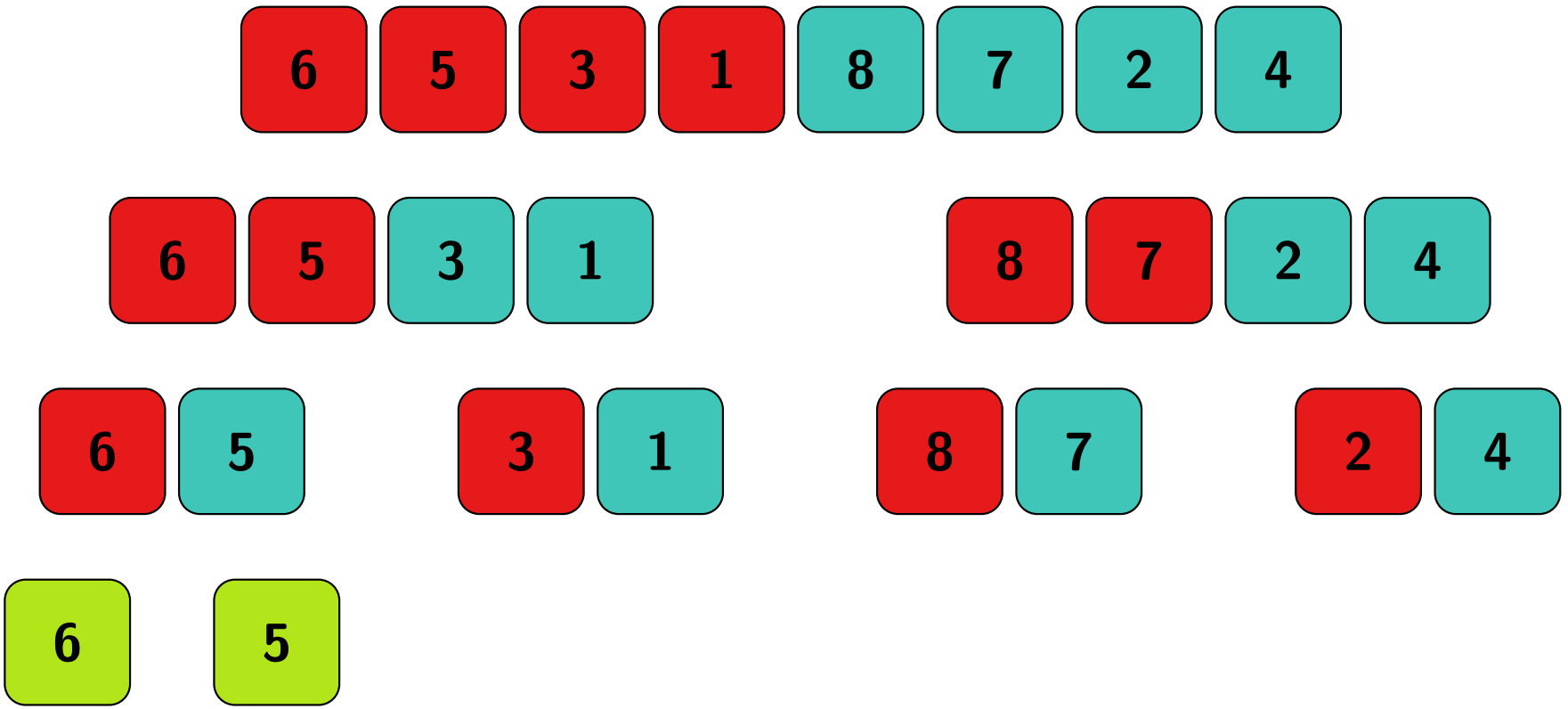
Merge Sort - Divide



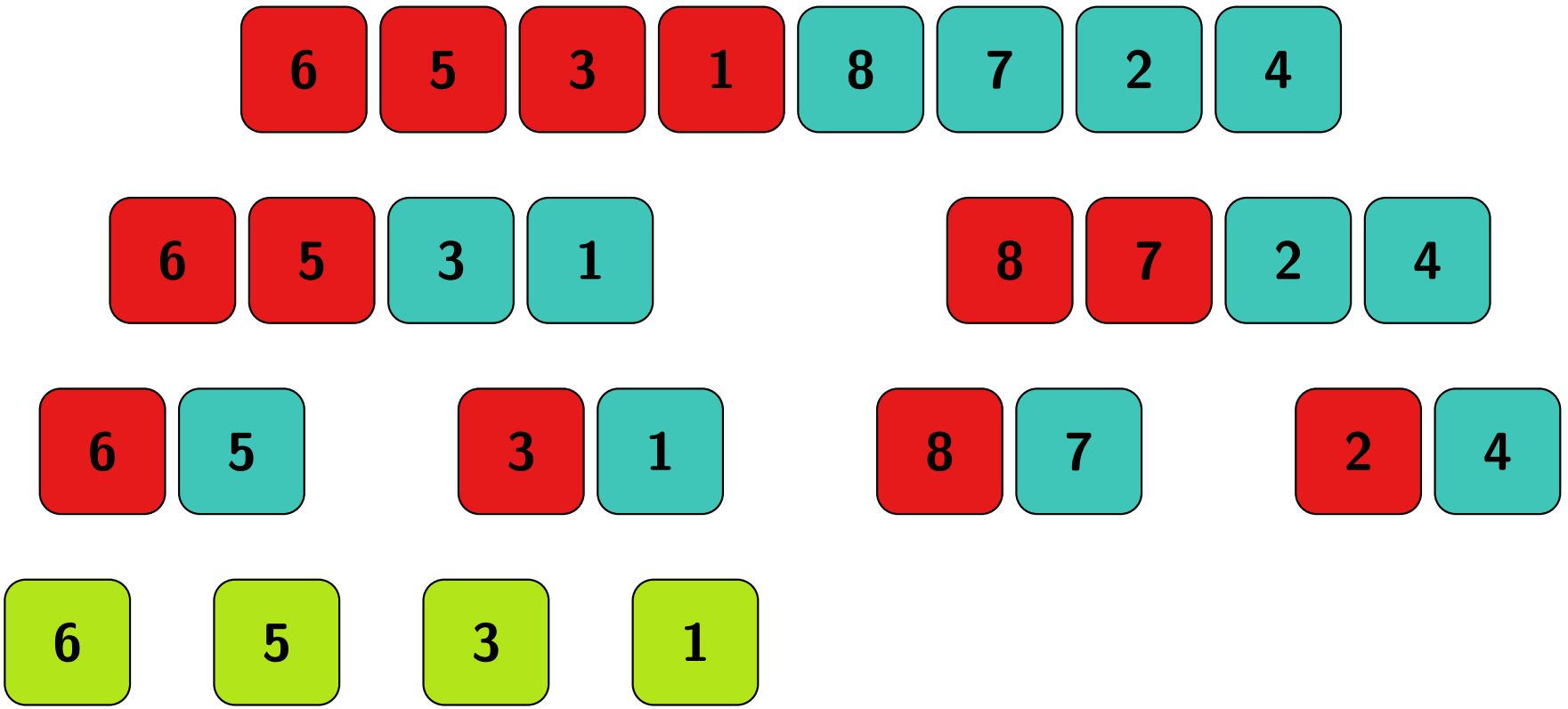
Merge Sort - Divide



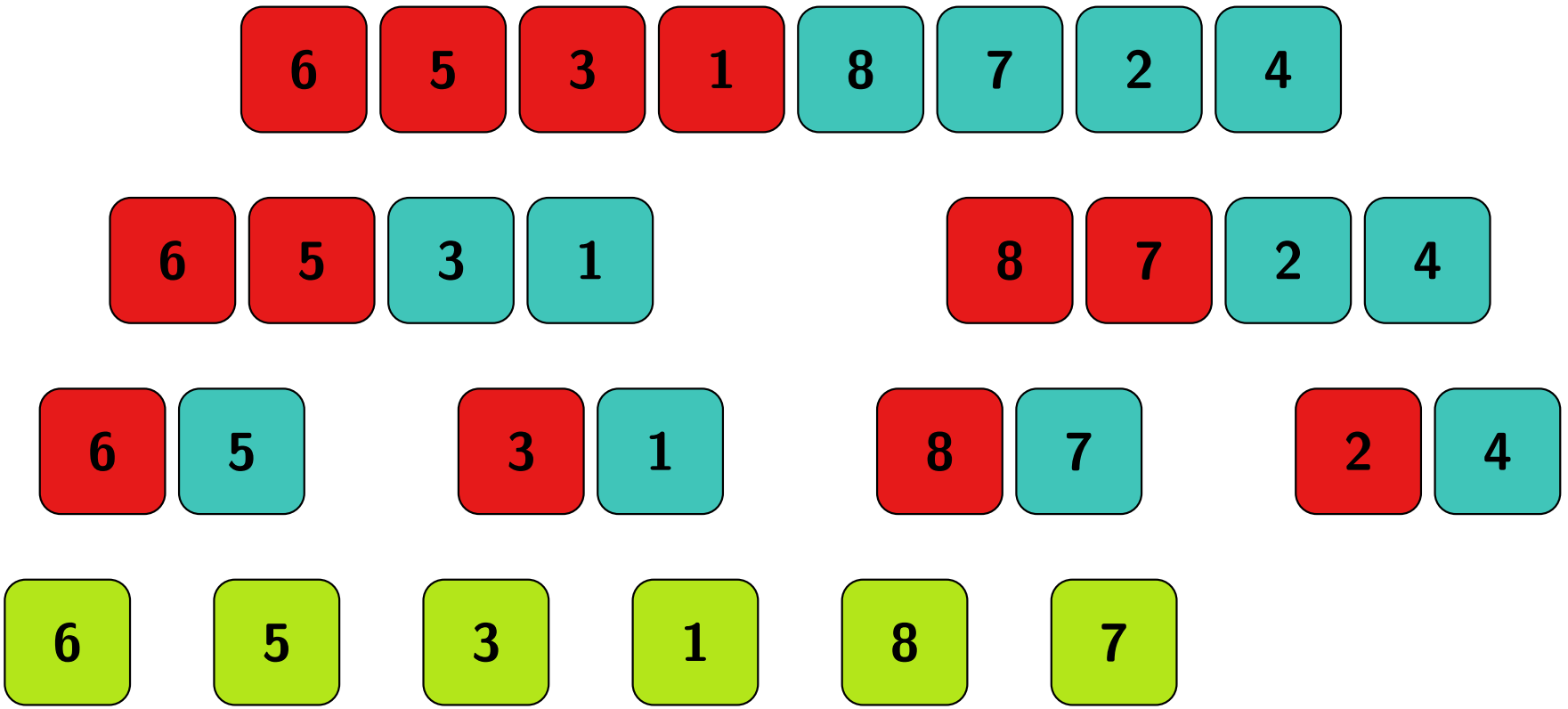
Merge Sort - Divide



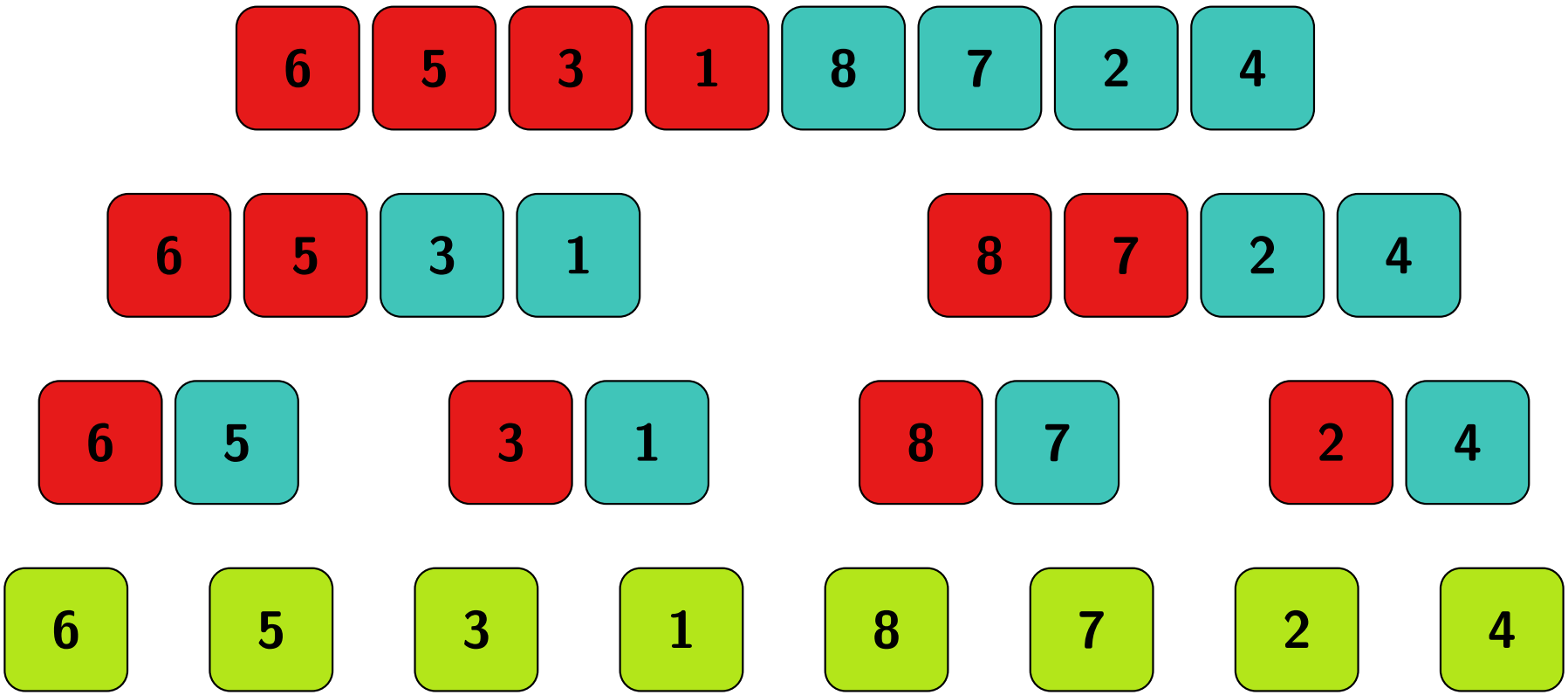
Merge Sort - Divide



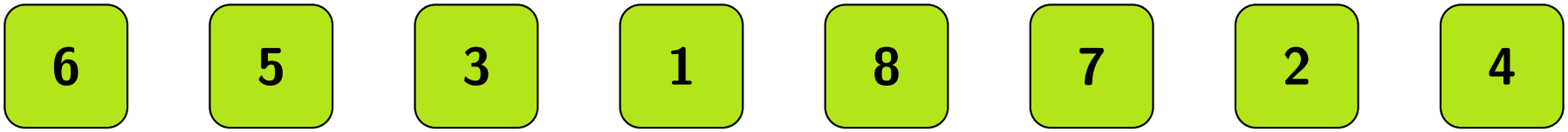
Merge Sort - Divide



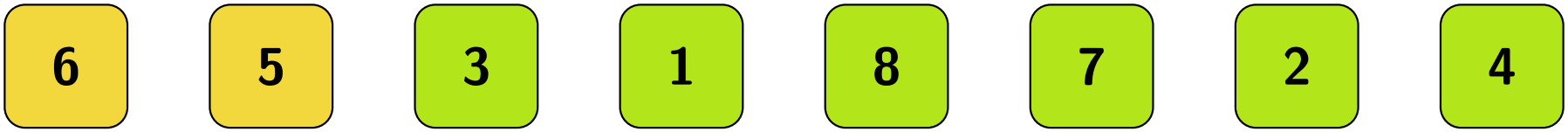
Merge Sort - Divide



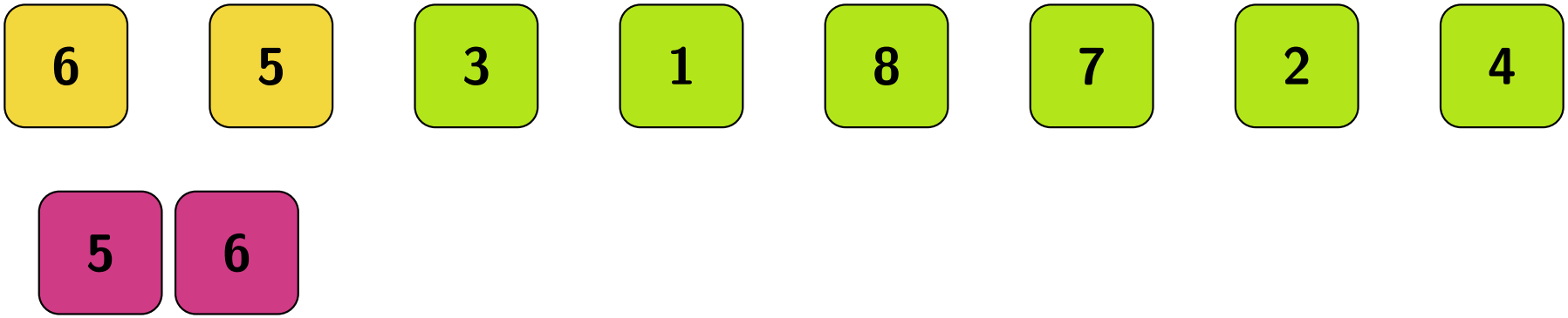
Merge Sort - Merge



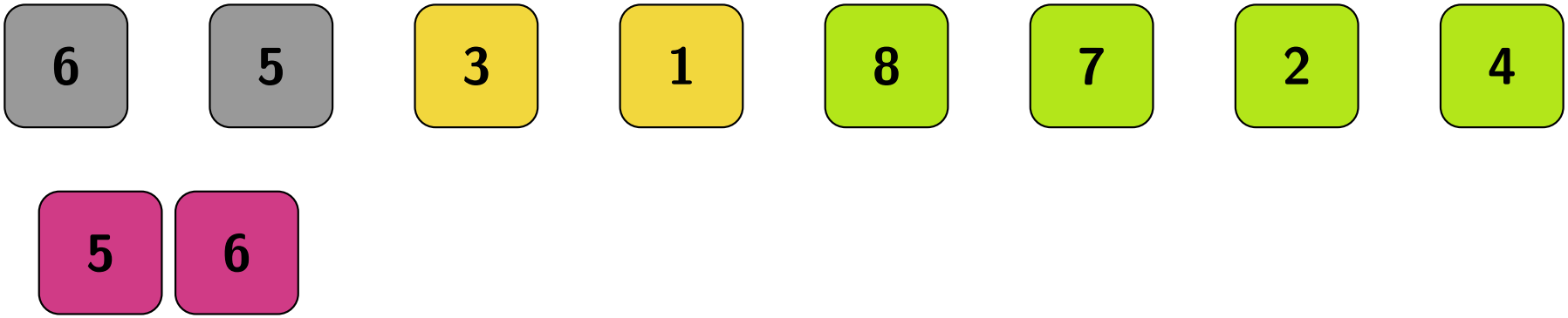
Merge Sort - Merge



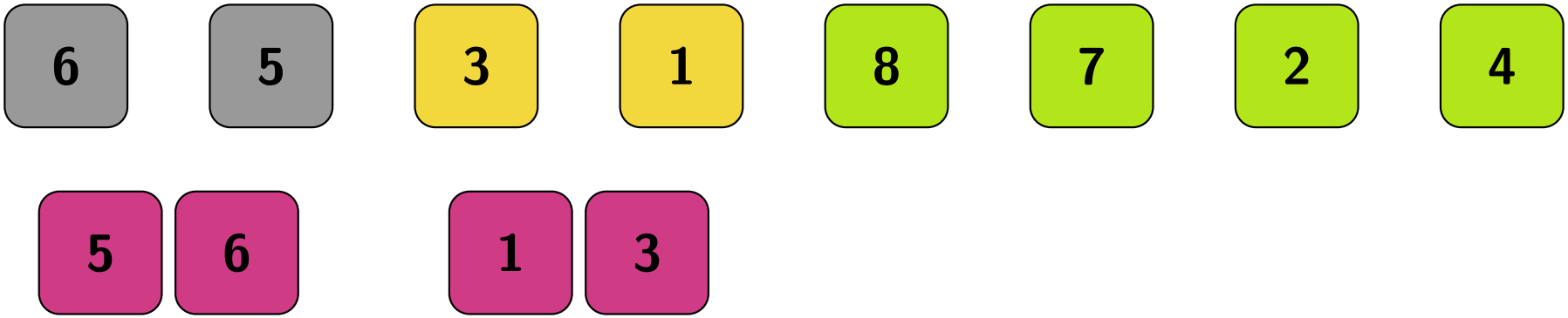
Merge Sort - Merge



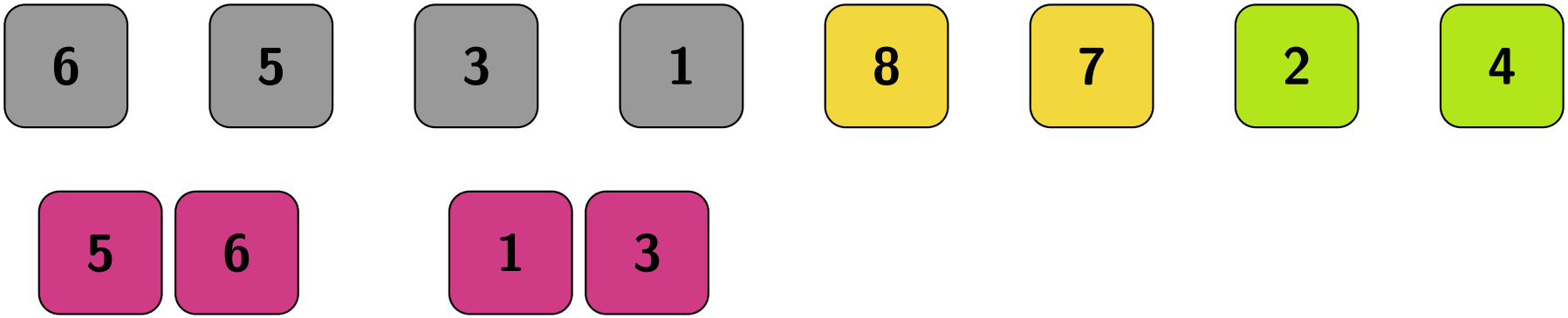
Merge Sort - Merge



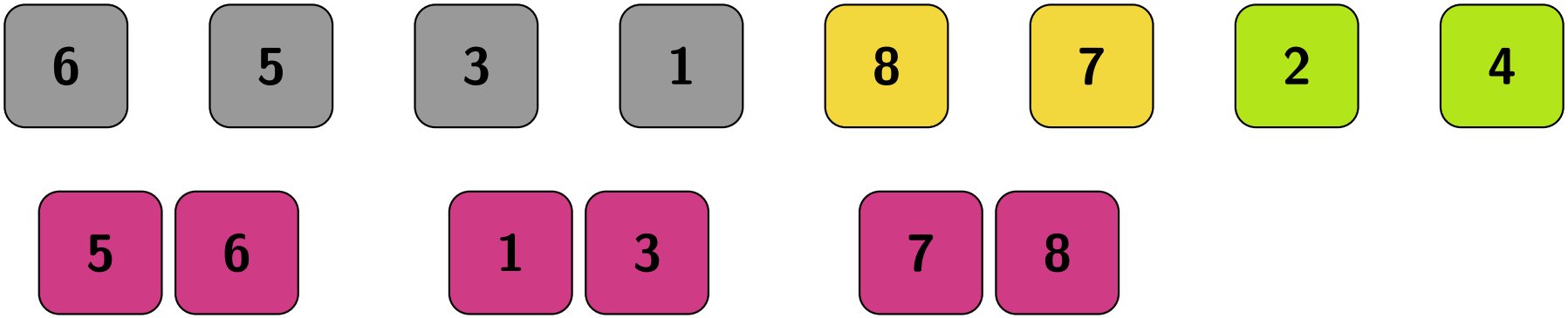
Merge Sort - Merge



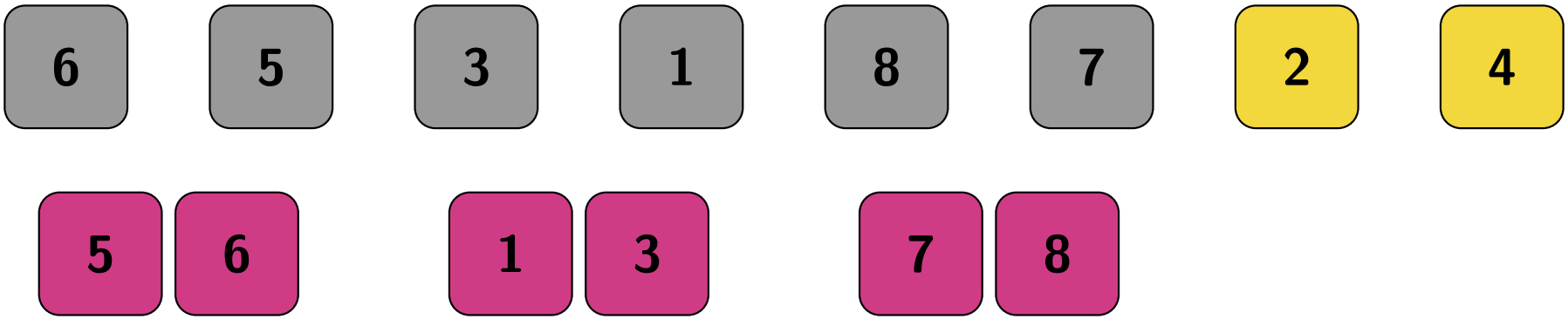
Merge Sort - Merge



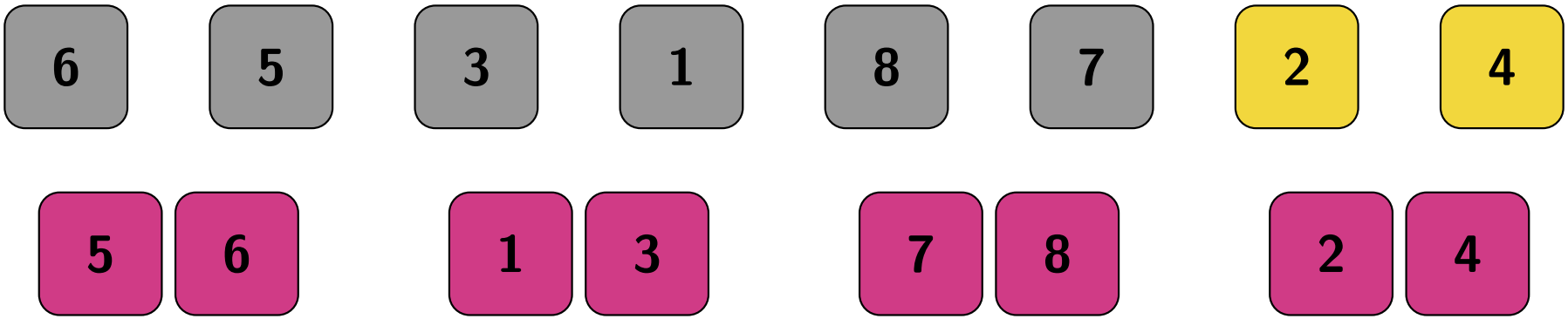
Merge Sort - Merge



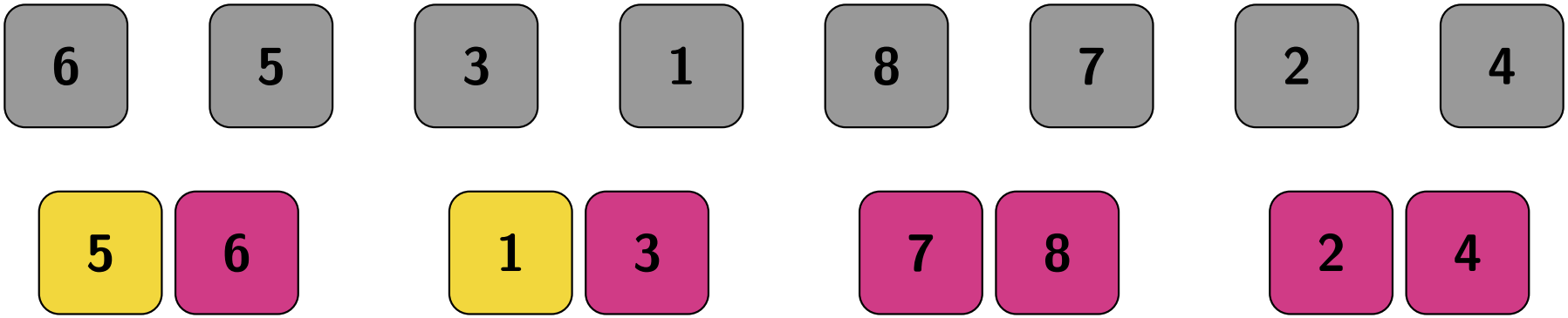
Merge Sort - Merge



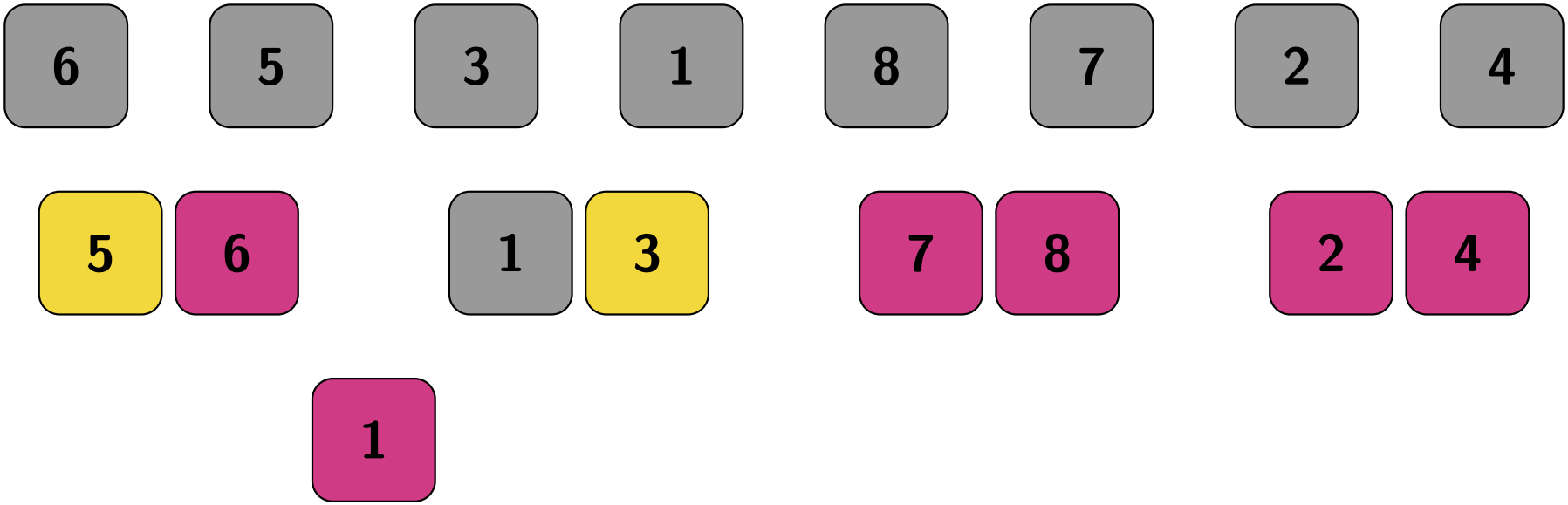
Merge Sort - Merge



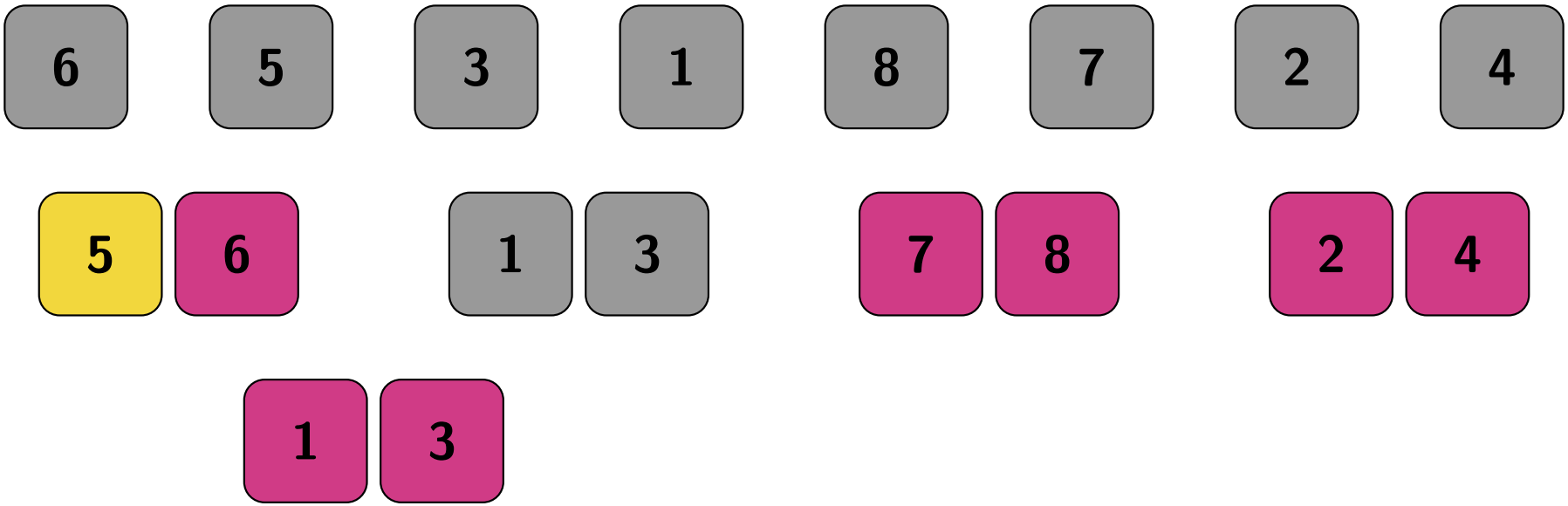
Merge Sort - Merge



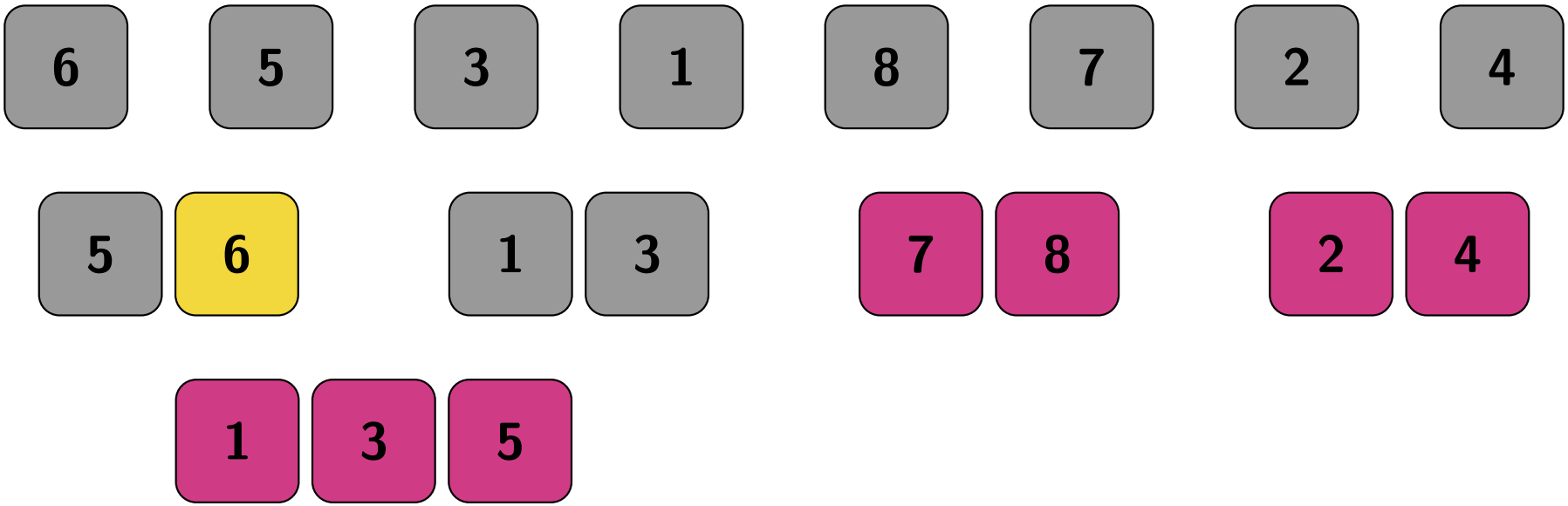
Merge Sort - Merge



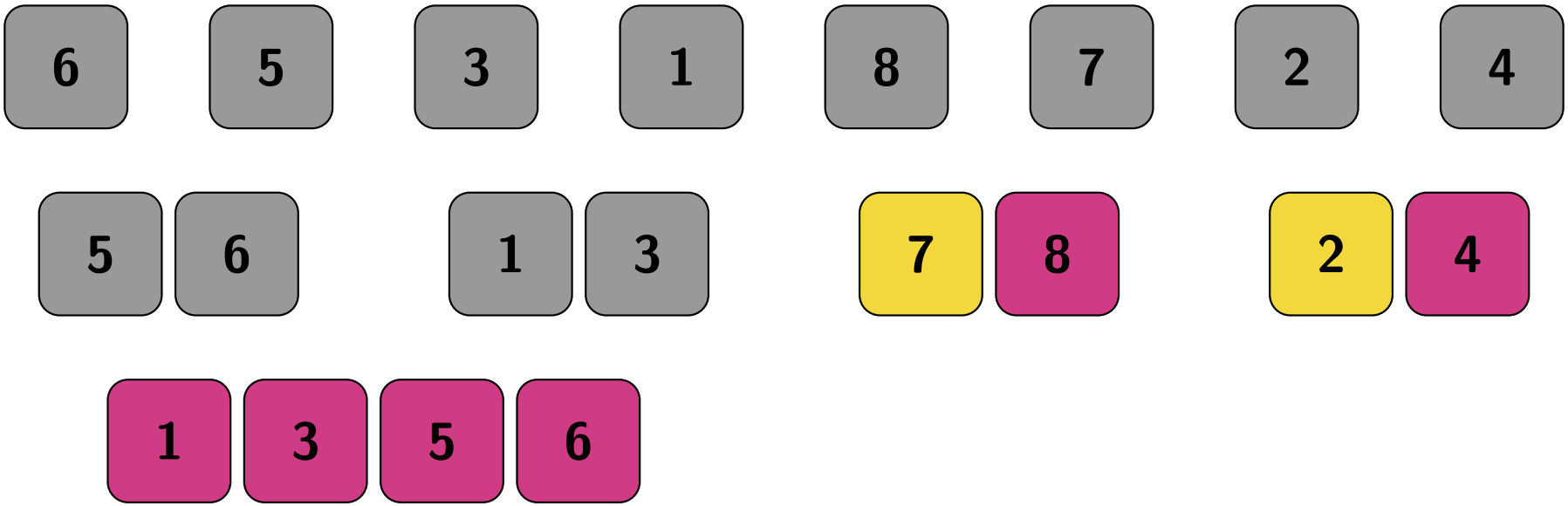
Merge Sort - Merge



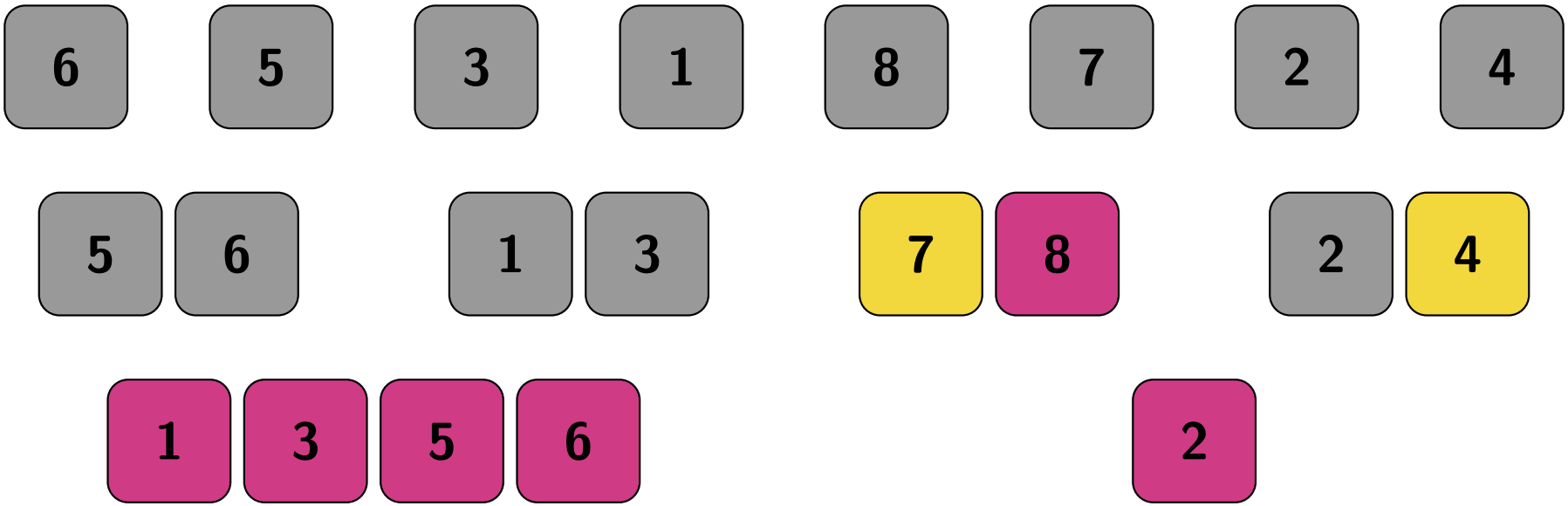
Merge Sort - Merge



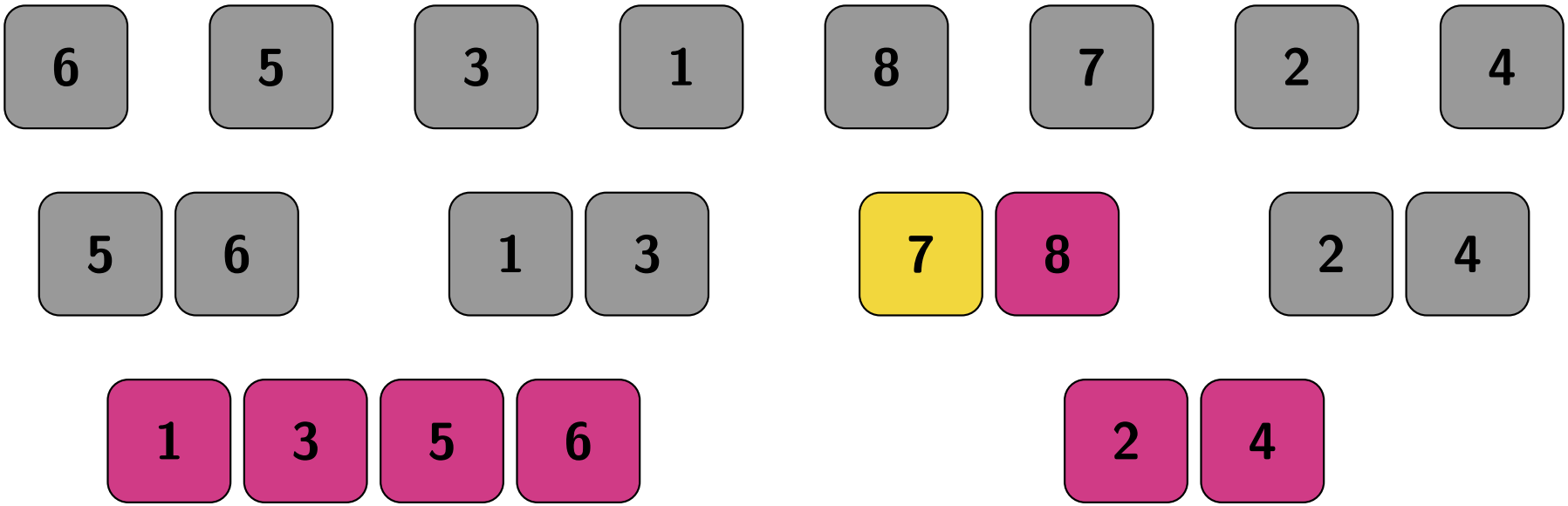
Merge Sort - Merge



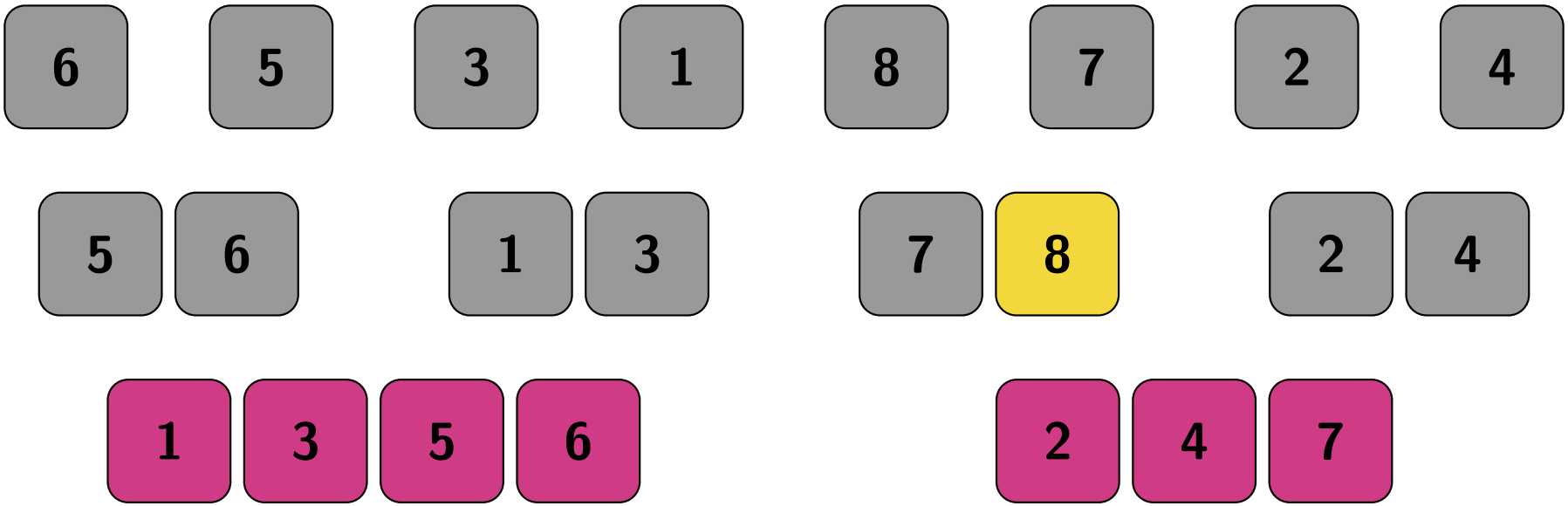
Merge Sort - Merge



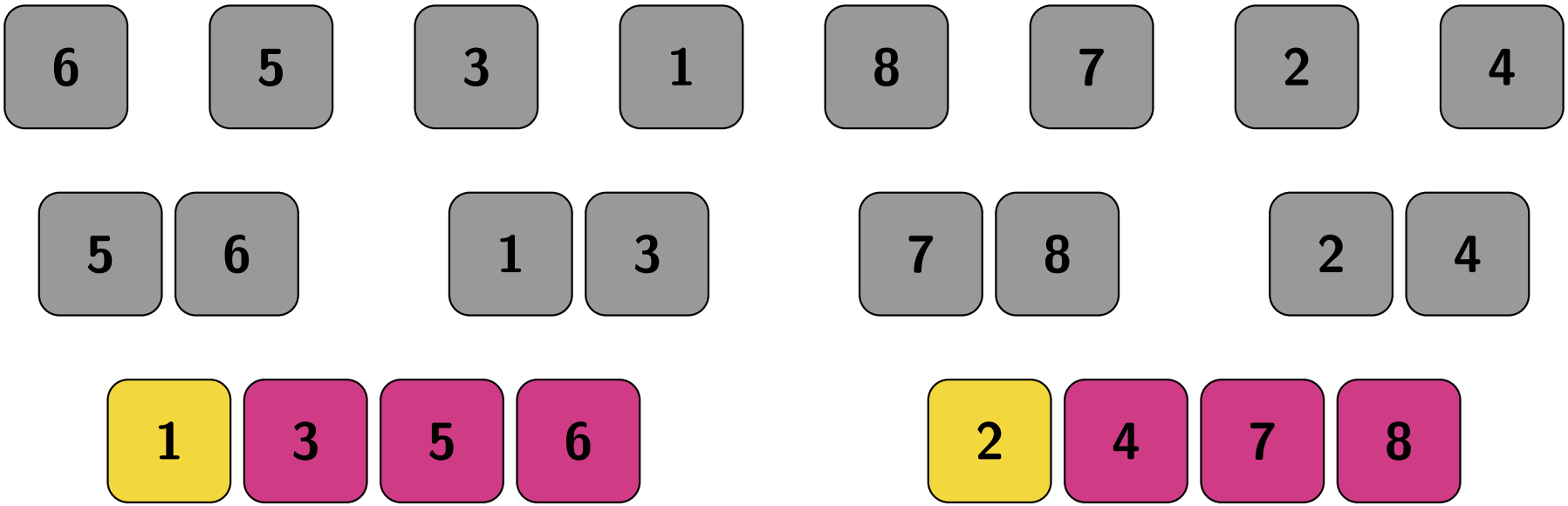
Merge Sort - Merge



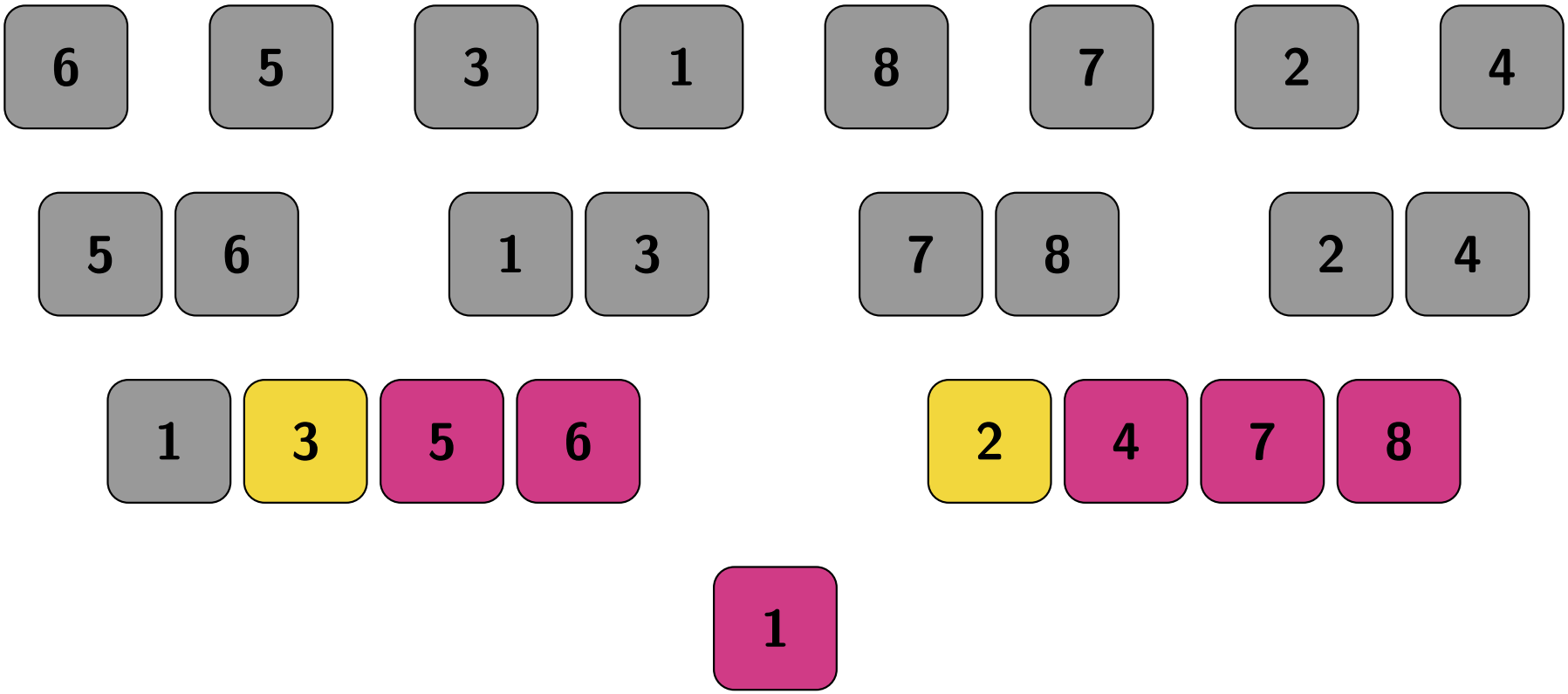
Merge Sort - Merge



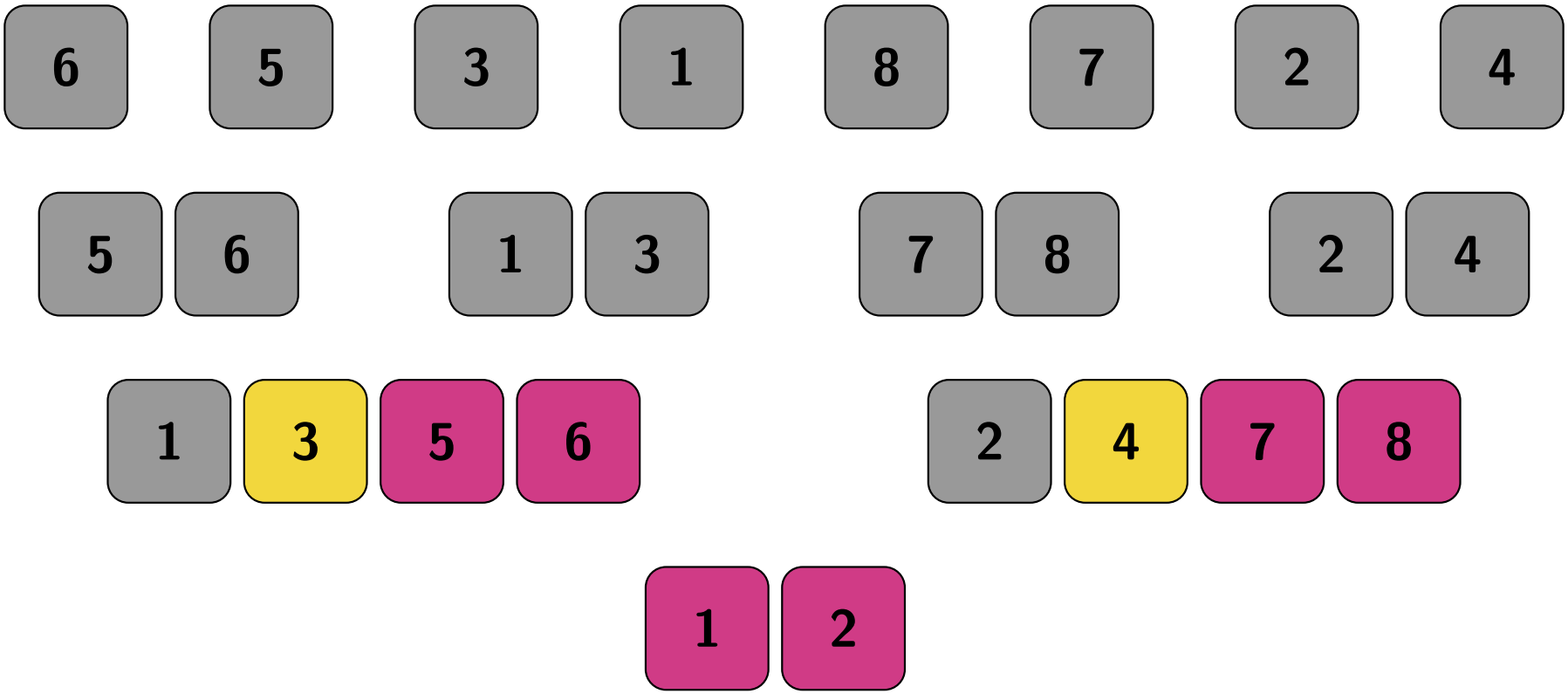
Merge Sort - Merge



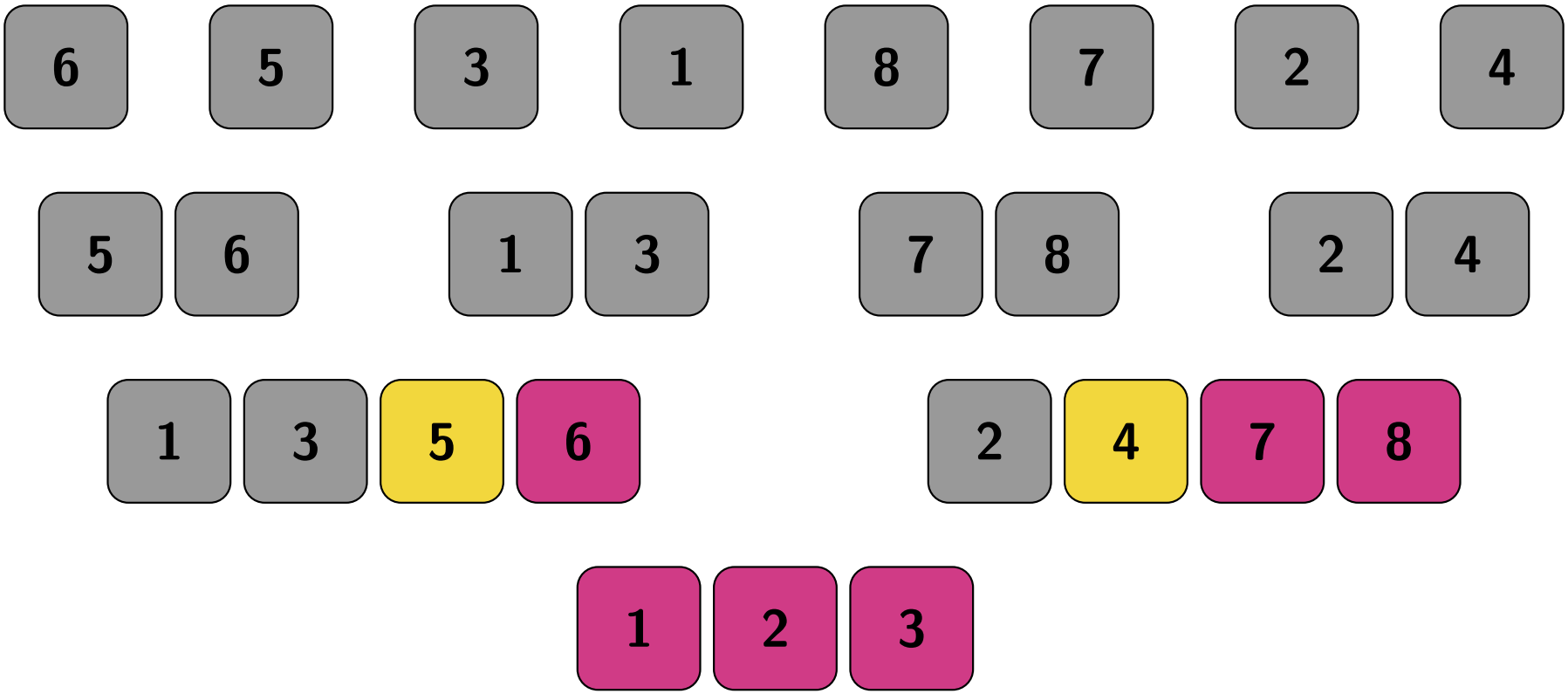
Merge Sort - Merge



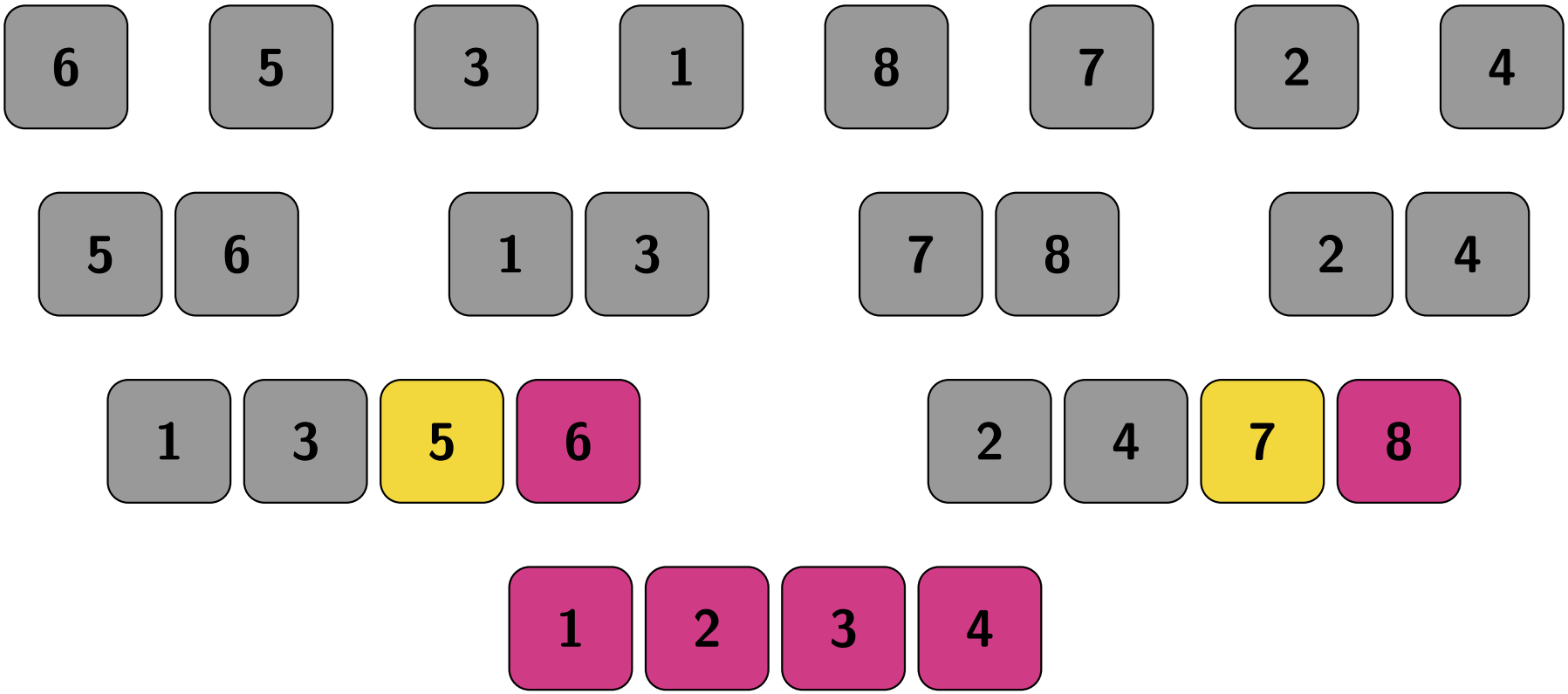
Merge Sort - Merge



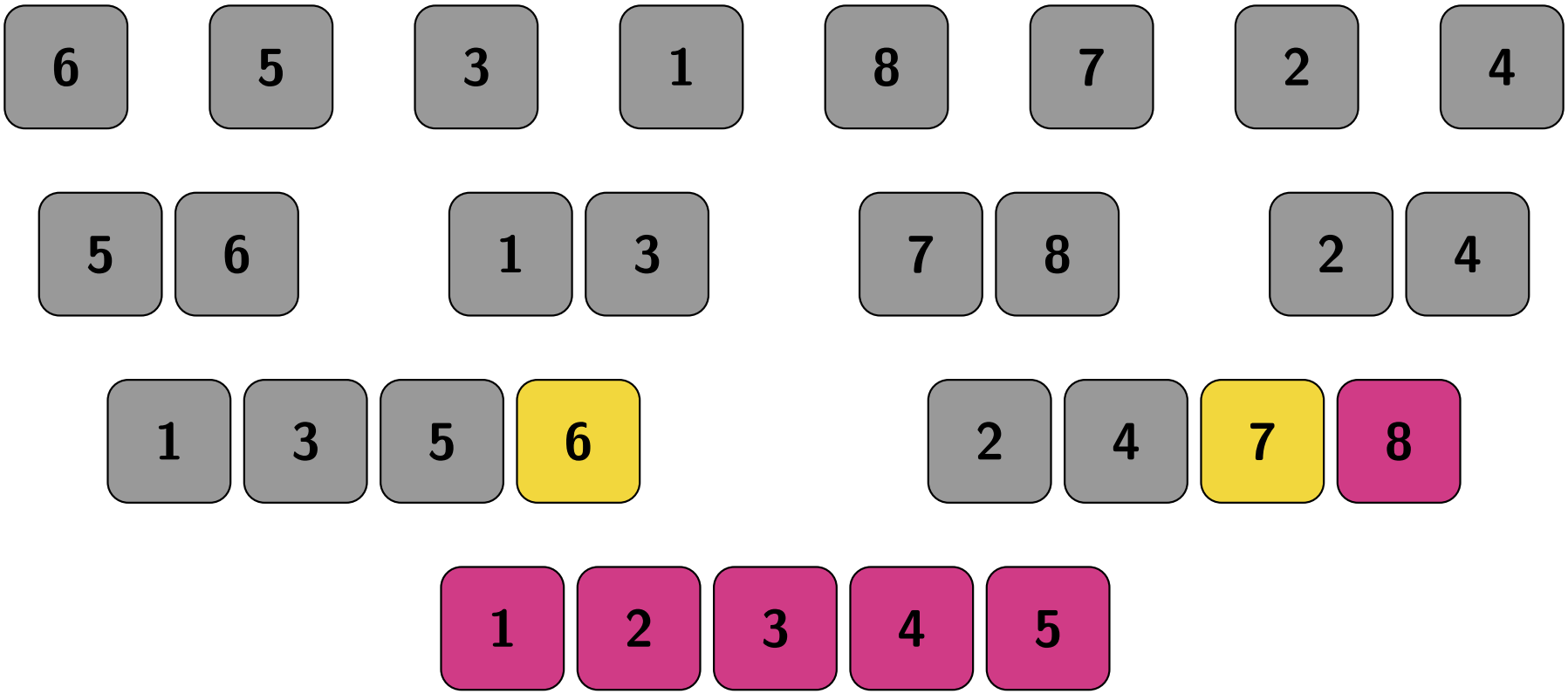
Merge Sort - Merge



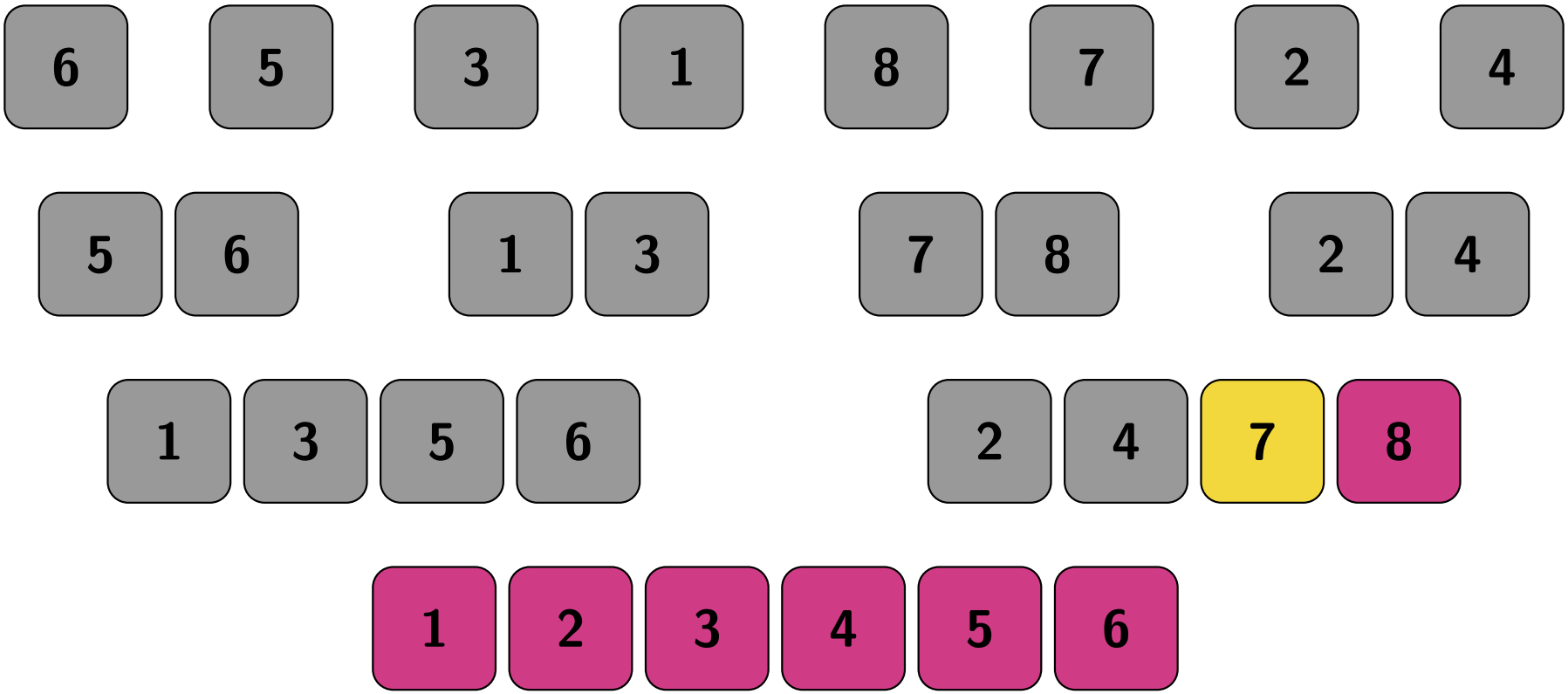
Merge Sort - Merge



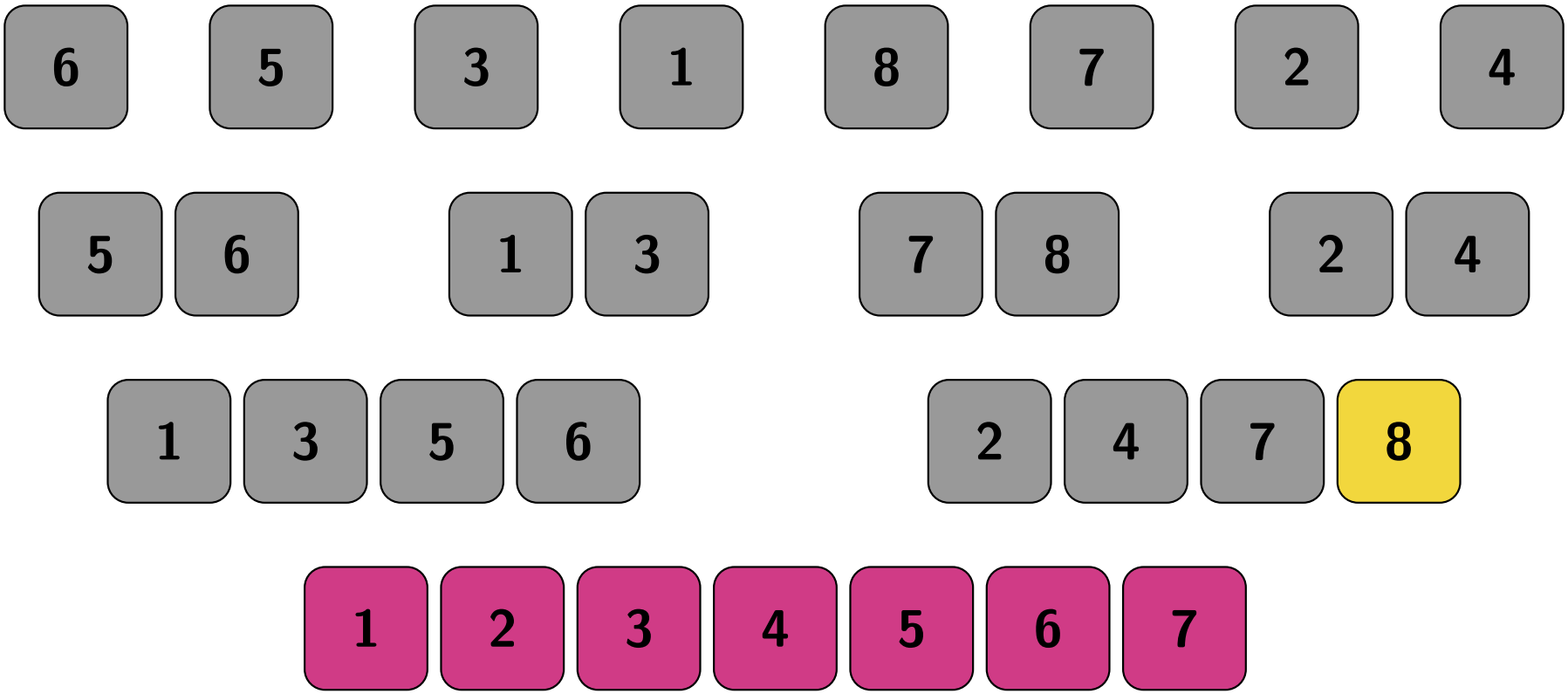
Merge Sort - Merge



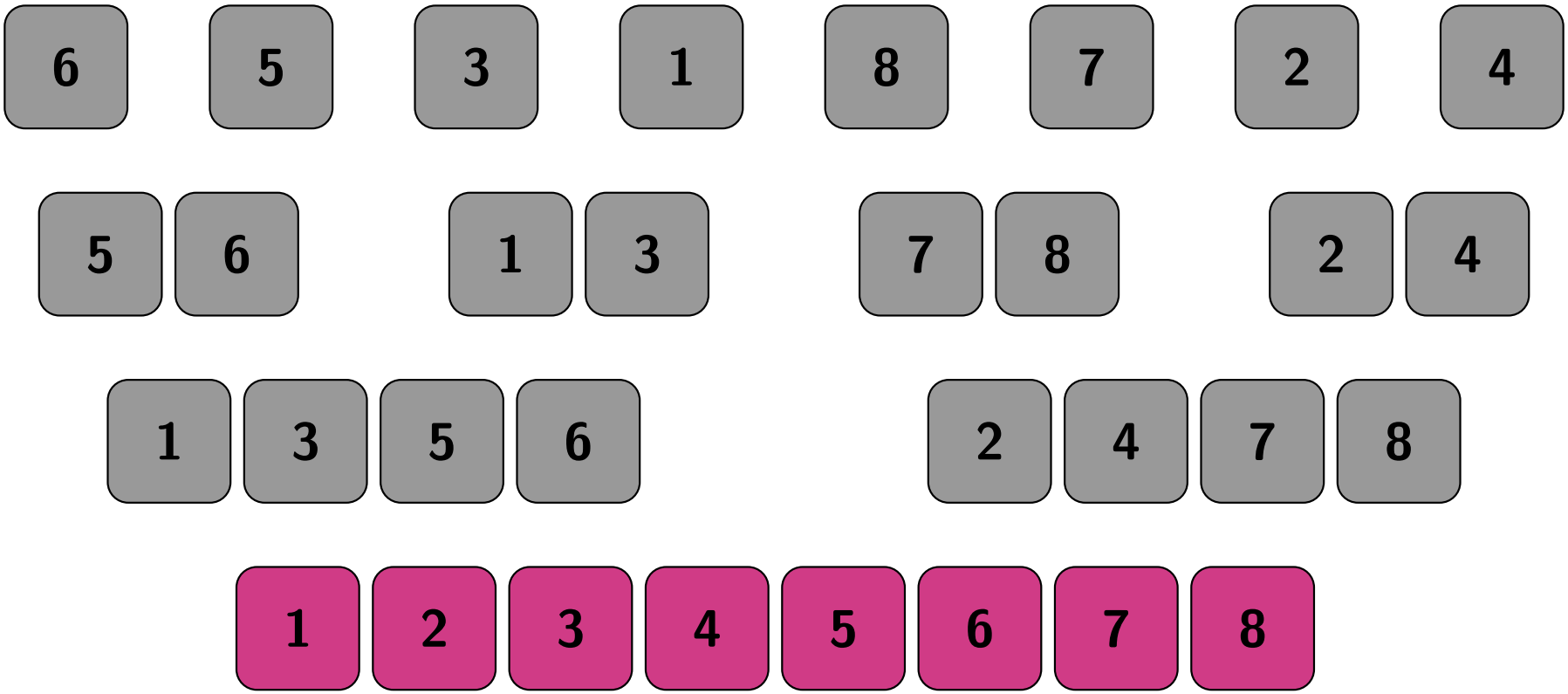
Merge Sort - Merge



Merge Sort - Merge



Merge Sort - Merge



Mulțimi

O mulțime este o colecție sortată ce nu conține duplicate.

- Definiția structurii de date este similară cu cea folosită anterior pentru o colecție.

```
1  typedef int T;
2
3  typedef struct set {
4      T *elements;
5      long size;
6      long capacity;
7  } *TSet;
```

Trebuie modificată implementarea pentru *add* astfel încât:

1. să nu adăugăm elemente duplicate;
2. să păstrăm ordinea elementelor după inserare.

Definirea TAD-ului pentru o mulțime

Constructori

- Aceștia au ca rezultat o mulțime nouă cu elemente de tip T.
- Considerăm ca nume pentru **TAD** $TSet$.
  1. Inițializarea colecției – $init :\rightarrow TSet$
  2. Adăugarea unui element – $add : TSet \times T \rightarrow TSet$
     - Ne asigurăm că păstrăm mulțime sortată și fără duplicate.
  3. Eliminarea unui element – $delete : TSet \times T \rightarrow TSet$
  4. Reuniunea – $union : TSet \times TSet \rightarrow TSet$
  5. Intersecția – $intersection : TSet \times TSet \rightarrow TSet$
  6. Diferență – $difference : TSet \times TSet \rightarrow TSet$

Funcții

- Operații care furnizează informații despre o mulțime.
  1. $empty : TSet \rightarrow \{0, 1\}$
  2. $size : TSet \rightarrow Int$
  3. $contains : TSet \times T \rightarrow \{0, 1\}$
     - Putem folosi algoritmul de căutare binară.

Adăugarea unui element într-o mulțime

```
1  TSet add(TSet set, T element) {
2      T *vect;
3      if (contains(set, element))
4          return set;
5      if (set->size == set->capacity) {
6          set->capacity *= 2;
7          vect = realloc(set->elements, set->capacity *
   ↪  sizeof(T));
8          if (vect == NULL)
9              return set;
10         set->elements = vect;
11     }
12     set->elements[set->size] = element;
13     set->size++;
14     qsort(set->elements, set->size, sizeof(T), cmp);
15     return set;
16 }
```

Adăugarea unui element într-o mulțime I

Varianta Eficientă

```
1  TSet add(TSet set, T element) {
2      T *vect;
3      int i;
4      if (contains(set, element))
5          return set;
6      if (set->size == set->capacity) {
7          set->capacity *= 2;
8          vect = realloc(set->elements, set->capacity *
   ↪  sizeof(T));
9          if (vect == NULL)
10             return set;
11         set->elements = vect;
12     }
```

Adăugarea unui element într-o mulțime II

```
13      i = set->size - 1;
14      for (; (i >= 0 && set->elements[i] > element); i--) {
15          set->elements[i + 1] = set->elements[i];
16      }
17      set->elements[i + 1] = element;
18      set->size++;
19      return set;
20  }
```

Complexitatea

- Complexitatea pentru *contains* – $O(\log_2 N)$
- Complexitatea pentru realocarea memoriei – $O(N)$
- Complexitatea pentru adăugarea pe poziția corespunzătoare – $O(N)$

$$T(N) = O(\log_2 N) + O(N) + O(N) = O(N)$$

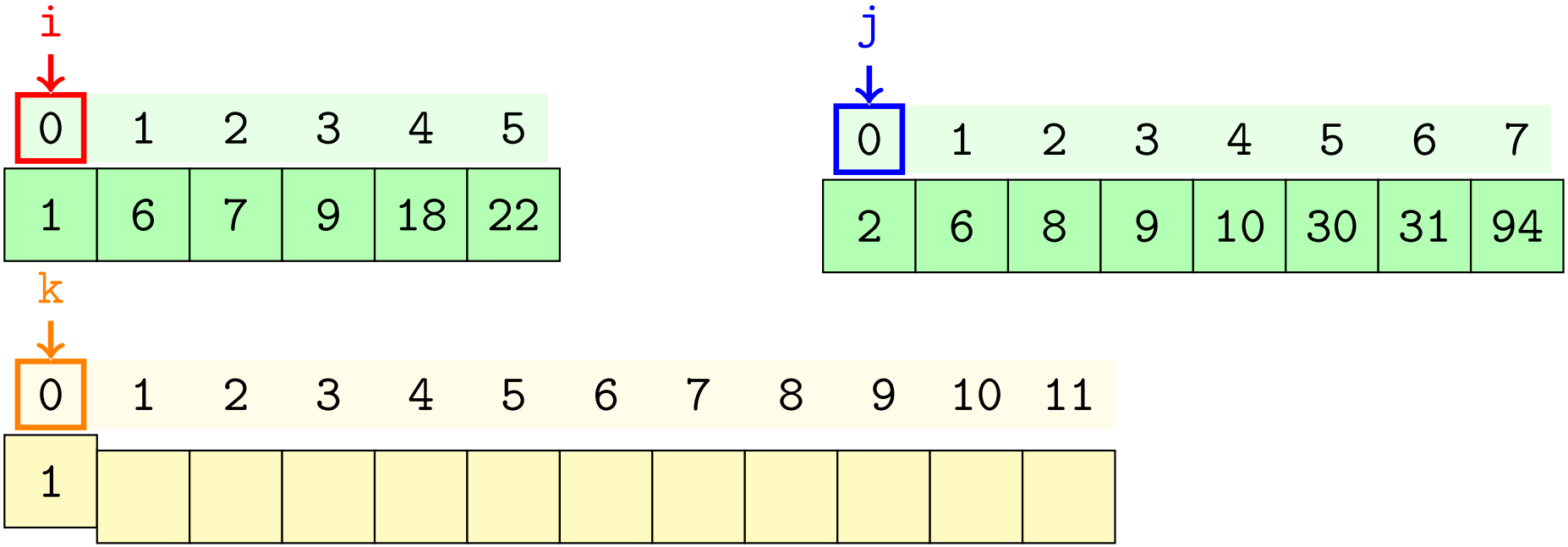
Reuniunea a două mulțimi I

```
1  TSet union(TSet set1, TSet set2) {
2      TSet result = malloc(sizeof(struct set));
3      result->size = 0;
4      result->capacity = set1->size + set2->size;
5      result->elements = malloc(result->capacity * sizeof(T));
6      int i = 0, j = 0, k = 0;
7      while (i < set1->size && j < set2->size) {
8          if (set1->elements[i] < set2->elements[j]) {
9              result->elements[k++] = set1->elements[i];
10             i++;
11         } else if (set1->elements[i] > set2->elements[j]) {
12             result->elements[k++] = set2->elements[j];
13             j++;
14         }
```

Reuniunea a două mulțimi II

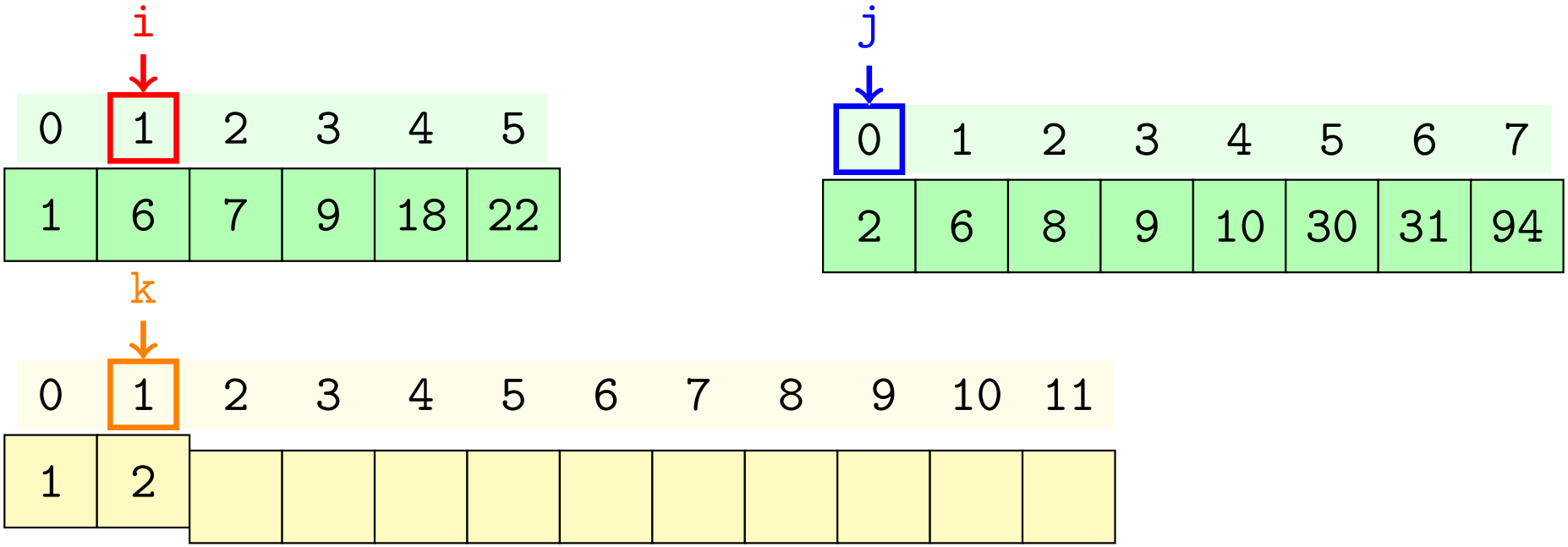
```
15          else {
16              result->elements[k++] = set1->elements[i];
17              i++;
18              j++;
19          }
20      }
21      while (i < set1->size)
22          result->elements[k++] = set1->elements[i++];
23      while (j < set2->size)
24          result->elements[k++] = set2->elements[j++];
25      result->size = k;
26      return result;
27  }
```

Reuniunea a două mulțimi – Exemplu



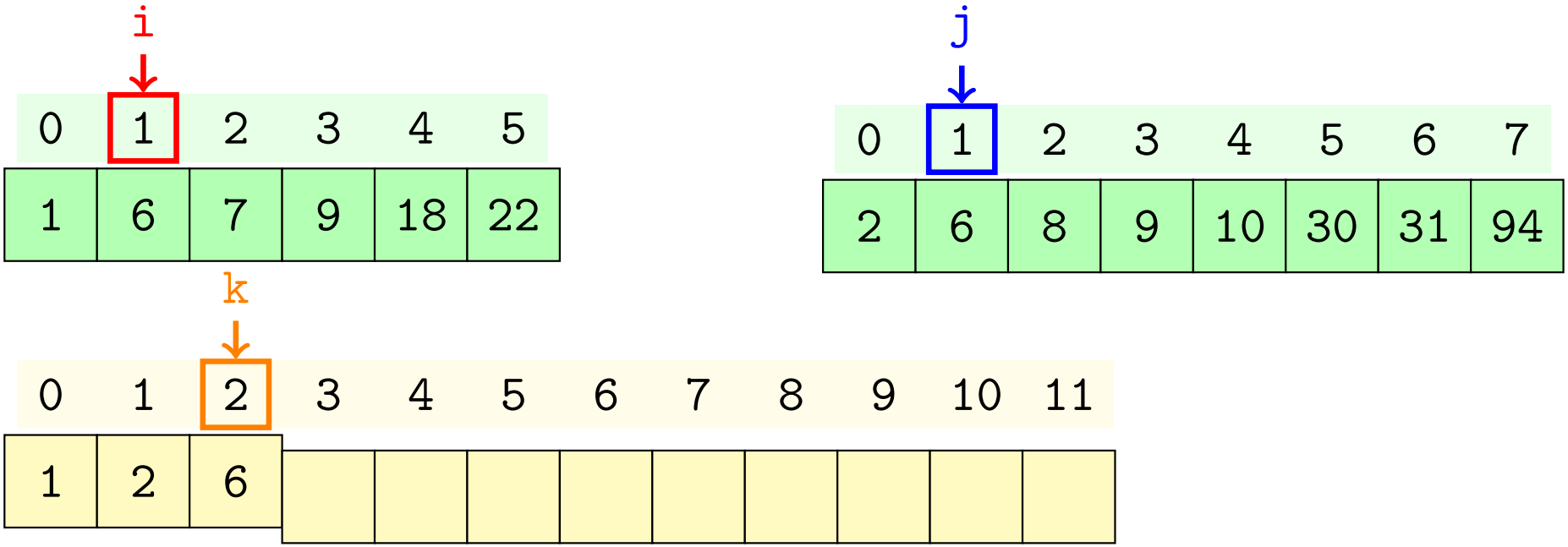

```
7   while (i < set1->size && j < set2->size) {
8       if (set1->elements[i] < set2->elements[j]) {
9           result->elements[k++] = set1->elements[i];
10          i++;
11      } else if (set1->elements[i] > set2->elements[j]) {
12          result->elements[k++] = set2->elements[j];
13          j++;
14      }
15      else {
16          result->elements[k++] = set1->elements[i];
17          i++;
18          j++;
19      }
20  }
```

Reuniunea a două mulțimi – Exemplu



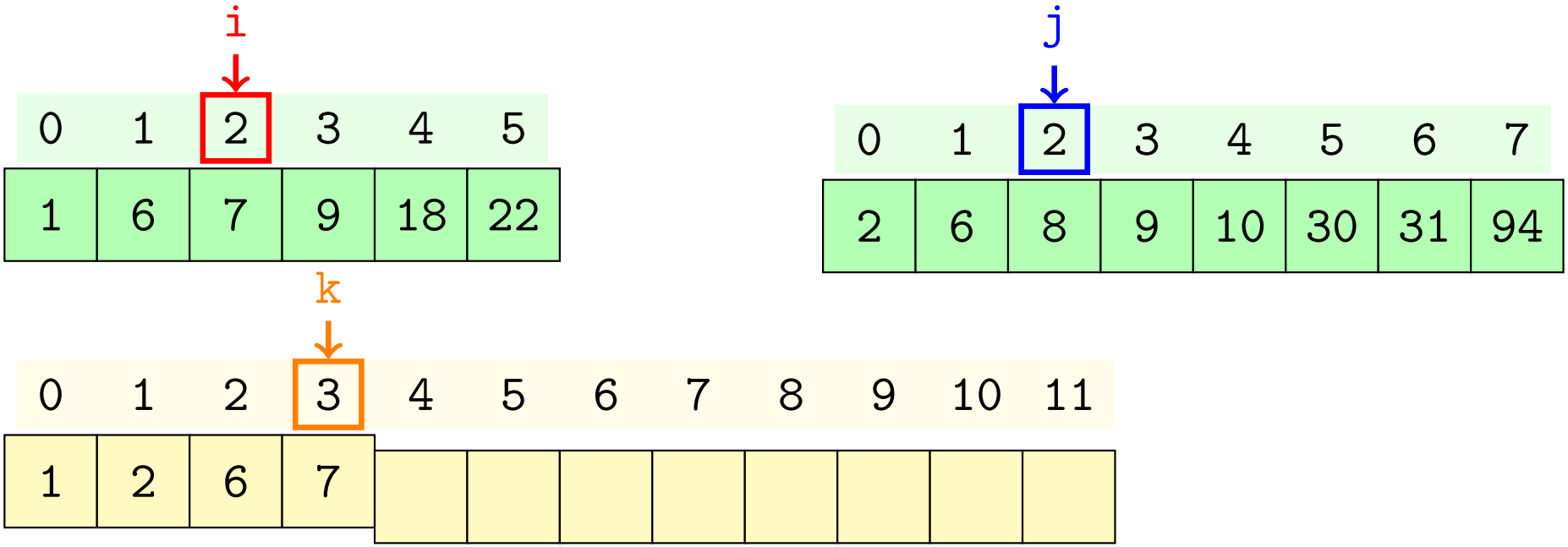

```
 7   while (i < set1->size && j < set2->size) {
 8       if (set1->elements[i] < set2->elements[j]) {
 9           result->elements[k++] = set1->elements[i];
10           i++;
11       } else if (set1->elements[i] > set2->elements[j]) {
12           result->elements[k++] = set2->elements[j];
13           j++;
14       }
15       else {
16           result->elements[k++] = set1->elements[i];
17           i++;
18           j++;
19       }
20   }
```

Reuniunea a două mulțimi – Exemplu



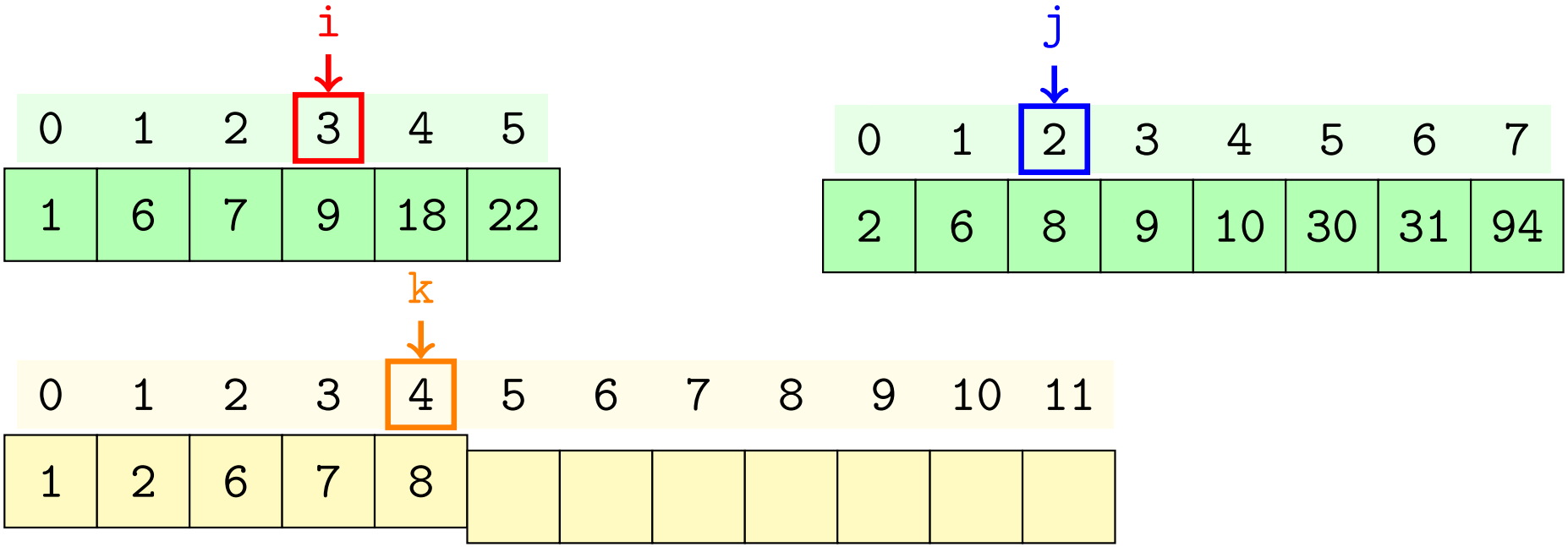

```
 7   while (i < set1->size && j < set2->size) {
 8       if (set1->elements[i] < set2->elements[j]) {
 9           result->elements[k++] = set1->elements[i];
10           i++;
11       } else if (set1->elements[i] > set2->elements[j]) {
12           result->elements[k++] = set2->elements[j];
13           j++;
14       }
15       else {
16           result->elements[k++] = set1->elements[i];
17           i++;
18           j++;
19       }
20   }
```

Reuniunea a două mulțimi – Exemplu



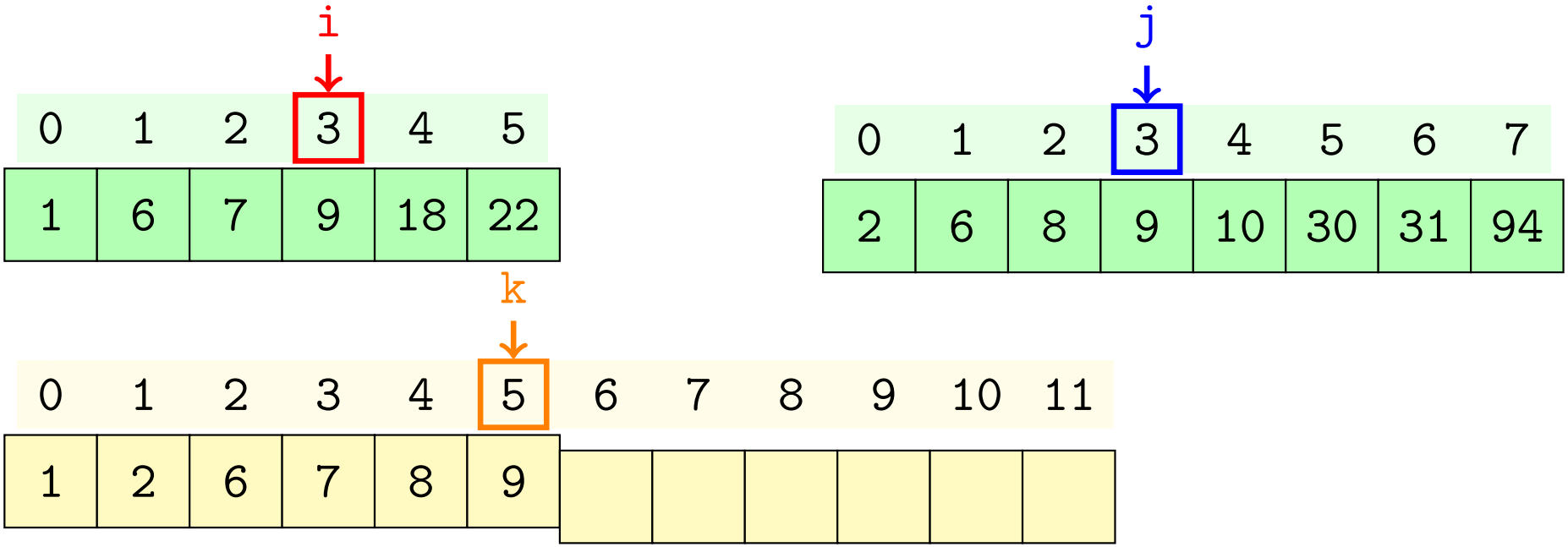

```
7   while (i < set1->size && j < set2->size) {
8       if (set1->elements[i] < set2->elements[j]) {
9           result->elements[k++] = set1->elements[i];
10          i++;
11      } else if (set1->elements[i] > set2->elements[j]) {
12          result->elements[k++] = set2->elements[j];
13          j++;
14      }
15      else {
16          result->elements[k++] = set1->elements[i];
17          i++;
18          j++;
19      }
20  }
```

Reuniunea a două mulțimi – Exemplu



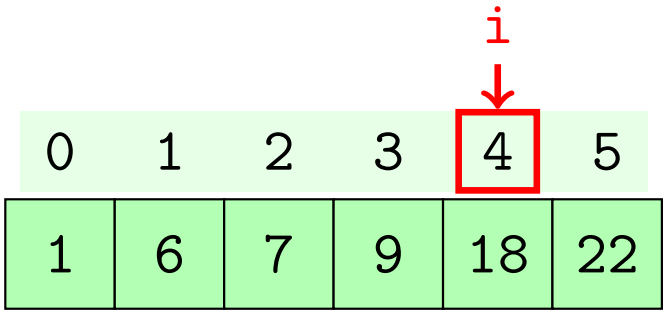

```
 7   while (i < set1->size && j < set2->size) {
 8       if (set1->elements[i] < set2->elements[j]) {
 9           result->elements[k++] = set1->elements[i];
10           i++;
11       } else if (set1->elements[i] > set2->elements[j]) {
12           result->elements[k++] = set2->elements[j];
13           j++;
14       }
15       else {
16           result->elements[k++] = set1->elements[i];
17           i++;
18           j++;
19       }
20   }
```

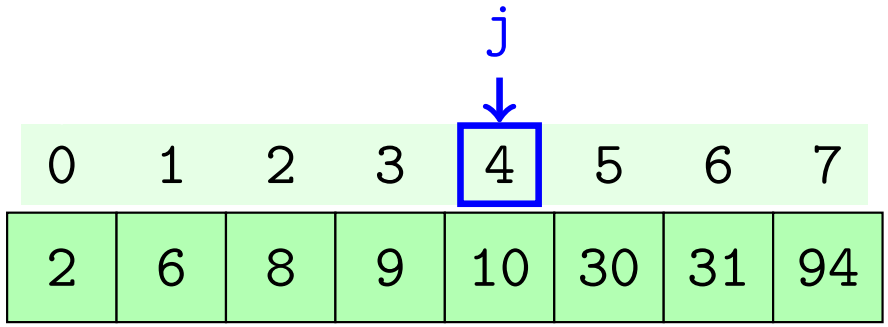
Reuniunea a două mulțimi – Exemplu

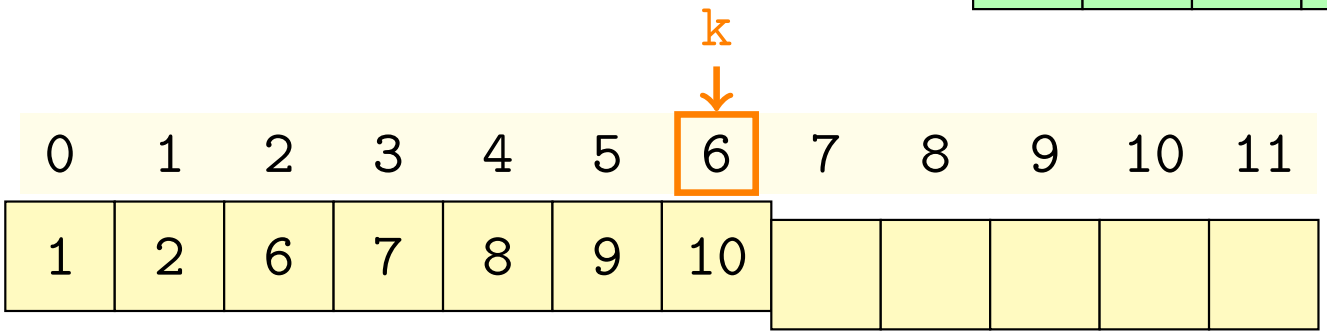



```
 7   while (i < set1->size && j < set2->size) {
 8       if (set1->elements[i] < set2->elements[j]) {
 9           result->elements[k++] = set1->elements[i];
10           i++;
11       } else if (set1->elements[i] > set2->elements[j]) {
12           result->elements[k++] = set2->elements[j];
13           j++;
14       }
15       else {
16           result->elements[k++] = set1->elements[i];
17           i++;
18           j++;
19       }
20   }
```

Reuniunea a două mulțimi – Exemplu

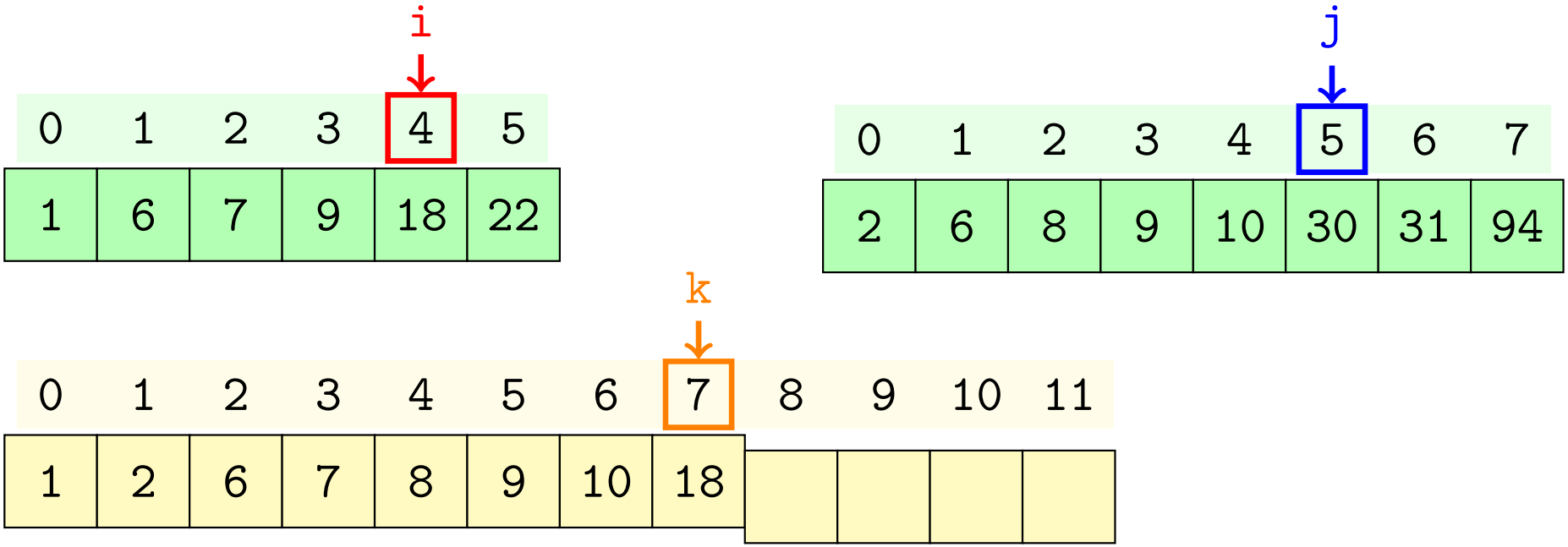






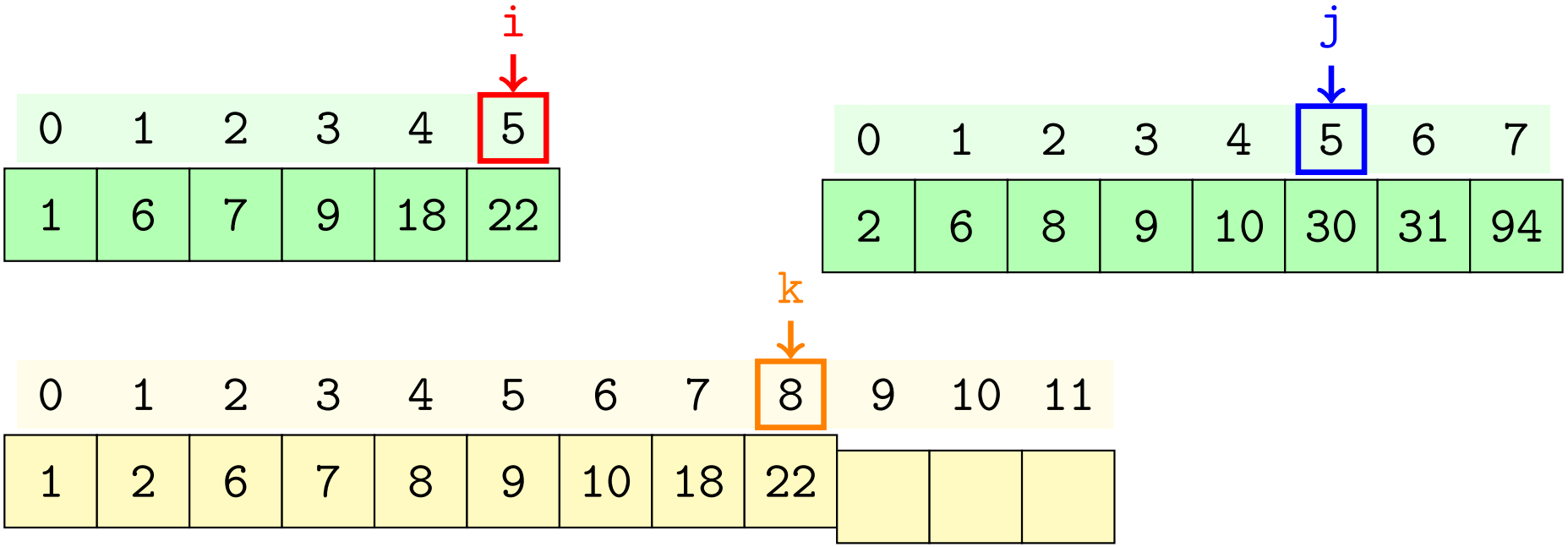

```
 7   while (i < set1->size && j < set2->size) {
 8       if (set1->elements[i] < set2->elements[j]) {
 9           result->elements[k++] = set1->elements[i];
10           i++;
11       } else if (set1->elements[i] > set2->elements[j]) {
12           result->elements[k++] = set2->elements[j];
13           j++;
14       }
15       else {
16           result->elements[k++] = set1->elements[i];
17           i++;
18           j++;
19       }
20   }
```

Reuniunea a două mulțimi – Exemplu



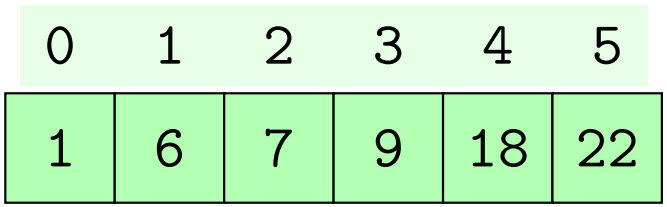

```
7   while (i < set1->size && j < set2->size) {
8       if (set1->elements[i] < set2->elements[j]) {
9           result->elements[k++] = set1->elements[i];
10          i++;
11      } else if (set1->elements[i] > set2->elements[j]) {
12          result->elements[k++] = set2->elements[j];
13          j++;
14      }
15      else {
16          result->elements[k++] = set1->elements[i];
17          i++;
18          j++;
19      }
20  }
```

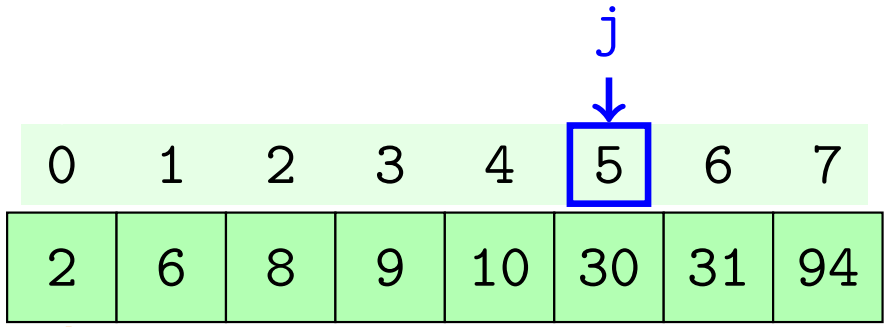
Reuniunea a două mulțimi – Exemplu

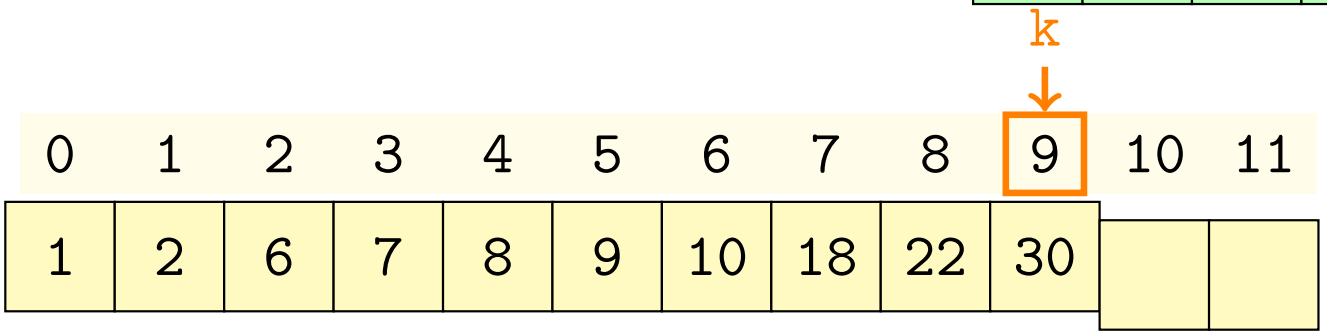



```
7   while (i < set1->size && j < set2->size) {
8       if (set1->elements[i] < set2->elements[j]) {
9           result->elements[k++] = set1->elements[i];
10          i++;
11      } else if (set1->elements[i] > set2->elements[j]) {
12          result->elements[k++] = set2->elements[j];
13          j++;
14      }
15      else {
16          result->elements[k++] = set1->elements[i];
17          i++;
18          j++;
19      }
20  }
```

Reuniunea a două mulțimi – Exemplu

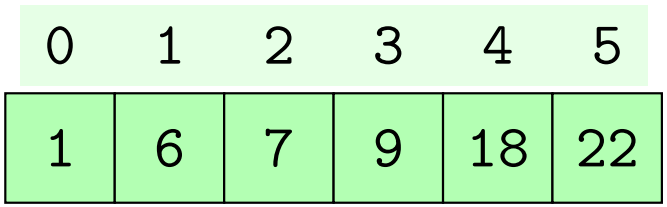


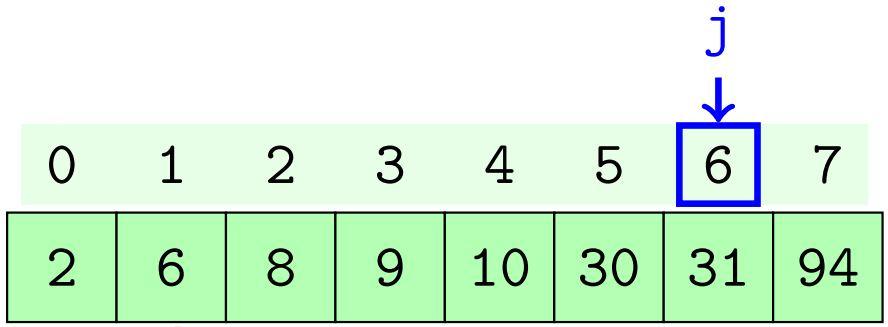


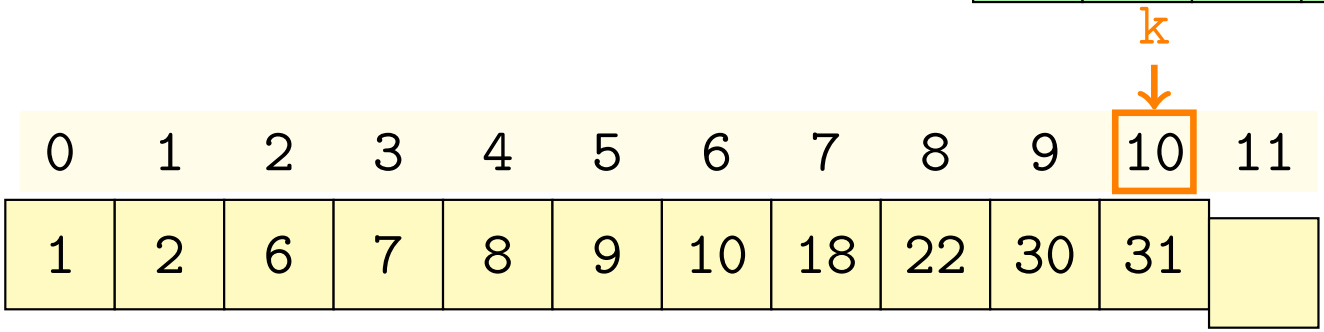



```
23    while (j < set2->size)
24        result->elements[k++] = set2->elements[j++];
```

Reuniunea a două mulțimi – Exemplu

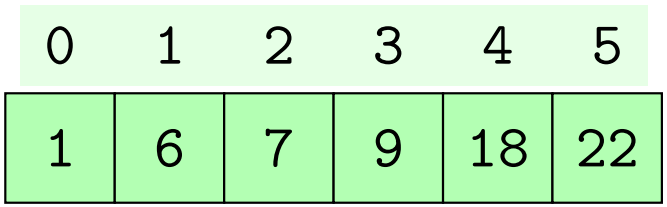


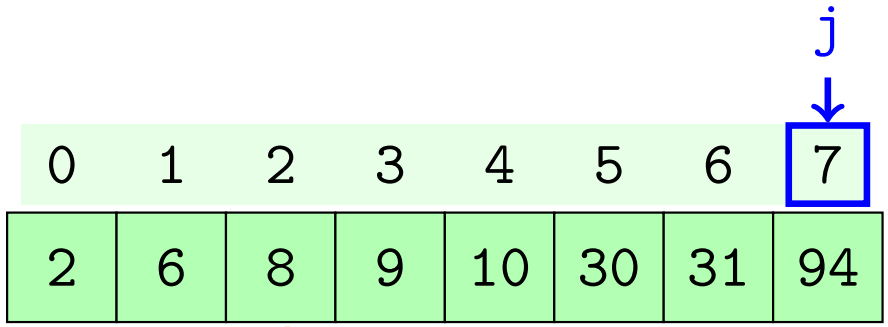


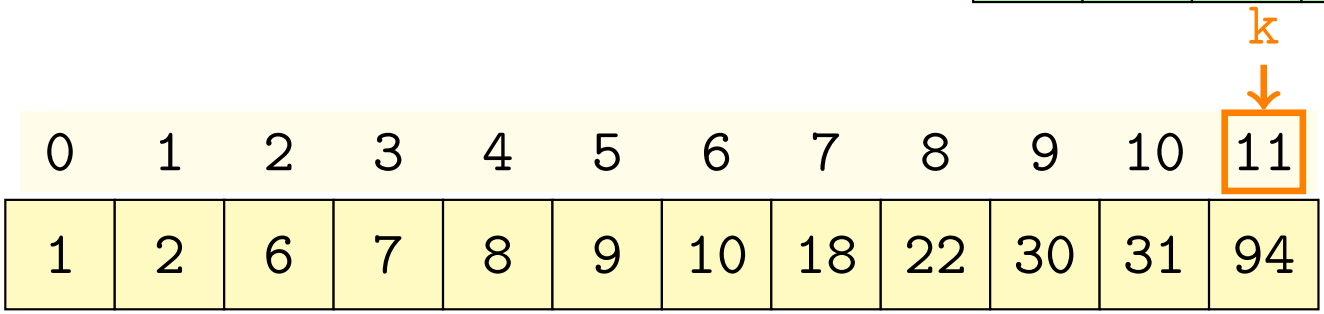



```
23    while (j < set2->size)
24        result->elements[k++] = set2->elements[j++];
```

Reuniunea a două mulțimi – Exemplu








```
23    while (j < set2->size)
24        result->elements[k++] = set2->elements[j++];
```

Vă mulțumesc pentru atenție!