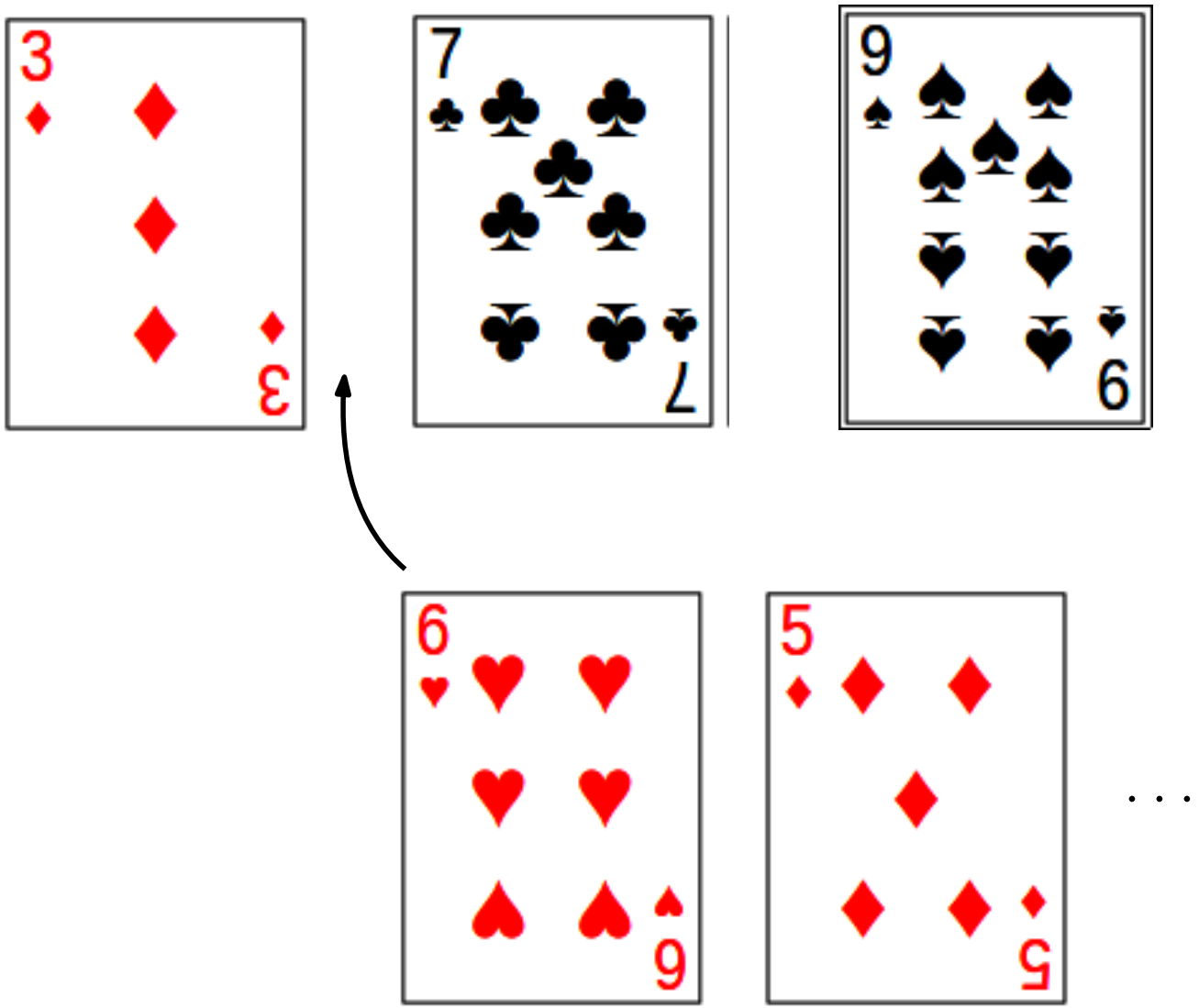
Sorting

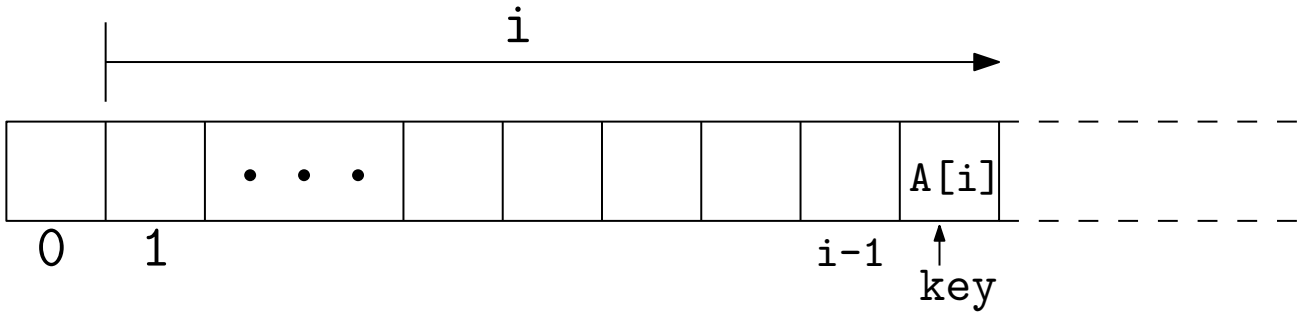
Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2020/index.html

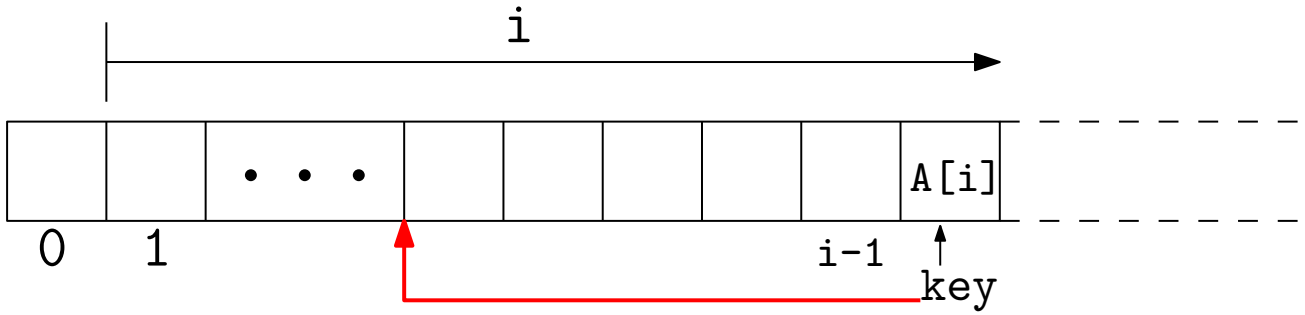
Insertion sort (from day 0)



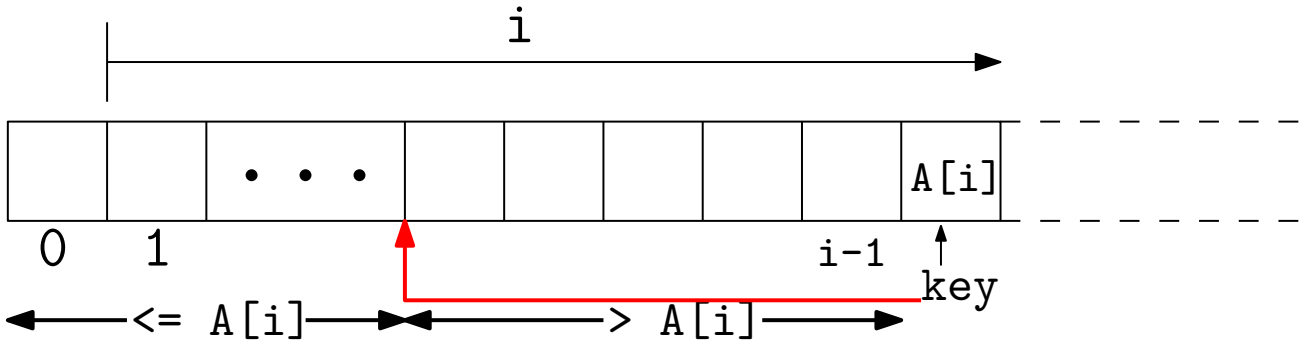
Insertion sort



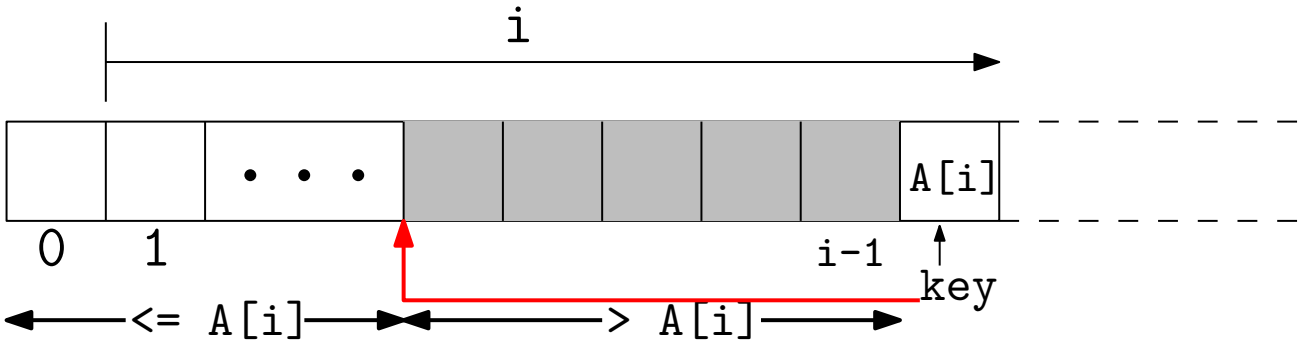
Insertion sort



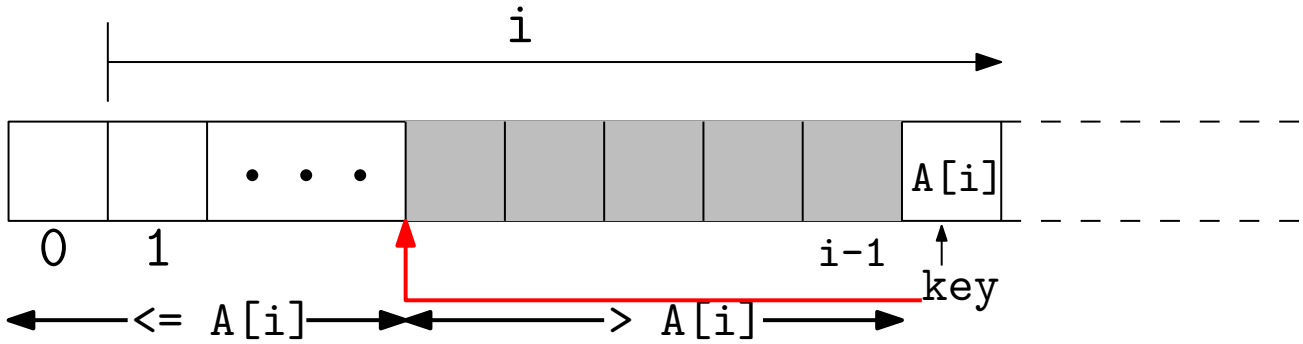
Insertion sort



Insertion sort

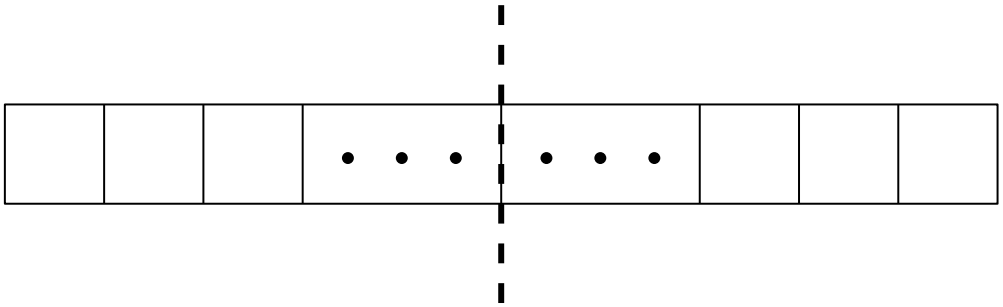


Insertion sort




```
for (i = 1; i < n; i++) {
    key = A[i];
    /* Find the right place for A[i] in A[0 .. i-1] */
    for (j = i-1; j >= 0 && A[j] > key; j--)
        A[j+1] = A[j];
    A[j+1] = key;
}
```

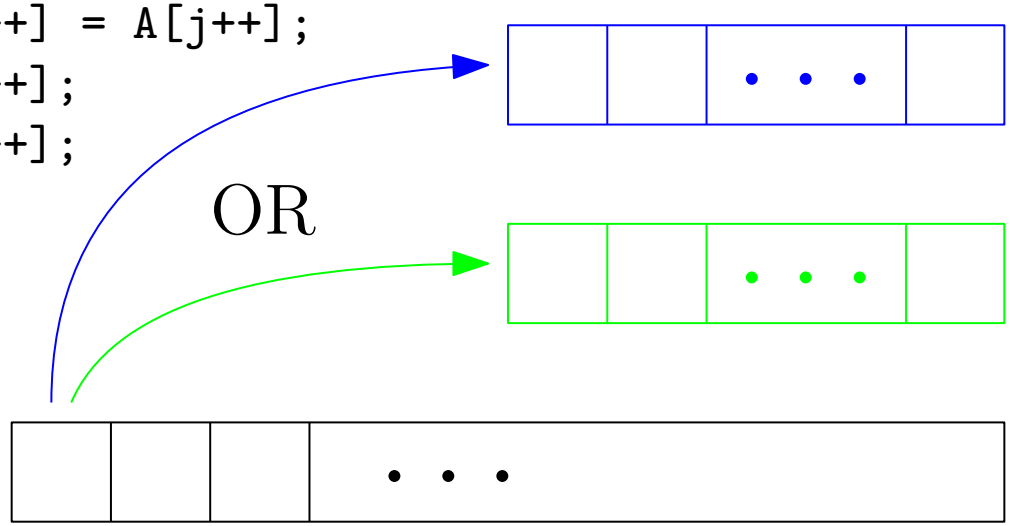
Merge sort



```
void msort(int *A, int beginning, int end) {
     if (beginning < end) {
        middle = (beginning + end) / 2;
        msort(A, beginning, middle);
        msort(A, middle+1, end);
        merge(A, beginning, middle, end);
     }
}
```

Merge sort

```
void merge(A, b, m, e) {
     int i, j, k;
     /* allocate space for auxiliary array B */
     for (i = b, j = m+1, k = 0; i <= m && j <= e; )
         if (A[i] < A[j])
            B[k++] = A[i++];
         else if (A[i] > A[j])
            B[k++] = A[j++];
         else
            B[k++] = A[i++], B[k++] = A[j++];
     while (i <= m) B[k++] = A[i++];
     while (j <= e) B[k++] = A[j++];
     assert(...);
}
```



Heap sort

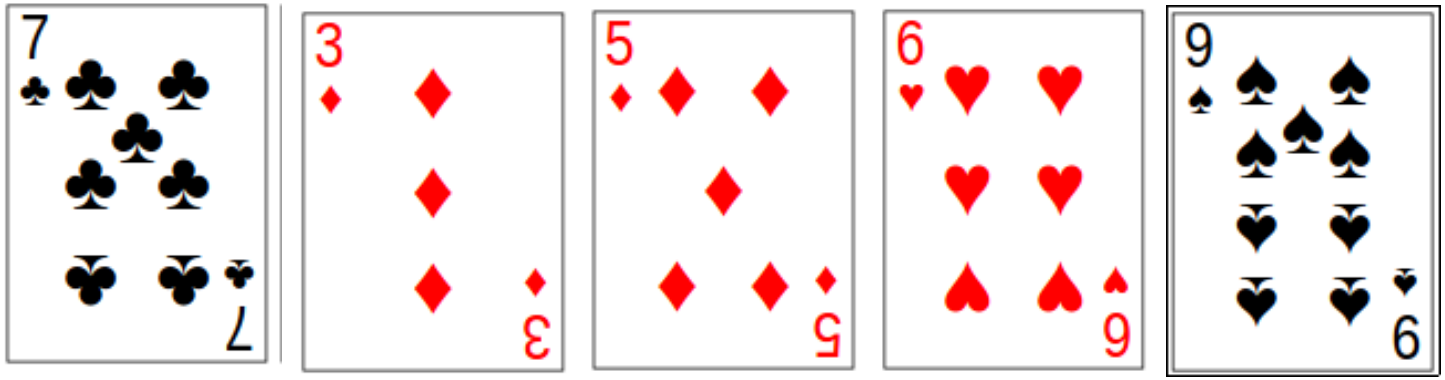
Reference: Sedgewick and Wayne, Section 2.4

NOTE: Indexing from 1!

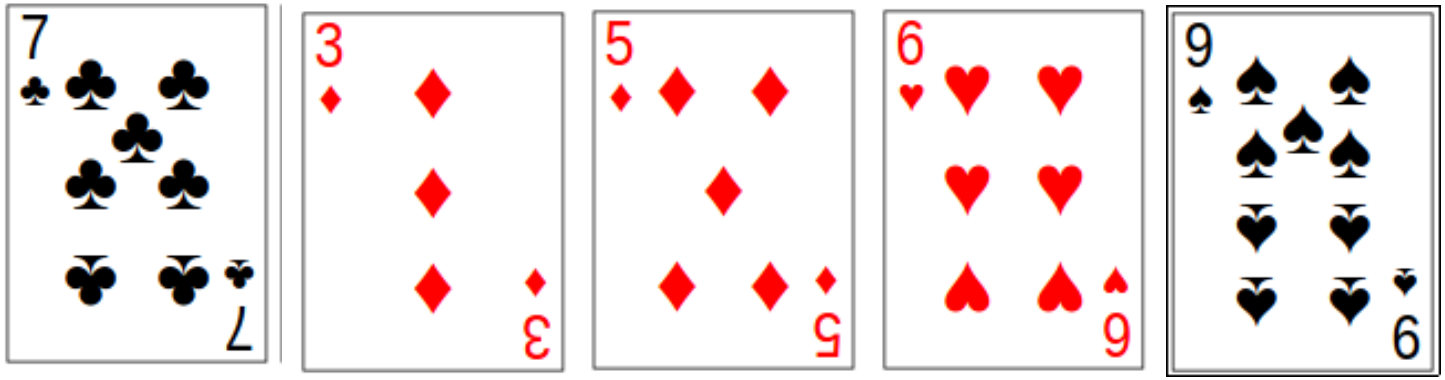
```
void heapsort(int *a, int N) {
    int tmp;
    INT_HEAP h;

    h.num_allocated = h.num_used = N;
    h.array = a;
    /* Make heap out of array */
    buildheap(&h);
    /* Sort by successive deleteMax */
    while (h.num_used > 1) {
        tmp = h.array[1],
            h.array[1] = h.array[h.num_used],
            h.array[h.num_used] = tmp; // move max to end
        h.num_used--;
        swapDown(&h, 1);
    }
```

```
for (k = h->num_used / 2; k >= 1; k--)
    swapDown(h, k);
```

Bucket sort (from day 0)

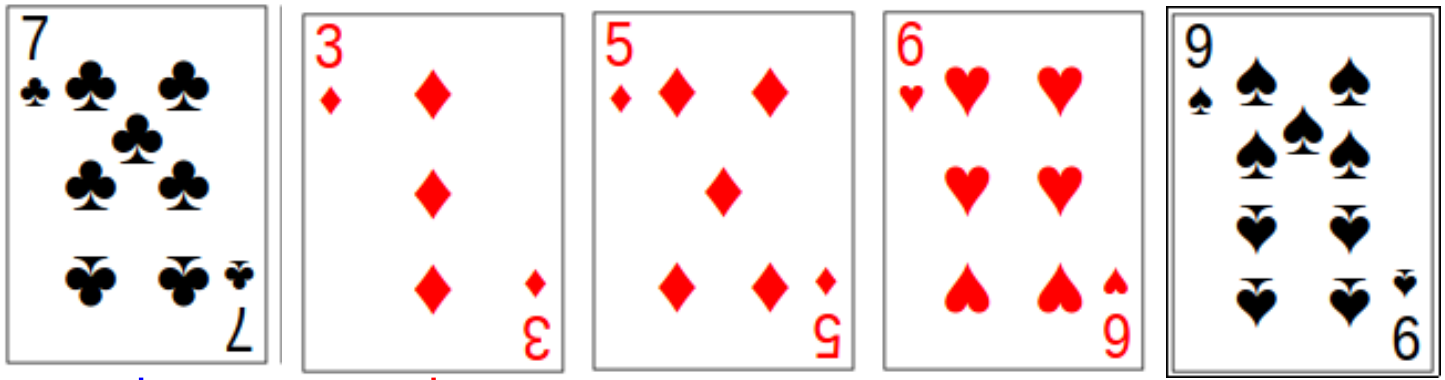


Bucket sort (from day 0)



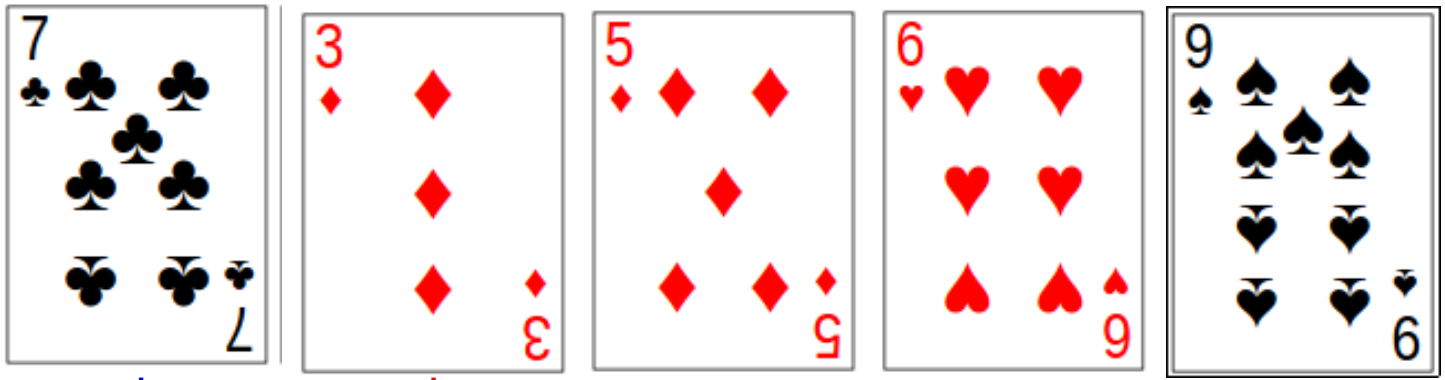
Spades				Hearts				Diamonds				Clubs			
2	3	...	**Ace**	2	3	...	**Ace**	2	3	...	**Ace**	2	3	...	**Ace**

Bucket sort (from day 0)



Spades				Hearts				Diamonds				Clubs			
2	3	...	**Ace**	2	3	...	**Ace**	2	3	...	**Ace**	2	3	...	**Ace**

Bucket sort (from day 0)



Spades				Hearts				Diamonds				Clubs			
2	3	. . .	**Ace**	2	3	. . .	**Ace**	2	3	. . .	**Ace**	2	3	. . .	**Ace**

Counting sort

Given:

- $A[0], A[1], \ldots, A[n-1]$
- All inputs are within a (relatively small) fixed range (say $0 \leq A[i] \leq k$)

Algorithm

1. Create an array `counts` with `k+1` elements, *initialised to all 0s*.
2. Count how many times each element has occurred in A, i.e., for each i in $0, 1, \ldots, n-1$, increment `counts[A[i]]`.
3. Compute cumulative frequencies (number of elements having value less than i for $0 \leq i \leq k$).

```
C[0] ← 0
C[1] ← C[0]
C[2] ← C[0] + C[1]
C[3] ← C[0] + C[1] + C[2] ...
```

Counting sort

Given:

- $A[0], A[1], \ldots, A[n-1]$
- All inputs are within a (relatively small) fixed range (say $0 \leq A[i] \leq k$)

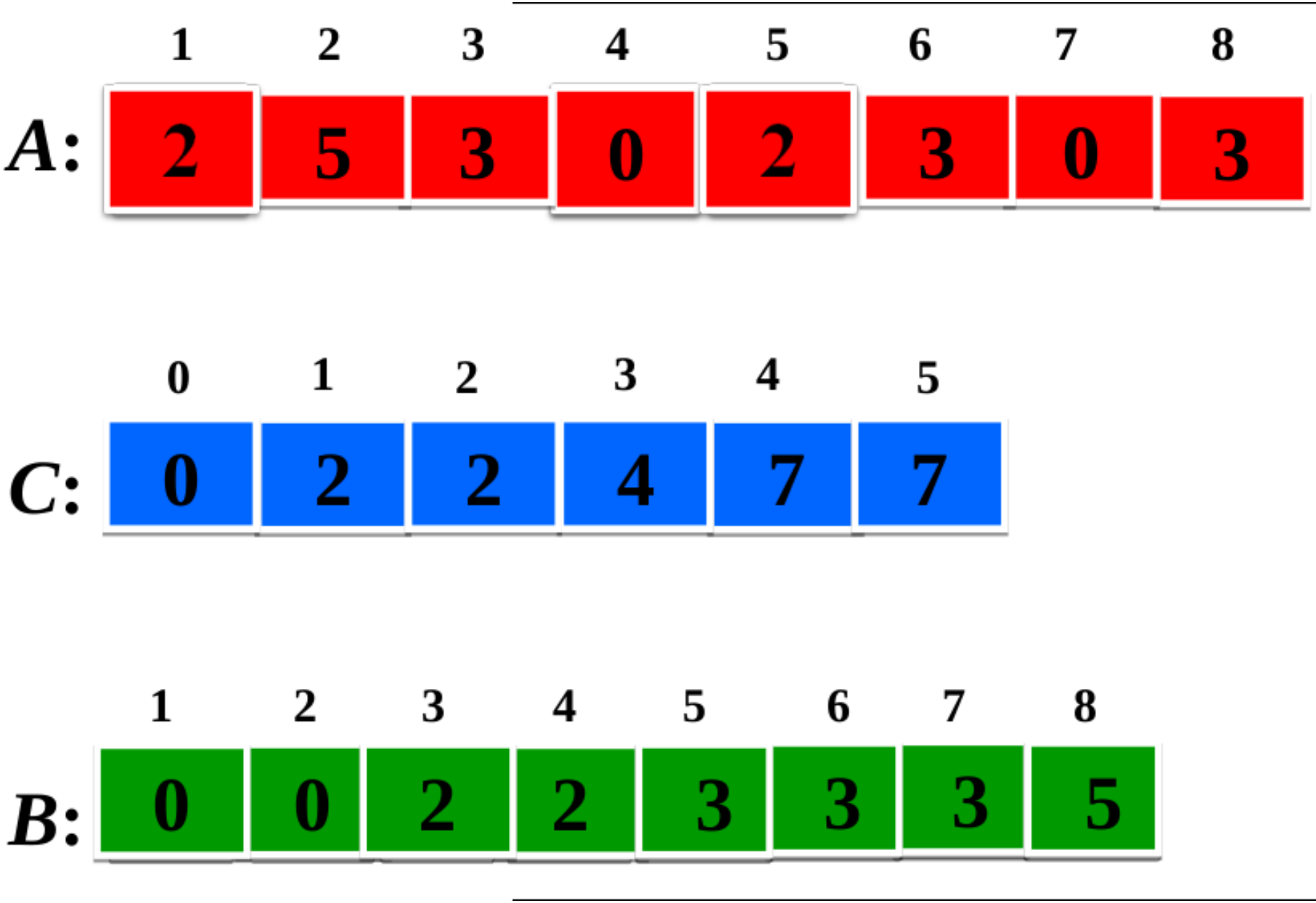
Algorithm

1. Create an array `counts` with `k+1` elements, *initialised to all 0s*.
2. Count how many times each element has occurred in A, i.e., for each i in $0, 1, \ldots, n-1$, increment `counts[A[i]]`.
3. Compute cumulative frequencies (number of elements having value less than i for $0 \leq i \leq k$).

   ```
   C[0] ← 0
   C[1] ← C[0]
   C[2] ← C[0] + C[1]
   C[3] ← C[0] + C[1] + C[2] ...
   ```

   Do this in reverse order!
4. $A[i]$ will have to be placed between `C[A[i]]` and `C[A[i+1]]`.

Counting sort



Digression I: function pointers

- Declaring function pointers

```
<return type> (* <function name>) ( <parameter list> )
```

Example:

```
int *aFunction(int), *(*aFunctionPointer)(int);
```

- Using function pointers

```
(*f)(...)
```

- Setting function pointer variables / passing function pointers as arguments: simply use the name of the function

Example:

```
aFunctionPointer = aFunction;
```

Digression II: void pointers

- Pointers to a *generic* chunk of memory
- Programmer needs to know the actual type in order to do anything useful
- Useful for writing *generic* code

Type independent

```
char c, *cp;
char *str, **sp;
int i, *ip;
float f;

void *voidPointer;
```

```
voidPointer = (void *) &c;
* ((char *) voidPointer) = 'A';
printf("%c\n", c);

voidPointer = (void *) &i;
* ((int *) voidPointer) = 10;
printf("%d\n", i);
```

Generic sort/search routines

```
#include <stdlib.h>
```

Sorting

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Searching

```
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Comparator routine: examples

```
int compare_int (void *elem1, void *elem2)
{
    int *ip1 = elem1;
    int *ip2 = elem2;
    return *ip1 - *ip2;
    /* Or more explicitly:
       int i1 = *((int *) elem1);
       int i2 = *((int *) elem2);
       return i1 - i2;
     */
}

int compare_strings (void *elem1, void *elem2)
{
    char **s1 = elem1; // Alt.: char *s1 = *((char **) elem1);
    char **s2 = elem2; // Alt.: char *s2 = *((char **) elem2);
    return strcmp (*s1, *s2);   // Alt.: return strcmp(s1, s2);
}
```

Using `qsort` and `bsearch`

```
char **strings;
int *a;
int num_strings, N;

qsort(a, N, sizeof(int), compare_int);
qsort(strings, num_strings, sizeof(char *), compare_strings);
```

Timsort

- https://en.wikipedia.org/wiki/Timsort
- https://realpython.com/sorting-algorithms-python/#pythons-built-in-sorting-algorithm

Exercise

The password file on a GNU/Linux machine looks like the example available from `https://www.dropbox.com/s/7x17j7z3alfkfdc/etc-passwd.txt?dl=0`. This file lists the accounts of MTCS students of ISI from the 2017-19 batch. It consists of multiple columns, separated by ':'. Columns 1,3, and 5 correspond respectively to a student's roll number, user ID and name.

Write a C program to read and store this information in the form of an array of structures, with each structure representing a single student. Now write appropriate comparator functions so that the `qsort` function can be used to sort the array in

- ascending order of roll numbers;
- descending order of roll numbers;
- alphabetic order by name.