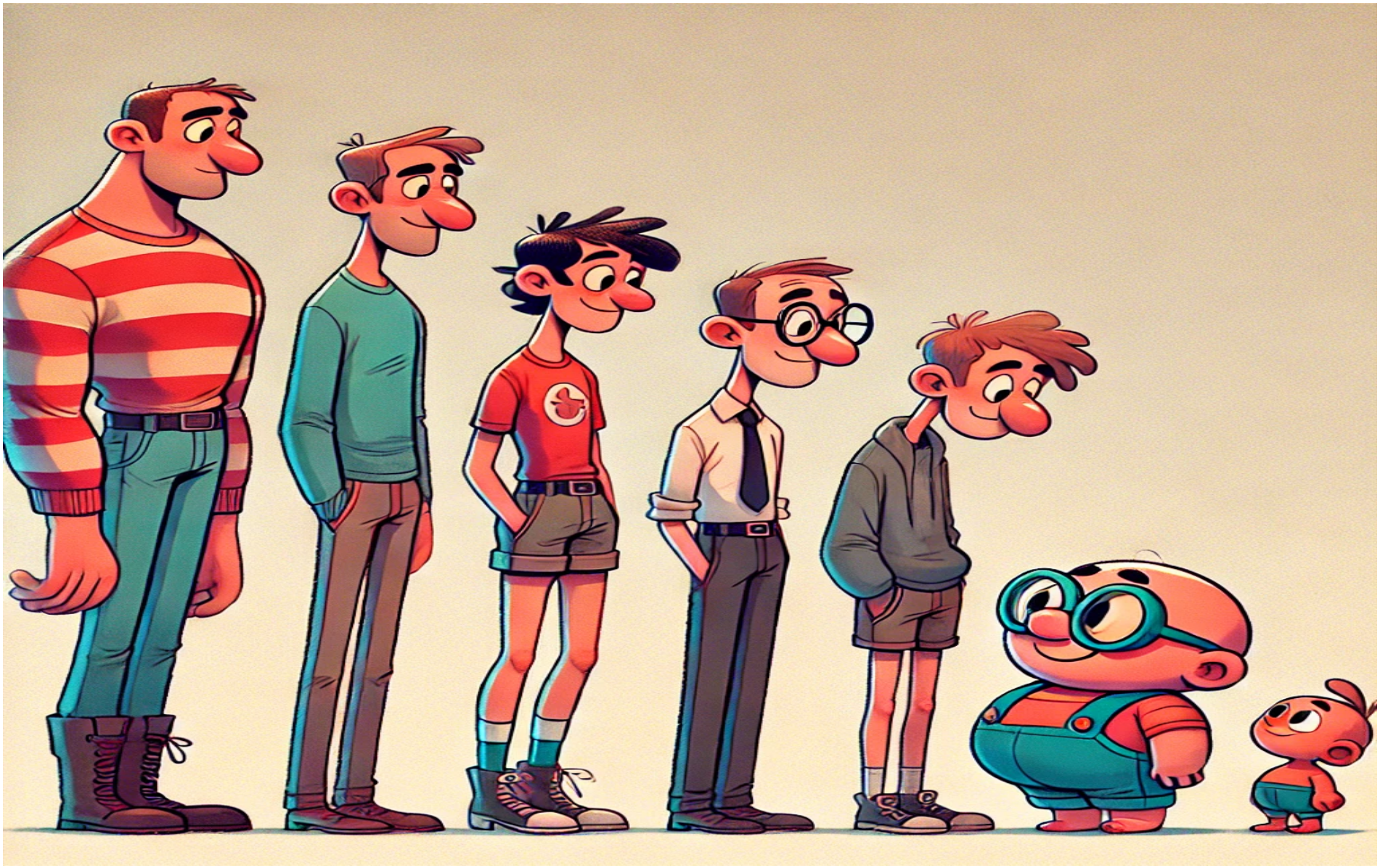
Programming and Data Structures - I

Lecture 17

Kripabandhu Ghosh

CDS, IISER Kolkata

SORTING



BUBBLE SORT

Bubble Sort

Key steps

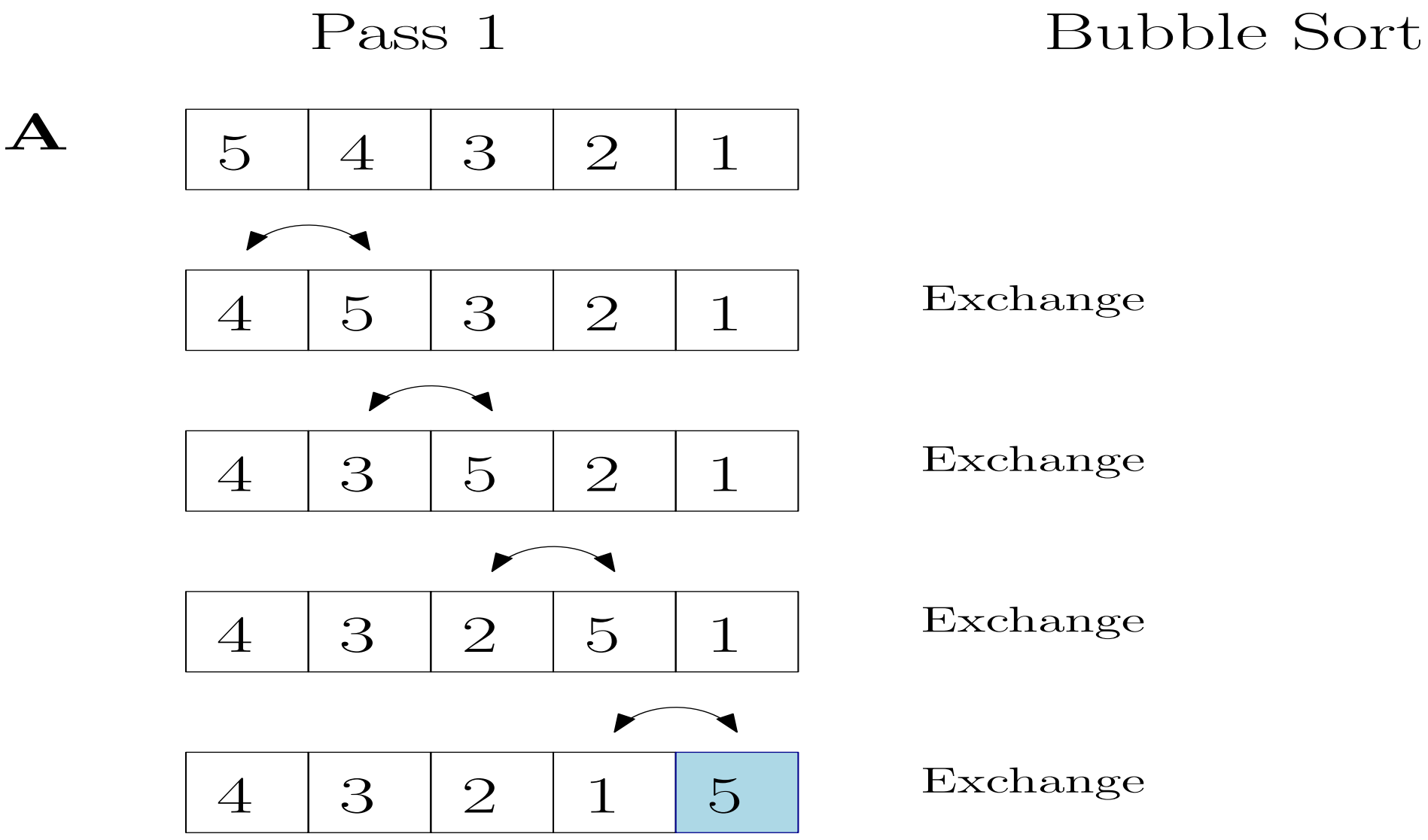
- Multiple passes
- In each pass,
  - an element (say a[j]) is compared with its successor (a[j + 1])
  - a[j] and a[j + 1] are interchanged if not in the desired order (for ascending order, interchange if a[j] $>$ a[j + 1])
- After the first pass, for ascending order sort, the largest element is at the last position (In general, after the *i*th interation, a[n - i + 1] is placed in its proper position)

Bubble Sort: Implementation

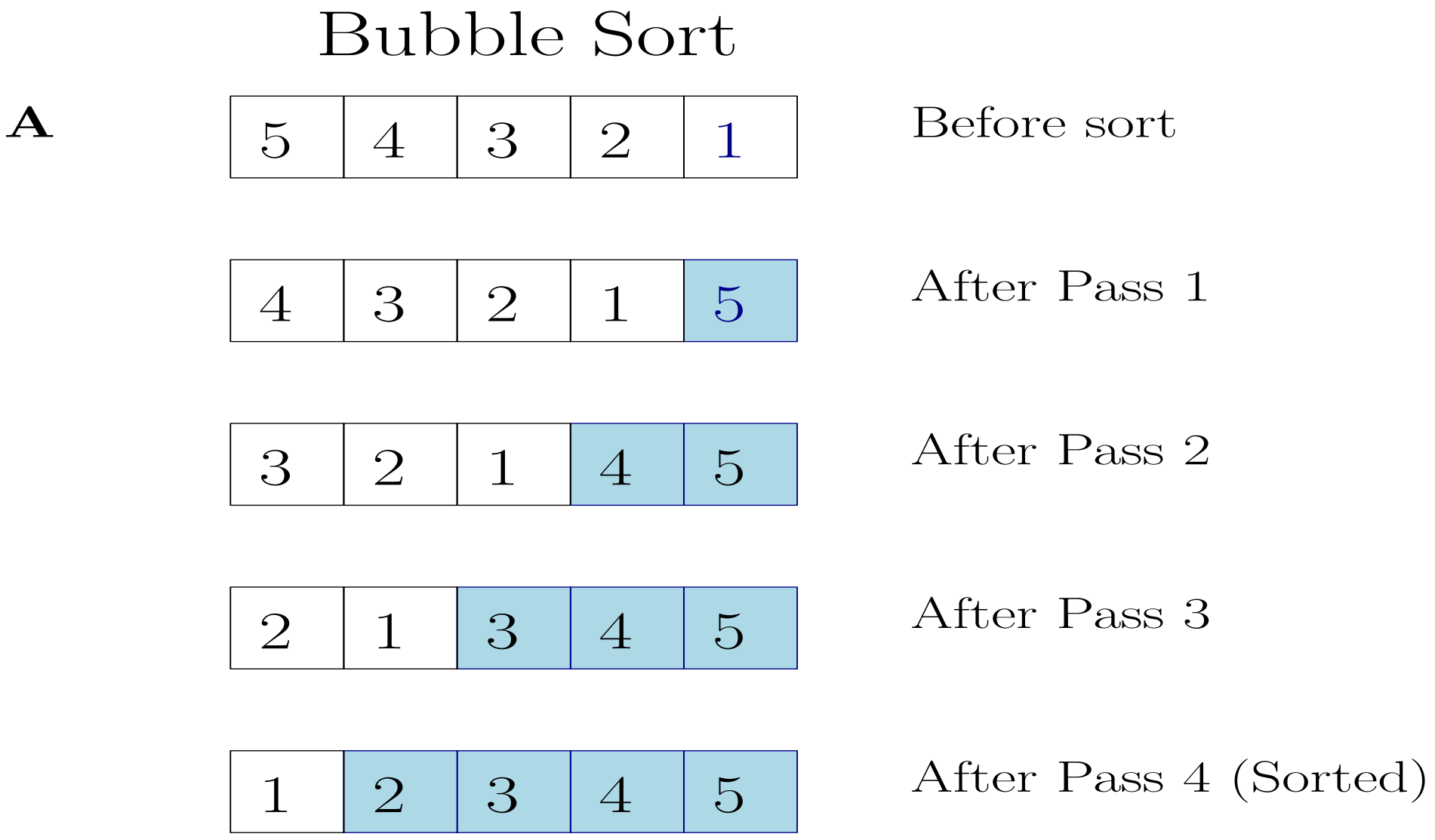
Code

```
for(i=0; i<n - 1; i++)
{
   for(j=0; j<n - 1 - i; j++)
   {
      if(arr[j] > arr[j+1])
      {
         temp = arr[j];
         arr[j] = arr[j+1];
         arr[j+1] = temp;
      }
   }
}
```

Bubble sort: illustration



Bubble sort: illustration (contd.)



Bubble Sort: Time Complexity

Intuition

- Number of passes = n - 1
- Number of steps per pass $\propto$ n - i (i = 1, 2, ..., no of passes)
- Total number of operations $\propto$ (n - 1) + (n - 2) + ... + 1 = $\frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}$

Asymptotic worst case time complexity

$$\mathcal{O}(n^2)$$

Bubble Sort: Conclusions

Advantages

- Easy to understand
- Introductory sorting algorithm
- In-place sort: No extra space requirement

Disadvantages

- Worst performing sorting algorithm: best, worst and average case complexity is $\mathcal{O}(n^2)$
- Impractical for large data

SELECTION SORT

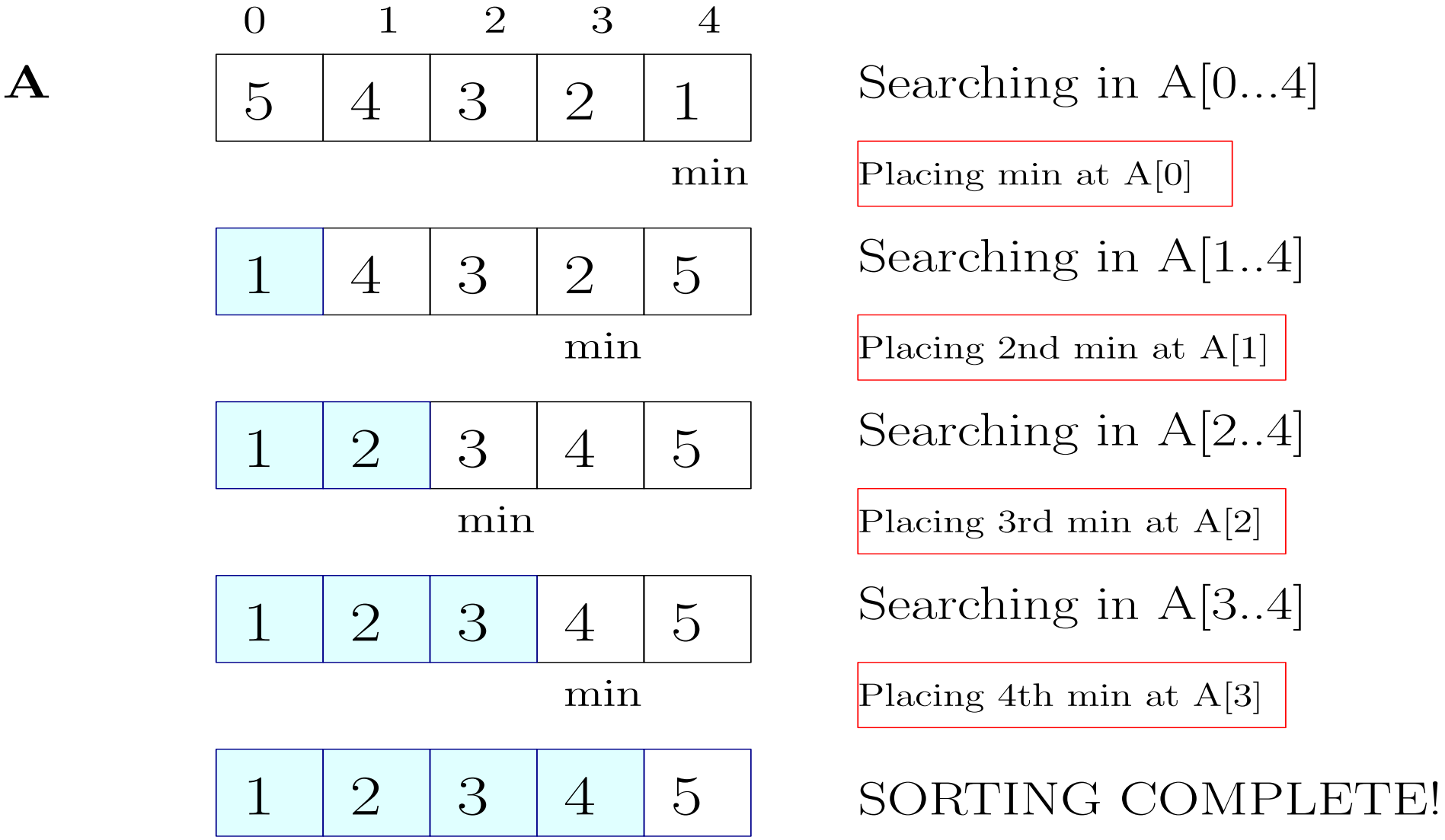
Selection Sort

Key steps

For an array $a[0..n-1]$

- In step 1, the smallest element in $a[0..n-1]$ is placed in $a[0]$; In step 2, the smallest element in $a[1..n-1]$ is placed in $a[1]$,..., in step n - 1, the smallest element in $a[n-2..n-1]$ is placed in $a[n-2]$
- In general, in step i, the smallest element in $a[i-1..n-1]$ is placed in $a[i-1]$
- At each step, the original element in $a[i-1]$ is swapped with the smallest element

Selection Sort: illustration



Selection Sort

Code

```
for(i=0; i<n-1; i++)
{
/*Find the index of smallest element*/
   min=i;
   for(j=i+1; j<n ; j++)
   {
      if(arr[min] > arr[j])
         min=j ;
   }
   if(i!=min) Swap smallest element with arr[i]
   {
      temp = arr[i];
      arr[i] = arr[min];
      arr[min] = temp ;
   }
}
```

Selection Sort: Time Complexity

Intuition

- Number of passes = n - 1
- Number of steps per pass $\propto$ (n - i) + 1 (i = 1, 2, ..., no of passes)
- Total number of operations $\approx$ ((n - 1) + (n - 2) + ... + 1 + n - 1) = $\frac{(n-1)(n-1+1)}{2} + (n-1) = \frac{(n-1)n}{2} + n - 1 = \frac{n^2}{2} + \frac{n}{2}$ - 1

Asymptotic worst case time complexity

$$\mathcal{O}(n^2)$$

Selection Sort: Conclusions

Advantages

- Easy to understand and implement
- In-place sort: No extra space requirement
- Considered to be faster than Bubble sort as it takes lesser (i.e. $\mathcal{O}(n)$) swaps

Disadvantages

- Best, worst and average case complexity is $\mathcal{O}(n^2)$
- Impractical for large data

QUICK SORT

Quick Sort (partition exchange sort)

Key steps

In an array *arr* of length n

- choose an element *pivot* (e.g. the first element)
- **Partitioning**
  - place *pivot* in the expected position in the ascending sorted permutation of *arr*
  - the sub-array *arr*[0...*pivot* - 1] contains elements in *arr* which are smaller than (or equal to) *pivot*
  - the sub-array *arr*[*pivot* + 1 ... n - 1] contains elements in a which are greater than (or equal to) *pivot*
- Partitioning continues in the subarrays until all the *pivot*s are correctly placed in *arr*

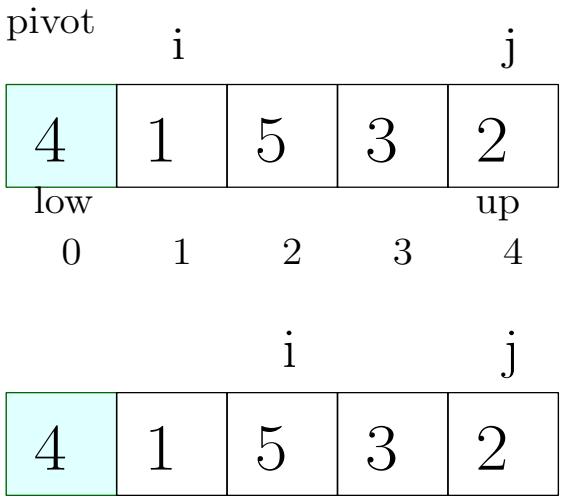
Developed by British computer scientist **Tony Hoare** (Turing award winner).

Quick Sort: Partitioning

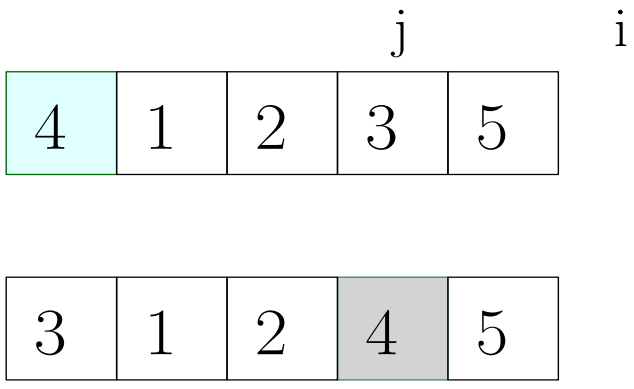
Code

```
int partition(int arr[], int low, int up)
{
  int temp, i, j, pivot;
  i = low + 1;
  j = up;
  pivot = arr[low];

  while(i <= j)
  {
    while(arr[i] < pivot && i<up)
      i++;

    while( arr[j] > pivot )
      j- -;

    if(i < j)
    {
      temp=arr[i];
      arr[i]=arr[j];
      arr[j]=temp;
      i++;
      j- -;
    }
    else
      i++;
  }

  arr[low] = arr[j];
  arr[j] = pivot;
  return j;
}
```

Quick sort partitioning: illustration




```
while(arr[i] < pivot && i<up)
      i++;
 while(arr[j] > pivot )
      j−−;

      if(i < j)
         swap(arr[i], arr[j])
      else
          i++;
```



Swapping and continuation of the loop

```
arr[low]=arr[j];
arr[j]=pivot;
```

Partitioning at 4

left subarray right subarray

Quick Sort: Subroutine

Code

```
void quick(int arr[],int low,int up)
{
   int pivloc;
   if(low >= up)
      return;
   pivloc = partition(arr, low, up);
   quick(arr,low, pivloc - 1); /*recursive call to left subarray*/
   quick(arr, pivloc + 1, up); /*recursive call to right subarray*/
}
```

Quick Sort: main

Code

```
main()
{
   int array[MAX], n, i;
   printf("Enter the number of elements : ");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      printf("Enter element %d : ",i+1);
      scanf("%d", &array[i]);
   }
   quick(array, 0, n-1);
   printf("Sorted list is :\n");
   for( i=0; i<n; i++ )
      printf("%d ",array[i]);
   printf("\n");
}
```

Quick Sort: Time Compexity

Best and average case

- If partitioning divides the array (of length n) into near equal halves every time, the number of partition steps be k
- $\frac{n}{2^k} = 1 \Rightarrow k \approx log_2 n$ (from n we end when the sub-arrays contain single elements)
- Each partition step has maximum of n operations (approx)
- Total number of steps $\approx n * log_2 n = \mathcal{O}(nlog_2 n)$

Quick Sort: Time Compexity

Worst case

- If partitioning divides the array (of length n) into subarrays of size 0 and n - 1 (e.g. if the array is in descending order), the number of partition steps be k
- k = n - 1 (from n we end when the sub-arrays contain single elements)
- Each partition step has maximum of n operations (approx)
- Total number of steps $\approx n * (n - 1) = \mathcal{O}(n^2)$

Quick Sort: Conclusions

Worst case

- Worst case rarely happens
- Even for highly unbalanced partitions (e.g. 9:1 or even 99:1) the complexity remains $\mathcal{O}(nlog_2 n)$
- Preferred algorithm for large data

MERGE

Merge

Purpose

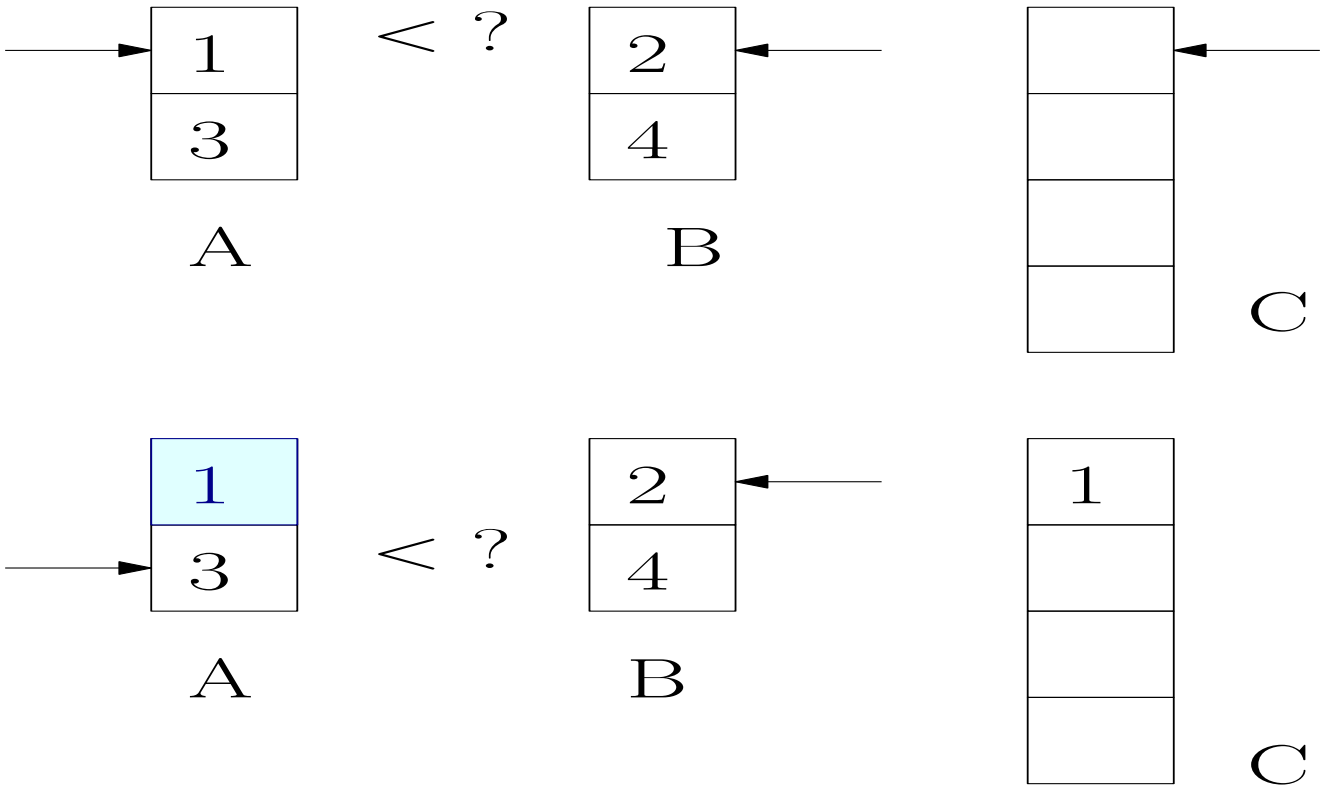
Produce a sorted list from sorted sublists

Features

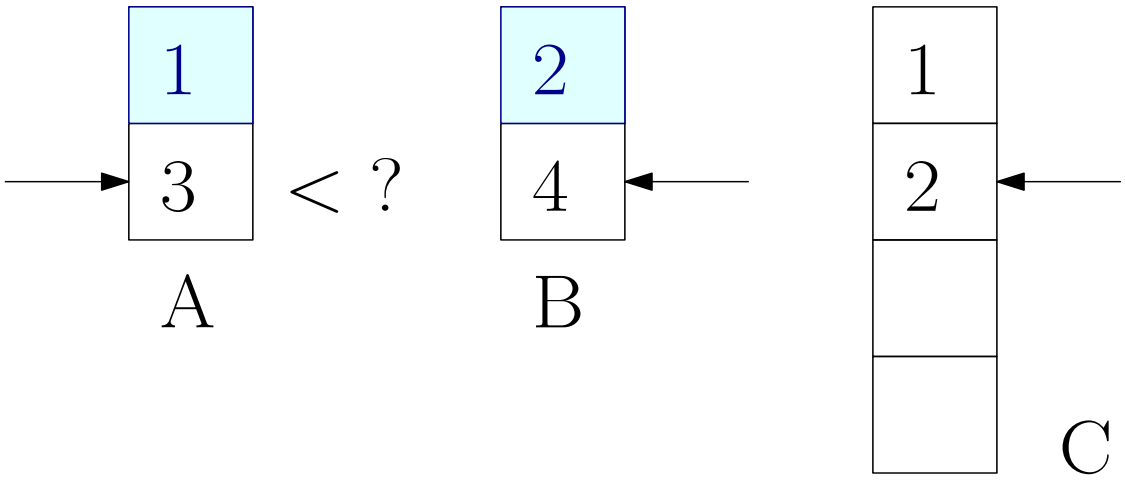
For two sorted lists $A[0...n-1]$ and $B[0...m-1]$ and an empty list C

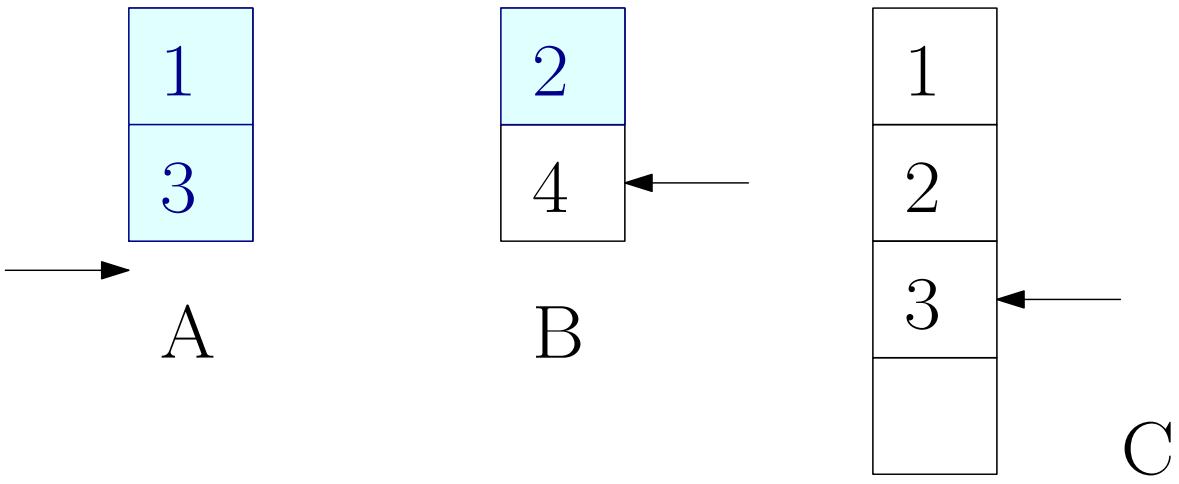
1. Compare $A[0]$ and $B[0]$ and put the smaller in C; consider the element next to which is put in C
2. In general, put the smaller of $A[i]$ and $B[j]$ in C; increment i or j according as whether $A[i]$ or $B[j]$ is copied to C
3. If all elements in A (or B) are copied to C, copy the remaining elements of B (or A) to C

Merge: illustration



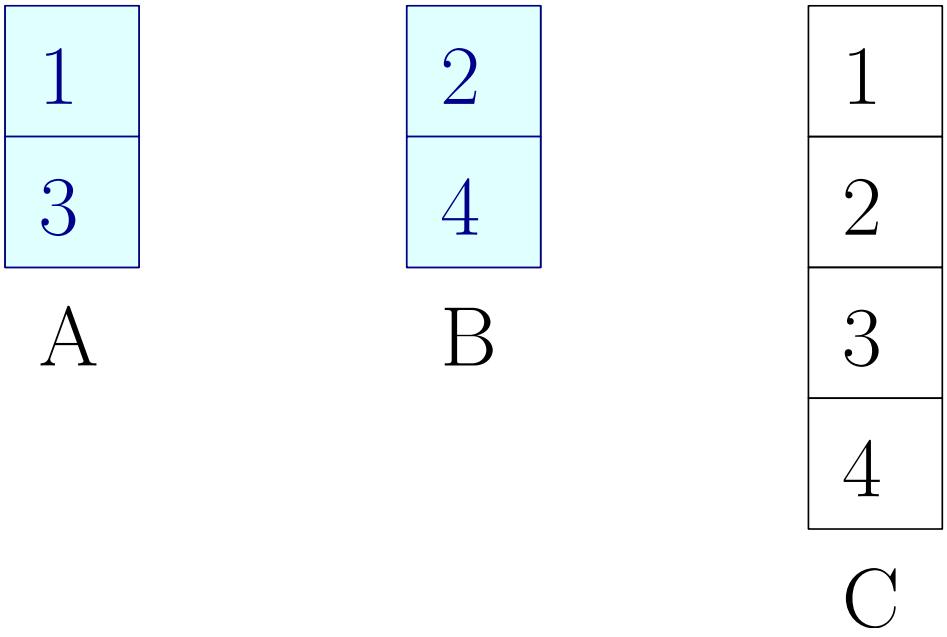
Merge: illustration





Excess element in B

Merge: illustration



MERGE COMPLETE!

Merge: implementation

Code

```
void merge(int arr1[], int arr2[], int temp[], int n1, int n2 ) //Merge arr1 and arr2 to temp
{
   int i = 0;
   int j = 0;
   int k = 0;

   while(i <= n1 && j <= n2)
   {
      if( arr1[i] <= arr2[j])
         temp[k++] = arr1[i++];
      else
         temp[k++] = arr2[j++];
   }
   while(i <= n1) //Copy excess elements of arr1 to temp
   {
      temp[k++] = arr1[i++];
   }
   while(j<=n2) //Copy excess elements of arr2 to temp
   {
      temp[k++] = arr2[j++];
   }
}
```

Merge: Conclusions

- Time complexity $= \mathcal{O}(m + n)$ (m, n being lengths of the two sublists)
- Vital component of efficient sorting algorithms like **Merge Sort**

MERGE SORT

Merge Sort

Key steps

For a list (array) $A[0...n-1]$

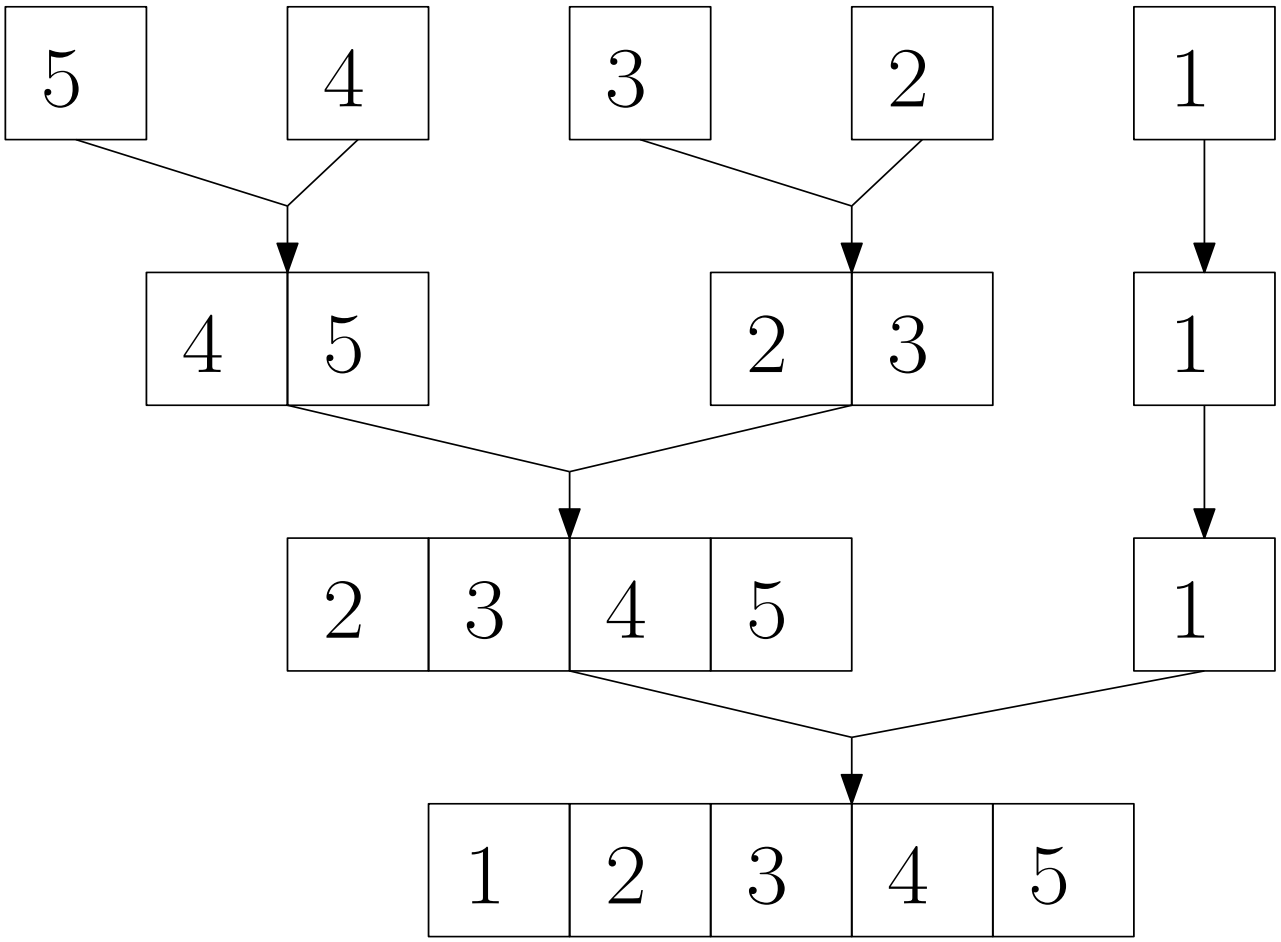
1. Split into singleton sublists $A[0]$, $A[1]$, ..., $A[n-1]$
2. Use **merge** algorithm to merge these sublists of sizes 1, 2, ...
3. Finally a sorted list is produced

Approaches

- **Bottom Up**: Merging sublists of size 1 iteratively to produce the sorted list.
- **Top down**: Recursively splitting the list until each sublist is of size 1 and merging them to produce the sorted list.

Developed by **Herman Heine Goldstine** and **John von Neumann** in 1948.

Merge Sort: illustration of merging



Singleton sublists

Merging of size 1

Merging of size 2

Final merge

Merge Sort: Time Complexity

- No of passes (merging of sizes 1, 2, ...) $\propto \lceil log_2 n \rceil$
- No of operations per pass $\propto n$ (maximum size of each sublist is $\frac{n}{2}$; merge operation has complexity $\mathcal{O}(n)$)
- Total number of steps $\propto n \lceil log_2 n \rceil = \mathcal{O}(nlog_2 n)$

Merge Sort: Conclusions

Advantage

Time complexity is always $\mathcal{O}(nlog_2 n)$

Disdvantage

Requires $\mathcal{O}(n)$ extra space (an extra array of length n)


AHKNST!
AHKNST!