# Contents

# Section outline

1. **Simple sorting**
   - Selection Sort
   - Bubble Sort
   - Insertion Sort

# Motivation of Selection Sort

- Select smallest element
- Interchange with top element
- Repeat procedure leaving out the top element

# Recursive Selection Sort

**Editor:**

```
void selectionSortR(int Z[], int sz) {
 int sel, i, t;
   if (sz<=0) return;
   for (i=sz-1,minI=i,i--;i=>0;i--)
    // select the smallest element
    if (Z[i]<Z[minI]) minI = i;
    // interchange the min element with the top element
    t=Z[minI];
    Z[minI]=Z[0];
    Z[0]=t;
    // now sort the rest of the array
    selectionSortR(Z+1, sz-1);
}
```

# Iterative Selection Sort

**Editor:**

```
void selectionSortI(int Z[], int sz) {
 int sel, i, t;
 for (j=sz; j>0; j--) { // from full array, decrease
   for (i=sz-1,minI=i,i--;i=>sz-j;i--)
   // sz-j varies from 0 to sz-1 and i from sz-2 to sz-j
    // select the smallest element
    if (Z[i]<Z[minI]) minI = i;
    // interchange the min element with the top element
    t=Z[minI];
    Z[minI]=Z[sz-j];
    Z[sz-j]=t;
    // now sort the rest of the array
 }
}
```

# Motivation of Bubble Sort

- Start from the bottom and move upwards
- If an element is smaller than the one over it, then interchange the two
- The smaller element bubbles up
- Smallest element at top at the end of the pass
- Repeat procedure leaving out the top element

# Recursive Bubble Sort

**Editor:**

```
void bubbleSortR(int Z[], int sz) {
 int i;
   if (sz<=0) return;
   for (i=sz-1;i>0;i--)
    // the smallest element bubbles up to the top
    if (Z[i]<Z[i-1]) {
      int t;
      t=Z[i];
      Z[i]=Z[i-1];
      Z[i-1]=t;
    }
    // now sort the rest of the array
    bubbleSortR(Z+1, sz-1);
}
```

# Iterative Bubble Sort

**Editor:**

```
void bubbleSortI(int Z[], int sz) {
 int i, j;
 for (j=sz; j>0; j--) // from full array, decrease
   for (i=sz-1;i>sz-j;i--)
    // the smallest element bubbles up to the top
    if (Z[i]<Z[i-1]) {
      int t;
      t=Z[i];
      Z[i]=Z[i-1];
      Z[i-1]=t;
    }
}
```

# Insert sorted

**Editor:**

```
void insertSorted(int Z[], int ky, int sz) {
// insert ky at the correct place
// original array should have free locations
// sz is number of elements currently in the array
// sz is not the allocated size of the array
 int i, pos=searchBinRAF(Z, ky, sz, 0);
 if (pos<0) pos=-(pos+10);
 // compensation specific to searchBinRAF
 // now shift down all elements from pos onwards
 for (i=sz;i>pos;i--) // start from the end! (why?)
  Z[i]=Z[i-1];
 Z[pos]=ky; // now the desired position is available
}
```

# Insertion Sort

**Editor:**

```
void insertionSort(int Z[], int sz) {
 int i;
 for (i=1;i<sz;i++)
  // elements 0..(i-1) are sorted, element Z[i]
  // is to be placed so that elements 0..i are also
sorted
  insertSorted(Z, Z[i], i);
}
```

# Section outline

2 **Mergesort**
- Merging two sorted arrays
- Merge sort
- Complexity of mergesort
- In-place merging
- Analysing mergesort with in-place merging

# Merging two sorted arrays

First array | 2 | 5 | 9 | 23 | 40 |

| 1 | 3 | 4 | 29 | 55 | 65 | 68 | Second array

Merged sequence

# Merging two sorted arrays

First array | 2 | 5 | 9 | 23 | 40 |

| 3 | 4 | 29 | 55 | 65 | 68 | Second array

Merged sequence | 1 |

# Merging two sorted arrays

First array

| | | | |
|---|---|---|---|
| 5 | 9 | 23 | 40 |

| | | | | | |
|---|---|---|---|---|---|
| 3 | 4 | 29 | 55 | 65 | 68 |

Second array

Merged sequence

| | |
|---|---|
| 1 | 2 |

# Merging two sorted arrays

First array

| | | | |
|---|---|---|---|
| 5 | 9 | 23 | 40 |

Second array

| | | | | |
|---|---|---|---|---|
| 4 | 29 | 55 | 65 | 68 |

Merged sequence

| | | |
|---|---|---|
| 1 | 2 | 3 |

# Merging two sorted arrays

First array | 5 | 9 | 23 | 40 |

| 29 | 55 | 65 | 68 | Second array

Merged sequence | 1 | 2 | 3 | 4 |

# Merging two sorted arrays

First array | 9 | 23 | 40 |

| 29 | 55 | 65 | 68 | Second array

Merged sequence | 1 | 2 | 3 | 4 | 5 |

# Merging two sorted arrays

First array | 23 | 40 |

| 29 | 55 | 65 | 68 | Second array

Merged sequence | 1 | 2 | 3 | 4 | 5 | 9 |

# Merging two sorted arrays

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| First array | 40 | | | 29 | 55 | 65 | 68 | Second array |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Merged sequence | 1 | 2 | 3 | 4 | 5 | 9 | 23 |

# Merging two sorted arrays

First array | 40 |

| 55 | 65 | 68 | Second array

Merged sequence | 1 | 2 | 3 | 4 | 5 | 9 | 23 | 29 |

# Merging two sorted arrays

First array

| | | |
|---|---|---|
| 55 | 65 | 68 |

Second array

Merged sequence

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 23 | 29 | 40 |

# Merging two sorted arrays

First array

| | |
|---|---|
| 65 | 68 |

Second array

Merged sequence

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 23 | 29 | 40 | 55 |

# Merging two sorted arrays

First array | 68 | Second array

Merged sequence | 1 | 2 | 3 | 4 | 5 | 9 | 23 | 29 | 40 | 55 | 65 |

# Merging two sorted arrays

First array

Second array

Merged sequence

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 23 | 29 | 40 | 55 | 65 | 68 |

# Recursive definition of merging

- $M$, $N$ are number of elements in $A$ and $B$, respectively
- Array indices start from 0
- Initial call: $\text{merge}(A, N, 0, B, M, 0, C, 0)$ (1.1)

# Recursive definition of merging

- $M$, $N$ are number of elements in $A$ and $B$, respectively
- Array indices start from 0
- Initial call: $\text{merge}(A, N, 0, B, M, 0, C, 0)$ (1.1)

$$\text{merge}(A, N, i, B, M, j, C, k)$$
$$= \begin{cases} \text{if } (i \geq N \wedge j \geq M) \text{ then done} & (1.2) \\ \text{else if } (i \geq N) \text{ then merge}(A, N, i, B, M, j+1, C, k+1), \\ \text{ st } C[k] = B[j] & (1.3) \\ \text{else if } (j \geq M) \text{ then merge}(A, N, i+1, B, M, j, C, k+1), \\ \text{ st } C[k] = A[i] & (1.4) \\ \text{else if } (A[i] \leq B[j]) \text{ then merge}(A, N, i+1, B, M, j, C, k+1), \\ \text{ st } C[k] = A[i] & (1.5) \\ \text{otherwise merge}(A, N, i, B, M, j+1, C, k+1), \\ \text{ st } C[k] = B[j] & (1.6) \end{cases}$$

Definition is tail recursive

# Recursive function for merging

## Editor:

```
void merge(int A[], int N, int i,
    int B[], int M, int j, int C[], int k) {
  if (i >= N && j >= M) return; // by clause (2)
  else if (i >= N) { // by clause (3)
    C[k] = B[j];
    merge(A, N, i, B, M, j + 1, C , k + 1);
  } else if (j >= M) { // by clause (4)
    C[k] = A[i];
    merge(A, N, i + 1, B, M, j, C , k + 1);
  } else if (A[i] <= B[j]) { // by clause (5)
    C[k] = A[i];
    merge(A, N, i + 1, B, M, j, C , k + 1);
  } else { // by clause (6)
    C[k] = B[j];
    merge(A, N, i, B, M, j + 1, C , k + 1);
  }
}
```

# Testing the recursive merging

**Editor:**

```
#include <stdio.h>
void showIArr(int A[], int n) {
  int i;
  for (i=0; i<n; i++) printf("%d ", A[i]);
  printf("\n");
}

int main() {
  int A[]={2, 5, 9, 23, 40};
  int B[]={1, 3, 4, 29, 55, 65, 68};
  int C[12];
  printf("after merging "); showIArr(A,5);
  printf("and "); showIArr(B,7);
  merge(A, 5, 0, B, 7, 0, C , 0);
  showIArr(C,12);
return 0; }
```

# Results of test

**Shell:**

```
$ make mergeSort ; ./mergeSort
cc     mergeSort.c   -o mergeSort
after merging 2 5 9 23 40
and 1 3 4 29 55 65 68
1 2 3 4 5 9 23 29 40 55 65 68
```

# Iterative function for merging

**Editor:**

```
void mergeI(int A[], int N, int B[], int M, int C[]) {
  int i=0, j=0, k=0; // by clause (1)
  do {
    if (i >= N && j >= M) break; // by clause (2)
    else if (i >= N) { // by clause (3)
      C[k] = B[j]; j++, k++;
    } else if (j >= M) { // by clause (4)
      C[k] = A[i]; i++; k++;
    } else if (A[i] <= B[j]) { // by clause (5)
      C[k] = A[i]; i++; k++;
    } else { // by clause (6)
      C[k] = B[j]; j++; k++;
    }
  } while (1);
}
```

# Testing the iterative merging

**Editor:**

```
void showIArr(int A[], int n) {
  int i;
  for (i=0; i<n; i++) printf("%d ", A[i]);
  printf("\n");
}

int main() {
  int A[]={2, 5, 9, 23, 40};
  int B[]={1, 3, 4, 29, 55, 65, 68};
  int C[12];
  printf("after merging "); showIArr(A,5);
  printf("and "); showIArr(B,7);
  printf ("by mergeR "); mergeR(A, 5, 0, B, 7, 0, C, 0);

  showIArr(C,12);
  printf ("by mergeI "); mergeI(A, 5, B, 7, C);
  showIArr(C,12);
```

# Results of test

**Shell:**

```
$ make mergeSort ; ./mergeSort
cc     mergeSort.c   -o mergeSort
after merging 2 5 9 23 40
and 1 3 4 29 55 65 68
by mergeR 1 2 3 4 5 9 23 29 40 55 65 68
by mergeI 1 2 3 4 5 9 23 29 40 55 65 68
```

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

- Sort second part

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

- Sort second part

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

- Sort second part

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

- After sorting the two parts:

First

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

Second

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

- Sort second part

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

- After sorting the two parts:

First

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

Second

- Merge the two sorted sequences

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

- Sort second part

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

- After sorting the two parts:

First

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

Second

- Merge the two sorted sequences

```
merge(A, nA, B, nB, C)
```

# Merging two sorted arrays

- Given array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Split given array into two parts

| | | | | | |
|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Sort first part

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

- Sort second part

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

- After sorting the two parts:

First

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 23 | 40 | 68 |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 29 | 55 | 65 |

Second

- Merge the two sorted sequences

```
merge(A, nA, B, nB, C)
```

- After merging the two sorted parts (the required result)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 23 | 29 | 40 | 55 | 65 | 68 |

# Recursive definition of mergesort

- $N$ is the number of elements in $A$
- Array indices start from 0

# Recursive definition of mergesort

- $N$ is the number of elements in $A$
- Array indices start from 0

$$\text{mergeSort}(A, N, C) = \begin{cases} \text{if } (N \leq 1) \text{ then done} & (2.1) \\ \text{let } M = N/2 & (2.2) \\ \text{do mergeSort}(A, M, C) & (2.3) \\ \text{do mergeSort}(A + M, N - M, C) & (2.4) \\ \text{do merge}(A, M, A + M, N - M, C) & (2.5) \\ \text{do copyBack}(A, C, N) & (2.6) \end{cases}$$

# Code for `mergeSort`

**Editor:**

```
void mergeSort(int A[], int N, int C[]) {
   int M;
   if (N<=1) return; // by clause (1)
   M = N/2; // by clause (2)
   mergeSort(A, M, C); // by clause (3)
   mergeSort(A + M, N - M, C); // by clause (4)
   mergeI(A, M, A + M, N - M, C); // by clause (5)
   copyBack(A, C, N); // by clause (6)
}
```

# Code for mergeSort

**Editor:**

```
void mergeSort(int A[], int N, int C[]) {
  int M;
  if (N<=1) return; // by clause (1)
  M = N/2; // by clause (2)
  mergeSort(A, M, C); // by clause (3)
  mergeSort(A + M, N - M, C); // by clause (4)
  mergeI(A, M, A + M, N - M, C); // by clause (5)
  copyBack(A, C, N); // by clause (6)
}

void copyBack(int A[], int C[], int N) {
  int i;
  for (i=0;i<N;i++) A[i]=C[i];
}
```

# Testing `mergeSort`

**Editor:**

```
int main() {
  int A[]={23,5,40,2,9,68,55,4,3,1,65,29};
  int C[12];
  printf("after sorting by mergeSort ");
  mergeSort(A, 12, C);
  printf("\n"); showIArr(A,12);
return 0; }
```

# Results of testing `mergeSort`

**Shell:**

```
$ make mergeSort ; ./mergeSort
cc     mergeSort.c   -o mergeSort
after sorting by mergeSort
1 2 3 4 5 9 23 29 40 55 65 68
```

# Complexity of mergesort

- $T(n) = T(n/2) + T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n)$
- $T(0) = \Theta(1)$
- $T(n) = \Theta(n \lg n)$
- Given implementation requires extra storage space (equal to size of input)
- In-place version not trivial if $O(N \lg N)$ is to be preserved
- Simple in-place version to be studied next

# Complexity of mergesort

- $T(n) = T(n/2) + T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n)$
- $T(0) = \Theta(1)$
- $T(n) = \Theta(n \lg n)$
- Given implementation requires extra storage space (equal to size of input)
- In-place version not trivial if $O(N \lg N)$ is to be preserved
- Simple in-place version to be studied next
- Mergesort was invented by John von Neumann in 1945

# In-place merging

- $M$ is number of elements in the first half of $A$
- $N$ is the total number of elements in $A$
- Array indices start from 0
- Initial call: $\mathrm{merge}(A, M, N)$ (4.1)

# In-place merging

- $M$ is number of elements in the first half of $A$
- $N$ is the total number of elements in $A$
- Array indices start from 0
- Initial call: $\text{merge}(A, M, N)$ (4.1)

$$\text{mergeInPl}(A, M, N) = \begin{cases} \text{if } (M \leq 0 \vee M \geq N \vee N \leq 0) \text{ then done} & (4.2) \\ \text{else if } (A[0] \leq A[M]) \text{ then mergeInPl}(A + 1, M - 1, N - 1) & (4.3) \\ \text{else mergeInPl}(\text{cpySft}(A, M), M, N - 1) & (4.4) \end{cases}$$

# In-place merging

- $M$ is number of elements in the first half of $A$
- $N$ is the total number of elements in $A$
- Array indices start from 0
- Initial call: $\text{merge}(A, M, N)$ (4.1)

$$\text{mergeInPl}(A, M, N)$$
$$= \begin{cases} \text{if } (M \leq 0 \vee M \geq N \vee N \leq 0) \text{ then done} & (4.2) \\ \text{else if } (A[0] \leq A[M]) \text{ then mergeInPl}(A+1, M-1, N-1) & (4.3) \\ \text{else mergeInPl}(\text{cpySft}(A, M), M, N-1) & (4.4) \end{cases}$$

**Editor:**

```
void mergeInPl(int A[], int M, int N) {
  if ( M<=0 or M >= N or N <= 0 ) return;
  else if ( A[0] <= A[M] ) mergeInPl(A+1, M-1, N-1);
  else { // cpySft() is inlined
    int i, T = A[M];
    for (i=M; i; i--) A[i] = A[i-1];
    A[0] = T;   mergeInPl(A+1, M-1, N-1);
  }}
```

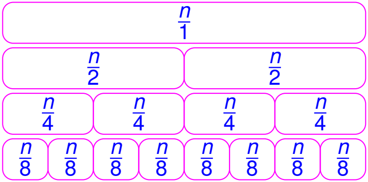
# Analysing mergesort with in-place merging

$$T(n) = \begin{cases} a & n = 1 \\ 2T\left(\frac{n}{2}\right) + bn + cn^2 + d & n > 1, n = 2^d, d \geq 0 \end{cases}$$

# Analysing mergesort with in-place merging

$$T(n) = \begin{cases} a & n = 1 \\ 2T\left(\frac{n}{2}\right) + bn + cn^2 + d & n > 1, n = 2^d, d \geq 0 \end{cases}$$

Let $n = 8$




$$b \times \frac{n}{1} \times 1 + c \times \frac{n^2}{1^2} \times 1 + d \times 1$$
$$b \times \frac{n}{2} \times 2 + c \times \frac{n^2}{2^2} \times 2 + d \times 2$$
$$b \times \frac{n}{4} \times 4 + c \times \frac{n^2}{4^2} \times 4 + d \times 4$$
$$a \times 8$$

# Analysing mergesort with in-place merging

$$T(n) = \begin{cases} a & n = 1 \\ 2T\left(\frac{n}{2}\right) + bn + cn^2 + d & n > 1, n = 2^d, d \geq 0 \end{cases}$$

Let $n = 8$

$\frac{n}{1}$

$\frac{n}{2}$ $\frac{n}{2}$

$\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$

$\frac{n}{8}$ $\frac{n}{8}$ $\frac{n}{8}$ $\frac{n}{8}$ $\frac{n}{8}$ $\frac{n}{8}$ $\frac{n}{8}$ $\frac{n}{8}$

$$b \times \frac{n}{1} \times 1 + c \times \frac{n^2}{1^2} \times 1 + d \times 1$$
$$b \times \frac{n}{2} \times 2 + c \times \frac{n^2}{2^2} \times 2 + d \times 2$$
$$b \times \frac{n}{4} \times 4 + c \times \frac{n^2}{4^2} \times 4 + d \times 4$$
$$a \times 8$$

- $\lg n \times b \times n + \left(1 + \frac{1}{2} + \ldots + \frac{1}{2^{\lg n - 1}}\right) \times c \times n^2 + (n-1) \times d + n \times a =$
  $\lg n \times b \times n + 2 \times \left(1 - 2^{-\lg n}\right) \times c \times n^2 + (n-1) \times d + n \times a =$
  $\lg n \times b \times n + 2 \times \left(\frac{n-1}{n}\right) \times c \times n^2 + (n-1) \times d + n \times a$
- Asymptotic bound: $T(n) \in \Theta(n^2)$

# Section outline

3 **Quicksort**

- Simple version of quicksort
- In-place Version of Quick Sort
- Worst and best cases of complexity of quicksort
- Average case complexity of quicksort
- Upper bound on harmonic series

# Partitioning Leading to Sorting

- | | | | | | | | | | | | |
  |---|---|---|---|---|---|---|---|---|---|---|---|
  | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
  |---|---|---|---|---|---|---|---|---|---|---|---|

- Pick up any element *p*, say 9

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
  |---|---|---|---|---|---|---|---|---|---|---|---|
- Pick up any element $p$, say 9
- Partition all elements in the array into two sets,

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
  |---|---|---|---|---|---|---|---|---|---|---|---|
- Pick up any element $p$, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ $(< 9)$,

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
  |---|---|---|---|---|---|---|---|---|---|---|---|
- Pick up any element $p$, say 9
- Partition all elements in the array into two sets, first set: elements that are $< p$ ($< 9$), second set: elements that are $> p$ ($> 9$)
- Disregard ordering of elements within each set

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element $p$, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ ($< 9$),
  second set: elements that are $> p$ ($> 9$)
- Disregard ordering of elements within each set
- First set | 5 | 2 | 4 | 3 | 1 |

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element *p*, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ ($< 9$),
  second set: elements that are $> p$ ($> 9$)
- Disregard ordering of elements within each set
- First set | 5 | 2 | 4 | 3 | 1 |    | 23 | 40 | 68 | 55 | 65 | 29 | second set

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element $p$, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ ($<$ 9),
  second set: elements that are $> p$ ($>$ 9)
- Disregard ordering of elements within each set
- First set | 5 | 2 | 4 | 3 | 1 |      | 23 | 40 | 68 | 55 | 65 | 29 | second set
- Sort first set

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element $p$, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ ($< 9$),
  second set: elements that are $> p$ ($> 9$)
- Disregard ordering of elements within each set
- First set | 5 | 2 | 4 | 3 | 1 |    | 23 | 40 | 68 | 55 | 65 | 29 | second set
- Sort first set
  | 1 | 2 | 3 | 4 | 5 |    | 9 |

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element $p$, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ ($< 9$),
  second set: elements that are $> p$ ($> 9$)
- Disregard ordering of elements within each set
- First set | 5 | 2 | 4 | 3 | 1 |     | 23 | 40 | 68 | 55 | 65 | 29 | second set
- Sort first set
  | 1 | 2 | 3 | 4 | 5 |     | 9 |     | 23 | 40 | 68 | 55 | 65 | 29 |

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element *p*, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ ($< 9$),
  second set: elements that are $> p$ ($> 9$)
- Disregard ordering of elements within each set
- First set | 5 | 2 | 4 | 3 | 1 | | 23 | 40 | 68 | 55 | 65 | 29 | second set
- Sort first set
  | 1 | 2 | 3 | 4 | 5 | | 9 | | 23 | 40 | 68 | 55 | 65 | 29 |
- Sort second set

# Partitioning Leading to Sorting

- | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element *p*, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ ($< 9$),
  second set: elements that are $> p$ ($> 9$)
- Disregard ordering of elements within each set
- First set | 5 | 2 | 4 | 3 | 1 |     | 23 | 40 | 68 | 55 | 65 | 29 | second set
- Sort first set
  | 1 | 2 | 3 | 4 | 5 |     | 9 |     | 23 | 40 | 68 | 55 | 65 | 29 |
- Sort second set
  | 23 | 29 | 40 | 55 | 65 | 68 |

# Partitioning Leading to Sorting

- | | | | | | | | | | | | |
  |---|---|---|---|---|---|---|---|---|---|---|---|
  | 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |
- Pick up any element *p*, say 9
- Partition all elements in the array into two sets,
  first set: elements that are $< p$ $(< 9)$,
  second set: elements that are $> p$ $(> 9)$
- Disregard ordering of elements within each set
- First set [5 | 2 | 4 | 3 | 1] [23 | 40 | 68 | 55 | 65 | 29] second set
- Sort first set
  [1 | 2 | 3 | 4 | 5] [9] [23 | 40 | 68 | 55 | 65 | 29]
- Sort second set
  [1 | 2 | 3 | 4 | 5] [9] [23 | 29 | 40 | 55 | 65 | 68]
- Entire array is now sorted

  | | | | | | | | | | | | |
  |---|---|---|---|---|---|---|---|---|---|---|---|
  | 1 | 2 | 3 | 4 | 5 | 9 | 23 | 29 | 40 | 55 | 65 | 68 |

# Outline of Quicksort

- Given an array $A$ of $N$ elements
- Pick up a suitable element $p$ from the array

# Outline of Quicksort

- Given an array $A$ of $N$ elements
- Pick up a suitable element $p$ from the array
- Simple choice is to pick up the first element
- Partition the elements of $A$ based on $p$
- Let first part be all elements $< p$ or possibly $\leq p$
- Second part – all elements $> p$

# Outline of Quicksort

- Given an array $A$ of $N$ elements
- Pick up a suitable element $p$ from the array
- Simple choice is to pick up the first element
- Partition the elements of $A$ based on $p$
- Let first part be all elements $< p$ or possibly $\leq p$
- Second part – all elements $> p$
- Sort the two parts (does not matter which part is sorted first)

# Outline of Quicksort

- Given an array $A$ of $N$ elements
- Pick up a suitable element $p$ from the array
- Simple choice is to pick up the first element
- Partition the elements of $A$ based on $p$
- Let first part be all elements $< p$ or possibly $\leq p$
- Second part – all elements $> p$
- Sort the two parts (does not matter which part is sorted first)
- Now the whole of $A$ is sorted

# Outline of Quicksort

- Given an array $A$ of $N$ elements
- Pick up a suitable element $p$ from the array
- Simple choice is to pick up the first element
- Partition the elements of $A$ based on $p$
- Let first part be all elements $< p$ or possibly $\leq p$
- Second part – all elements $> p$
- Sort the two parts (does not matter which part is sorted first)
- Now the whole of $A$ is sorted
- Quicksort was invented by Tony Hoare in 1960

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

  Smaller | 5 | ... | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller | 5 | ... | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller | 5 | 2 | ... | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | | | |
|---|---|---|---|---|---|---|
| 68 | 55 | 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller | 5 | 2 | … | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | | |
|---|---|---|---|---|---|
| 55 | 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller | 5 | 2 | ... | 68 | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | | |
|---|---|---|---|---|
| 4 | 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller | 5 | 2 | | 55 | 68 | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | | |
|---|---|---|---|
| 3 | 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller | 5 | 2 | 4 | | 55 | 68 | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | | |
|---|---|---|
| 1 | 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller

| | | | |
|---|---|---|---|
| 5 | 2 | 4 | 3 |

| | | | |
|---|---|---|---|
| 55 | 68 | 40 | 23 |

Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

| | |
|---|---|
| 65 | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

Smaller

| | | | | |
|---|---|---|---|---|
| 5 | 2 | 4 | 3 | 1 |

| | | | |
|---|---|---|---|
| 55 | 68 | 40 | 23 |

Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**

  | |
  |---|
  | 29 |

- Let pivot element be 9
- Partitioning is done in another array **B**:

  Smaller | 5 | 2 | 4 | 3 | 1 | ... | 65 | 55 | 68 | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**
- Let pivot element be 9
- Partitioning is done in another array **B**:

  Smaller | 5 | 2 | 4 | 3 | 1 | | 29 | 65 | 55 | 68 | 40 | 23 | Larger

# Simple Partitioning Scheme

- Elements are originally in an array **A**
- Let pivot element be 9
- Partitioning is done in another array **B**:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Smaller | 5 | 2 | 4 | 3 | 1 | 9 | 29 | 65 | 55 | 68 | 40 | 23 | Larger |

# Simple Partitioning Scheme

- Elements are originally in an array **A**
- Let pivot element be 9
- Partitioning is done in another array **B**:
  Smaller | 5 | 2 | 4 | 3 | 1 | 9 | 29 | 65 | 55 | 68 | 40 | 23 | Larger

- Need to be careful while partitioning to avoid getting into an infinite loop
- Can be ensured by getting out at least one copy of the pivot

# Recursive Definition of Simple Partitioning

- Assume that all elements are distinct
- $M$ is the number of elements in $A$
- Array indices start from 0
- Initial call: $\text{simPartR}(A, N, 0, B, -1, N, p)$ (1.1)
- Last clause skips over the pivot
- At termination, pivot should be at $B[j+1]$, where it is explicitly assigned

# Recursive Definition of Simple Partitioning

- Assume that all elements are distinct
- $M$ is the number of elements in $A$
- Array indices start from 0
- Initial call: $\text{simPartR}(A, N, 0, B, -1, N, p)$ (1.1)
- Last clause skips over the pivot
- At termination, pivot should be at $B[j+1]$, where it is explicitly assigned

$$\text{simPartR}(A, N, i, B, j, k, p)$$

$$= \begin{cases} \text{if } (i \geq N) \text{ then } (j+1) \text{ st } B[j+1] = p & (1.2) \\ \text{else if } (A[i] < p) \text{ then simPartR}(A, N, i+1, B, j+1, k, p) & \\ \quad \text{st } B[j+1] = A[i] & (1.3) \\ \text{else if } (A[i] > p) \text{ then simPartR}(A, N, i+1, B, j, k-1, p) & \\ \quad \text{st } B[k-1] = A[i] & (1.4) \\ \text{otherwise simPartR}(A, N, i+1, B, j, k-1, p) & (1.5) \end{cases}$$

# Recursive Definition of Simple Partitioning

- Assume that all elements are distinct
- $M$ is the number of elements in $A$
- Array indices start from 0
- Initial call: $\text{simPartR}(A, N, 0, B, -1, N, p)$ (1.1)
- Last clause skips over the pivot
- At termination, pivot should be at $B[j+1]$, where it is explicitly assigned

$$\text{simPartR}(A, N, i, B, j, k, p)$$
$$= \begin{cases} \text{if } (i \geq N) \text{ then } (j+1) \text{ st } B[j+1] = p & (1.2) \\ \text{else if } (A[i] < p) \text{ then simPartR}(A, N, i+1, B, j+1, k, p) & \\ \quad \text{st } B[j+1] = A[i] & (1.3) \\ \text{else if } (A[i] > p) \text{ then simPartR}(A, N, i+1, B, j, k-1, p) & \\ \quad \text{st } B[k-1] = A[i] & (1.4) \\ \text{otherwise simPartR}(A, N, i+1, B, j, k-1, p) & (1.5) \end{cases}$$

Definition is tail recursive

# Recursive Code for Simple Partitioning

**Editor:**

```
int simPartR(int A[], int N, int i,
             int B[], int j, int k, int p) {
  if (i >= N) {
    B[j+1] = p;
    return (j+1); // by clause (2)
  } else if (A[i] < p) {
    B[j+1] = A[i];
    return simPartR(A, N, i+1, B, j+1, k, p);
    // by clause (3)
  } else if (A[i] > p) {
    B[k-1] = A[i];
    return simPartR(A, N, i+1, B, j, k-1, p);
    // by clause (4)
  } else // by clause (5)
    return simPartR(A, N, i+1, B, j, k, p);
}
```

# Recursive Definition of Simple Quick Sort

- $N$ is the number of elements in $A$
- Array indices start from 0
- Recursive simple partitioning is used

# Recursive Definition of Simple Quick Sort

- $N$ is the number of elements in $A$
- Array indices start from 0
- Recursive simple partitioning is used

$$\text{quickSimSort}(A, N, B) = \begin{cases} \text{if } (N \leq 1) \text{ then done} & (1.1) \\ \text{let } p = \text{simPartR}(A, N, 0, B, -1, N, A[0]) \\ \text{do copyBack}(A, B, N) & (1.2) \\ \text{do quickSimSort}(A, p, B) & (1.3) \\ \text{do quickSimSort}(A + p + 1, N - p - 1, B) & (1.4) \end{cases}$$

# Code for Simple Quick Sort

**Editor:**

```
void quickSimSort(int A[], int N, int B[]) {
  int pPos;

  if (N<=1) return;
  pPos = simPartR (A, N, 0, B, -1, N, A[0]);
  // printf("p=%d, pPos=%d, A[N=%d]: ", p, pPos, N);
  // showIArr(A,N);
  // pPos = simPartI (A, N, B, A[0]);
  copyBack(A, B, N);
  quickSimSort(A, pPos, B);
  quickSimSort(A+pPos+1, N-pPos-1, B);
}
```

# Result of running Simple Quick Sort

**Editor:**

```
int main() {
  int A[]={23,5,40,2,9,68,55,4,3,1,65,29};
  int B[12];
  quickSimSort(A, 12, B);
  printf("after sorting by quickSimSort \n\t");
  showIArr(A,12);
return 0; }
```

**Shell:**

```
$ make quickSort ; ./quickSort
cc     quickSort.c   -o quickSort
after sorting by quickSimSort
        1 2 3 4 5 9 23 29 40 55 65 68
```

# Iterative Code for Simple Partitioning

**Editor:**

```
int simPartI(int A[], int N, int B[], int p) {
  int i=0, j=-1, k=N; // // by clause (1)
  for (;;) {
    if (i >= N) {
      B[j+1]=p;
      return (j+1); // by clause (2)
    } else if (A[i] < p) {
      B[j+1] = A[i]; // by clause (3)
      i++; j++;
    } else if (A[i] > p) {
      B[k-1] = A[i]; // by clause (4)
      i++; k--;
    } else i++; // by clause (5)
  }
}
```

# Code & Results for Simple Quick Sort

**Editor:**

```
void quickSimSort(int A[], int N, int B[]) {
  int pPos;

  if (N<=1) return;
  pPos = simPartI (A, N, B, A[0]);
  copyBack(A, B, N);
  quickSimSort(A, pPos+1, B);
  quickSimSort(A+pPos+1, N-pPos-1, B);
}
```

**Shell:**

```
$ make quickSort ; ./quickSort
cc     quickSort.c   -o quickSort
after sorting by quickSimSort
        1 2 3 4 5 9 23 29 40 55 65 68
```

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

as 29@h $> 9$

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

as 65@h $> 9$

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 1 | 65 | 29 |

stuck, 23@l $>$ 1@h

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 23 | 65 | 29 |

after interchange

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 23 | 65 | 29 |

as 1@l $<$ 9

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 23 | 65 | 29 |

as 5@l $<$ 9

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 23 | 65 | 29 |

as 23@h $> 9$

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 40 | 2 | 9 | 68 | 55 | 4 | 3 | 23 | 65 | 29 |

stuck, 40@l $>$ 3@h

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 9 | 68 | 55 | 4 | 40 | 23 | 65 | 29 |

after interchange

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 9 | 68 | 55 | 4 | 40 | 23 | 65 | 29 |

as 3@l $<$ 9

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 9 | 68 | 55 | 4 | 40 | 23 | 65 | 29 |

as 2@l $<$ 9

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 9 | 68 | 55 | 4 | 40 | 23 | 65 | 29 |

as 40@h $> 9$

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 9 | 68 | 55 | 4 | 40 | 23 | 65 | 29 |

stuck, 9@l $>$ 4@h

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | 68 | 55 | 9 | 40 | 23 | 65 | 29 |

after interchange

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | 68 | 55 | 9 | 40 | 23 | 65 | 29 |

as 4@I $<$ 9

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | 68 | 55 | 9 | 40 | 23 | 65 | 29 |

stuck, 68@l $>$ 9@h

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array A are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | 9 | 55 | 68 | 40 | 23 | 65 | 29 |

after interchange

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | 9 | 55 | 68 | 40 | 23 | 65 | 29 |

as 68@h $> 9$

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| 1 | 5 | 3 | 2 | 4 | •9• | 55 | 68 | 40 | 23 | 65 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|

as 55@h $> 9$

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | •9• | 55 | 68 | 40 | 23 | 65 | 29 |

stuck, 9@l $>$ 4@h

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | 9 | 55 | 68 | 40 | 23 | 65 | 29 |

as A[l]=A[h]

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# In-place Partitioning Scheme

- Let pivot element be 9
- Elements in array **A** are they are partitioned using the pivot:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 4 | 9 | 55 | 68 | 40 | 23 | 65 | 29 |

as A[l]=A[h], end as $l > h$

- Partitioning now terminates
- Skip over smaller elements on the left: `l++`
- Skip over larger elements on the right: `r--`
- When `A[l] == A[h] == p`, skip from left: `l++`
- Stuck if `A[l]>=p, A[h]<=p, A[l]!=A[h]`: interchange
- Position of pivot element is (l-1) or (h) at termination

**Invariant** Elements to the left of the pivot are no smaller

**Invariant** Elements to the right of the pivot are larger

**Invariant** Comparison of elements in between not known

# Code for Recursive In-place Partitioning

**Editor:**

```
int partitionR(int A[], int N, int l, int h, int p) {
  if (l > h) return (l-1);
  else if (A[l] < p) // skip smaller
    return partitionR(A, N, l+1, h, p);
  else if (A[h] > p) // skip larger
    return partitionR(A, N, l, h-1, p);
  else if (A[l]==A[h]) // A[l]==A[h]==p
  // only skip copy of p in the left part
    return partitionR(A, N, l+1, h, p);
  else { // stuck: A[l]>=p, A[h]<=p, A[l]!=A[h]
    int t=A[l]; A[l]=A[h]; A[h]=t;
    // after interchange: A[l]<p, A[h]>=p
    // if A[l] was p, then it is moved right
    return partitionR(A, N, l, h, p);
  }
}
```

# Code for Iterative In-place Partitioning

**Editor:**

```
int partitionI(int A[], int N, int p) {
  int l=0, h=N-1;
  for (;;) {
    if (l > h) return h; // instead of (l-1)
    else if (A[l] < p) l++;
    else if (A[h] > p) h--;
    else if (A[l]==A[h]) l++;
    else {
      int t=A[l]; A[l]=A[h]; A[h]=t;
    }
  }
}
```

# Code for Quicksort with In-place Partitioning

**Editor:**

```
void quickSort(int A[], int N) {
  int pPos;

  if (N<=1) return;
  pPos = partitionR (A, N, 0, N-1, A[0]);
  quickSort(A, pPos);
  quickSort(A+pPos+1, N-pPos-1);
}
```

**Editor:**

```
void quickSort(int A[], int N) {
  int pPos;

  if (N<=1) return;
  pPos = partitionI (A, N, A[0]);
  quickSort(A, pPos);
  quickSort(A+pPos+1, N-pPos-1);
}
```

# Testing Quick Sort

**Editor:**

```
int main() {
  int A[]={23,5,40,2,9,68,55,4,3,1,65,29};
  int B[12];
  quickSimSort(A, 12, B);
  printf("after sorting by quickSimSort \n\t");
  showIArr(A,12);

  quickSort(A, 12);
  printf("after sorting by quickSort \n\t");
  showIArr(A,12);
return 0; }
```

# Results of Running Quick Sort

**Shell:**

```
$ make quickSort ; ./quickSort
cc quickSort.c -o quickSort after sorting by quickSimSort
        1 2 3 4 5 9 23 29 40 55 65 68
after sorting by quickSort
        1 2 3 4 5 9 23 29 40 55 65 68
```

# Worst and best cases of complexity of quicksort

**Worst case** Pivot is placed at one of the two ends

- $T(n) = T(n-1) + \Theta(n)$
- $T(0) = \Theta(1)$
- $T(n) = \Theta(n^2)$

**Best case** Pivot is placed in the middle to generates sub-sroblem of the same size

- $T(n) = T(n/2) + T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n)$
- $T(0) = \Theta(1)$
- $T(n) = \Theta(n \lg n)$

**About**

- It is an in-place sorting algorithm
- It is an unstable sorting algorithm – elements of the same value may be re-orded
- Worst case when the pivot element get place at one of the ends
  Happens when the array is already sorted

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k] \{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k] \{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n} \sum_{k=1}^{k=n} \{T(k-1) + T(n-k)\} = cn + \frac{2}{n} \sum_{k=1}^{k=n} T(k-1)$

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k] \{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n} \sum_{k=1}^{k=n} \{T(k-1) + T(n-k)\} = cn + \frac{2}{n} \sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2 \sum_{k=1}^{k=n} T(k-1)$

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k] \{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n} \sum_{k=1}^{k=n} \{T(k-1) + T(n-k)\} = cn + \frac{2}{n} \sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2 \sum_{k=1}^{k=n} T(k-1)$
- $(n-1)T(n-1) = c(n-1)^2 + 2 \sum_{k=1}^{k=n-1} T(k-1)$, substituting $n-1$ for $n$

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k]\,\{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n} \sum_{k=1}^{k=n} \{T(k-1) + T(n-k)\} = cn + \frac{2}{n} \sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2 \sum_{k=1}^{k=n} T(k-1)$
- $(n-1)T(n-1) = c(n-1)^2 + 2 \sum_{k=1}^{k=n-1} T(k-1)$, substituting $n-1$ for $n$
- $nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$, after substraction

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k] \{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n} \sum_{k=1}^{k=n} \{T(k-1) + T(n-k)\} = cn + \frac{2}{n} \sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2 \sum_{k=1}^{k=n} T(k-1)$
- $(n-1)T(n-1) = c(n-1)^2 + 2 \sum_{k=1}^{k=n-1} T(k-1)$, substituting $n-1$ for $n$
- $nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$, after substraction
- $nT(n) = (n+1)T(n-1) + c(2n-1)$

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p=k]\{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n}\sum_{k=1}^{k=n}\{T(k-1) + T(n-k)\} = cn + \frac{2}{n}\sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2\sum_{k=1}^{k=n} T(k-1)$
- $(n-1)T(n-1) = c(n-1)^2 + 2\sum_{k=1}^{k=n-1} T(k-1)$, substituting $n-1$ for $n$
- $nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$, after substraction
- $nT(n) = (n+1)T(n-1) + c(2n-1)$
- $\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n}$

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k] \{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n} \sum_{k=1}^{k=n} \{T(k-1) + T(n-k)\} = cn + \frac{2}{n} \sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2 \sum_{k=1}^{k=n} T(k-1)$
- $(n-1)T(n-1) = c(n-1)^2 + 2 \sum_{k=1}^{k=n-1} T(k-1)$, substituting $n-1$ for $n$
- $nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$, after substraction
- $nT(n) = (n+1)T(n-1) + c(2n-1)$
- $\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n}$
- Let $S(n) = \frac{T(n)}{n+1}$, then $S(0) = \frac{T(0)}{1} = T(0)$ and

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p=k]\,\{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n}\sum_{k=1}^{k=n}\{T(k-1)+T(n-k)\} = cn + \frac{2}{n}\sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2\sum_{k=1}^{k=n} T(k-1)$
- $(n-1)T(n-1) = c(n-1)^2 + 2\sum_{k=1}^{k=n-1} T(k-1)$, substituting $n-1$ for $n$
- $nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$, after substraction
- $nT(n) = (n+1)T(n-1) + c(2n-1)$
- $\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n}$
- Let $S(n) = \frac{T(n)}{n+1}$, then $S(0) = \frac{T(0)}{1} = T(0)$ and
- $S(n) \leq S(n-1) + \frac{2c}{n} \leq S(n-2) + \frac{2c}{n-1} + \frac{2c}{n} \leq T(0) + \left(\frac{2c}{1} + \ldots + \frac{2c}{n}\right) = T(0) + 2cH_n$

# Average case complexity of quicksort

- Pivot may be anywhere with a uniform distribution
- $T(n) = \sum_{k=1}^{k=n} \Pr[p = k] \{cn + T(k-1) + T(n-k)\}$ with $T(0)$ as a constant
- $T(n) = cn + \frac{1}{n} \sum_{k=1}^{k=n} \{T(k-1) + T(n-k)\} = cn + \frac{2}{n} \sum_{k=1}^{k=n} T(k-1)$
- $nT(n) = cn^2 + 2 \sum_{k=1}^{k=n} T(k-1)$
- $(n-1)T(n-1) = c(n-1)^2 + 2 \sum_{k=1}^{k=n-1} T(k-1)$, substituting $n-1$ for $n$
- $nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$, after substraction
- $nT(n) = (n+1)T(n-1) + c(2n-1)$
- $\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n}$
- Let $S(n) = \frac{T(n)}{n+1}$, then $S(0) = \frac{T(0)}{1} = T(0)$ and
- $S(n) \leq S(n-1) + \frac{2c}{n} \leq S(n-2) + \frac{2c}{n-1} + \frac{2c}{n} \leq T(0) + \left(\frac{2c}{1} + \ldots + \frac{2c}{n}\right) = T(0) + 2cH_n$
- Thus, $T(n) \leq (n+1)(T(0) + 2cH_n) \leq (n+1)T(0) + 2c(n+1)\lg(n+1) \in O(n \lg n)$

# Upper bound on harmonic series

- Consider $H_n$ where $n = 2^k - 1$
- Now,

$$
\begin{aligned}
H_n &= \underbrace{\frac{1}{1}} + \underbrace{\frac{1}{2} + \frac{1}{3}} + \underbrace{\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}} + \ldots + \underbrace{\frac{1}{\frac{n+1}{2}} + \ldots + \frac{1}{n-1} + \frac{1}{n}} \\
&\le \underbrace{\frac{1}{1}}_{1} + \underbrace{\frac{1}{2} + \frac{1}{2}}_{2} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}_{4} + \ldots + \underbrace{\frac{1}{\frac{n+1}{2}} + \ldots + \frac{1}{\frac{n+1}{2}} + \frac{1}{\frac{n+1}{2}}}_{2^{k-1}} \\
&= \underbrace{1}_{2^0} + \underbrace{1}_{2^1} + \underbrace{1}_{2^2} + \ldots + \underbrace{1}_{2^{k-1}} \\
&= k
\end{aligned}
$$

- Total number of terms $n = 1 + 2 + \ldots + 2^{k-1} = 2^k - 1$
- Also, $2^{k-1} = \frac{n+1}{2}$ and $\frac{1}{2^{k-1}} = \frac{1}{\frac{n+1}{2}}$
- Thus, $k = \lg(n+1)$ and $H_n \le k = \lg(n+1)$

# Quicksort with In-place Partitioning – Showing Details

**Editor:**

```
void quickSort(int A[], int N) {
  int pPos; int p;

  if (N<=1) return;
  printf("before partition: "); showIArr(A,N);
  pPos = partitionI (A, N, p=A[0]);
  printf(" after pPos =%3d: ", pPos);
  showIArr(A,N); printf("\n");
  quickSort(A, pPos);
  quickSort(A+pPos+1, N-pPos-1);
}
```

# A Detailed Run of Quicksort

## Shell:

```
$ make quickSort ; ./quickSort
cc     quickSort.c   -o quickSort
before partition: 23 5 40 2 23 9 68 55 4 3 1 65 29 23
 after pPos =  8: 23 5 1 2 23 9 3 4 23 55 68 65 29 40

before partition: 23 5 1 2 23 9 3 4
 after pPos =  7: 4 5 1 2 23 9 3 23

before partition: 4 5 1 2 23 9 3
 after pPos =  3: 3 2 1 4 23 9 5

before partition: 3 2 1
 after pPos =  2: 1 2 3

before partition: 1 2
 after pPos =  0: 1 2
```

# A Detailed Run of Quicksort (Contd.)

**Shell:**

```
before partition: 23 9 5
 after pPos =  2: 5 9 23

before partition: 5 9
 after pPos =  0: 5 9

before partition: 55 68 65 29 40
 after pPos =  2: 40 29 55 65 68

before partition: 40 29
 after pPos =  1: 29 40

before partition: 65 68
 after pPos =  0: 65 68

after sorting by quickSort
        1 2 3 4 5 9 23 23 23 29 40 55 65 68
```

# Faulty Partitioning

**Editor:**

```
int partitionI(int A[], int N, int p) {
  int l=0, h=N-1;
  for (;;) {
    if (l > h) return h;
    else if (A[l] <= p) l++;
    else if (A[h] > p) h--;
    else {
      int t=A[l]; A[l]=A[h]; A[h]=t;
    }
  }
}
```

# Details of a Faulty Run of Quicksort

**Shell:**

```
$ make quickSort ; ./quickSort
cc      quickSort.c   -o quickSort
before partition: 23 5 40 2 23 9 68 55 4 3 1 65 29 23
 after pPos =   8: 23 5 23 2 23 9 1 3 4 55 68 65 29 40

before partition: 23 5 23 2 23 9 1 3
 after pPos =   7: 23 5 23 2 23 9 1 3

before partition: 23 5 23 2 23 9 1
 after pPos =   6: 23 5 23 2 23 9 1

before partition: 23 5 23 2 23 9
 after pPos =   5: 23 5 23 2 23 9

before partition: 23 5 23 2 23
 after pPos =   4: 23 5 23 2 23

before partition: 23 5 23 2
 after pPos =   3: 23 5 23 2
```

# Details of a Faulty Run of Quicksort (Contd.)

## Editor:

```
before partition: 23 5 23
 after pPos =  2: 23 5 23

before partition: 23 5
 after pPos =  1: 23 5

before partition: 55 68 65 29 40
 after pPos =  2: 55 40 29 65 68

before partition: 55 40
 after pPos =  1: 55 40

before partition: 65 68
 after pPos =  0: 65 68

after sorting by quickSort
        23 5 23 2 23 9 1 3 4 55 40 29 65 68
```