Sort Algorithms

Inst. Nguyễn Minh Huy

Contents

- Basic concepts.
- Quadratic algorithms.
- Logarithmic algorithms.

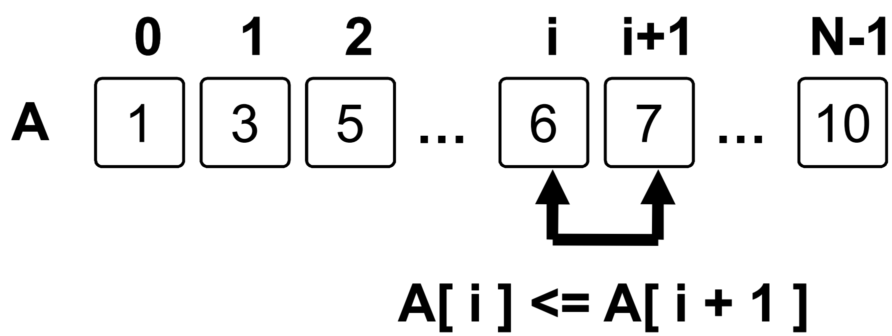
Contents

- **Basic concepts.**
- Quadratic algorithms.
- Logarithmic algorithms.

Basic concepts

- Array sorting problem:
  - Given an array A size N.
  - A is sorted in ascending order…

    ⇔ Adjacent pair $A_i <= A_{i+1}$



  - Brute force algorithm: O(N!).
  - *Reference: www.sorting-algorithms.com*

Basic concepts

- Algorithm analysis:

Families	Algorithms	Complexity			Space
		Best	Worst	Average	
Quadratic Comparison	Bubble sort	N	N^2	N^2	1
	Selection sort	N^2	N^2	N^2	1
	Insertion sort	N	N^2	N^2	1
Logarithmic Comparison	Merge sort	N logN	N logN	N logN	N
	Quick sort	N logN	N^2	N logN	logN
	Heap sort	N logN	N logN	N logN	1
Counting	Radix sort	K N	K N	K N	K + N

  - In-place sort: no extra temporary memory.
  - Stable sort: keep relative orders of equal elements.

Contents

- Basic concepts.
- **Quadratic algorithms.**
- Logarithmic algorithms.

Quadratic algorithms

- Bubble sort idea:
  - "Bubble" up lighter element.
    - A[i] is lighter ⇔ A[i] < A[i - 1].
  - A bubble up iteration:
    - For element A[i] from last to first.
    - If A[i] is lighter => swap up.
  - Do bubble iteration until no swap.

Iteration #1

A	4	1	5	8	2	swap
	4	1	5	2	8	swap
	4	1	2	5	8	swap
	1	4	2	5	8	

Iteration #2

A	1	4	2	5	8	swap
	1	2	4	5	8	

Iteration #3

A	1	2	4	5	8

Quadratic algorithms

- Bubble sort algorithm:

```
// Original version.
bubbleSort( array A, size N ) {
    do {
        isSwap = bubbleUp( A, N );
    } loop isSwap
}
```

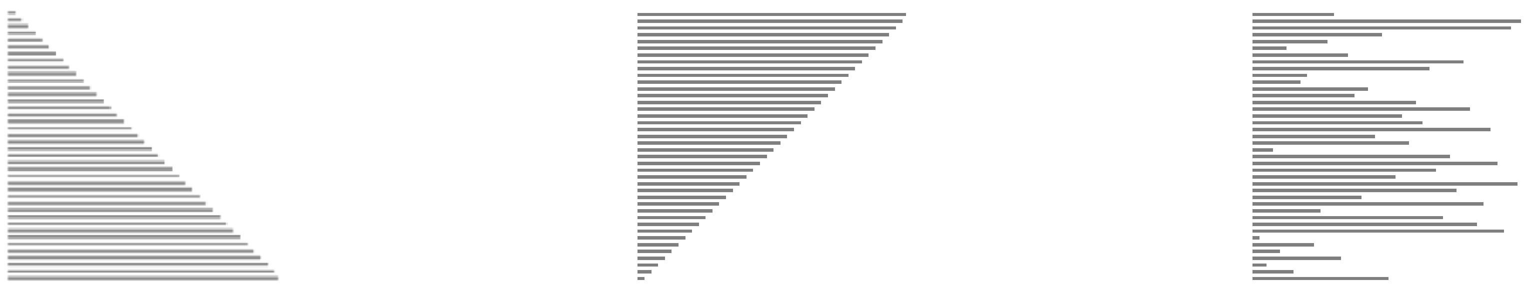
-
```
// Improved version.
bubbleSort2( array A, size N ) {
    do  {
            isSwap = bubbleUp( A, N, from last to i );
            i = i + 1;
    } loop swapFlag
}
```

Lightest element stable at each iteration

Quadratic algorithms

- Bubble sort analysis:

Scenario	When occur?	Complexity
Best-case	Array is already sorted	O(n)
Worst-case	Array is in reversed order	O(n^2)
Average-case	Array is in random order	O(n^2)



Best-case Worst-case Average-case

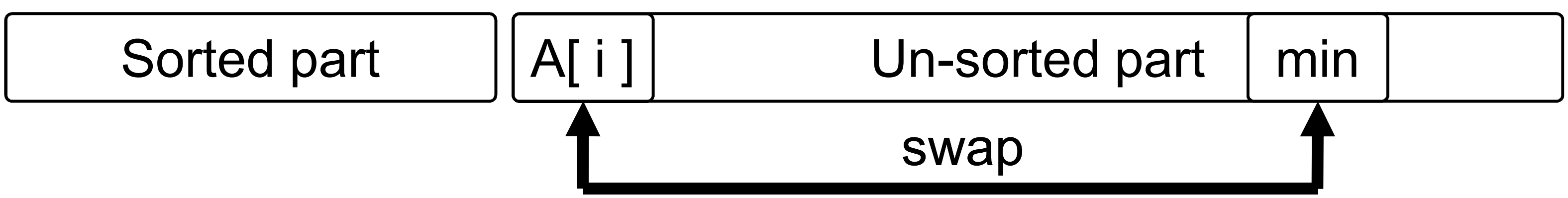
- A stable and in-place sort algorithm.
  - ➔ Space complexity: O(1).

➔ Slow and rarely used!

Quadratic algorithms

- Selection sort idea:
  - Select min element, move to first.
  - Repeat with the remaining elements.
  - Repeat i^{th}:
    - Select min from i^{th}.
    - Move min to i^{th} place.

Repeat i^{th}



Quadratic algorithms

- Selection sort algorithm:

```
// Original version…
selectionSort( array A, size N ) {
    for i from 0 to N - 1 {
        minpos = findMin( A, N, from i );
        swap( A[ i ],  A[ minpos ] );
    }
}

// Improved version…
selectionSort( array A, size N ) {
    for i from 0 to N - 2 {
        minpos = findMin( A, N, from i );

        if ( minpos != i )
            swap( A[ i ],  A[ minpos ] );
    }
}
```

Quadratic algorithms

- Selection sort analysis:

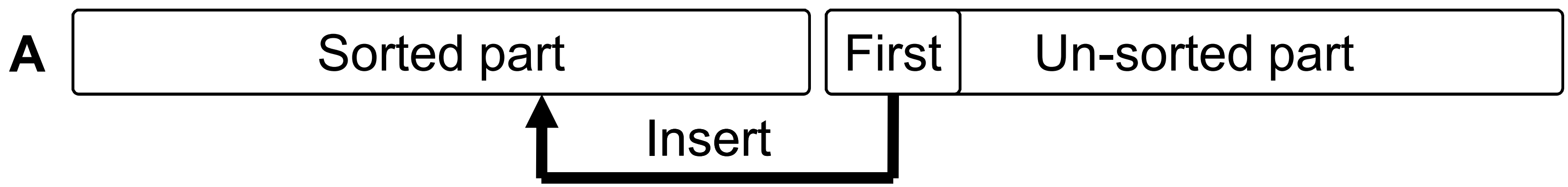
Scenario	When occur?	Complexity
Best-case	Array is already sorted	$O(n^2)$
Worst-case	Array is in reversed order	$O(n^2)$
Average-case	Array is in random order	$O(n^2)$

  - A stable and in-place sort algorithm.
    ➔ Space complexity: O(1).
  - Faster than bubble sort on average-case.
  - Simple way to sort **small** array.

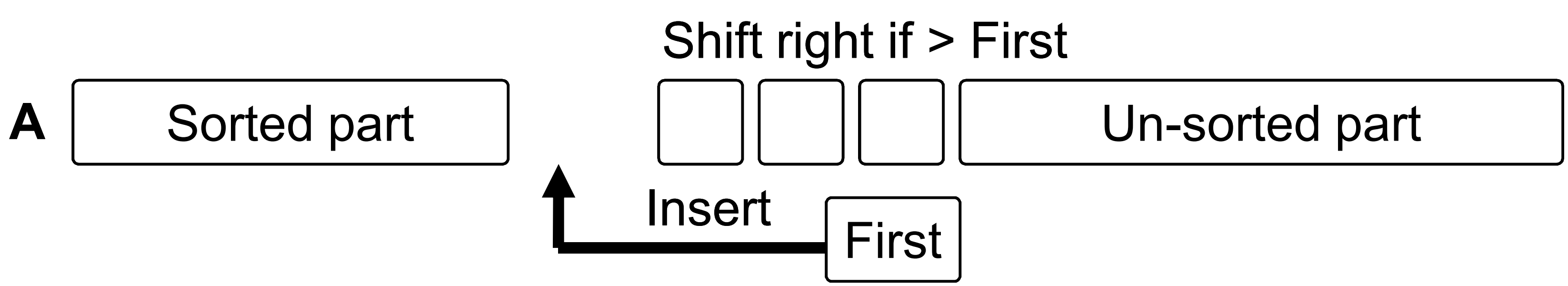
Quadratic algorithms

- Insertion sort idea:
  - Split array into sorted part & un-sorted part.
    - At beginning, sorted part is the first element.
  - Get first element of un-sorted part

    ➔ Insert backward into sorted part (keep order).



  - How to insert into sorted part (keep order)?



Quadratic algorithms

- Insertion sort algorithm:

```
insertBackward( array A, size N, index i ) {
        temp = A[ i ];
        for each A[ j ] befor A[ i ]
            if ( A[ j ] > temp )
                    Shift A[ j ] forward;
            else {
                    Insert temp after A[ j ];
                    Stop;
            }
}

insertionSort( array A, size N ) {
    for each A[ i ] in un-sorted part {         // begin from 2nd element.
        insertBackward( A, N, i );
    }
}
```

Quadratic algorithms

- Insertion sort analysis:

Scenario	When occur?	Complexity
Best-case	Array is nearly sorted	O(n)
Worst-case	Array is in reversed order	O(n^2)
Average-case	Array is in random order	O(n^2)

  - A stable and in-place sort algorithm.

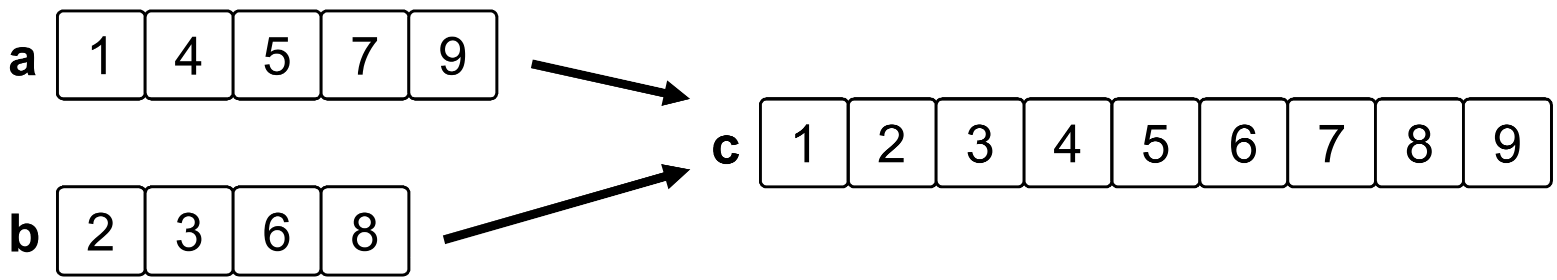
    ➔ Space complexity: O(1).
  - Faster than bubble & selection sort on average-case.
  - Efficient way to sort:
    - **Small** array (< 100 elements).
    - **Nearly sorted** array.

Contents

- Basic concepts.
- Quadratic algorithms.
- **Logarithmic algorithms.**

Logarithmic algorithms

- Merge array problem:
  - Merge two sorted arrays into sorted one?



➢ For each position in c, copy element from a or b?

```
// i, ia, ib current positions of c, a, b.
if ( a[ ia ] < b[ ib ] )
      c[ i++ ] = a[ ia++ ];
else
      c[ i++ ] = b[ ib++ ];
```

➔ c[i++] = (a[ia] < b[ib]) ? a[ia++] : b[ib++];

Logarithmic algorithms

- Merge array problem:

mergeArray(array **A**, size **NA**, array **B**, size **NB**, array **C**)
{
    set up i, ia, ib start positions of A, B, C.

    loop if **A, B are still not end**
        C[i++] = (A[ia] < B[ib]) ? A[ia++] : B[ib++];

    loop if **A is still not end**
        Copy element from A to C.

    loop if **B is still not end**
        Copy element from B to C.
}

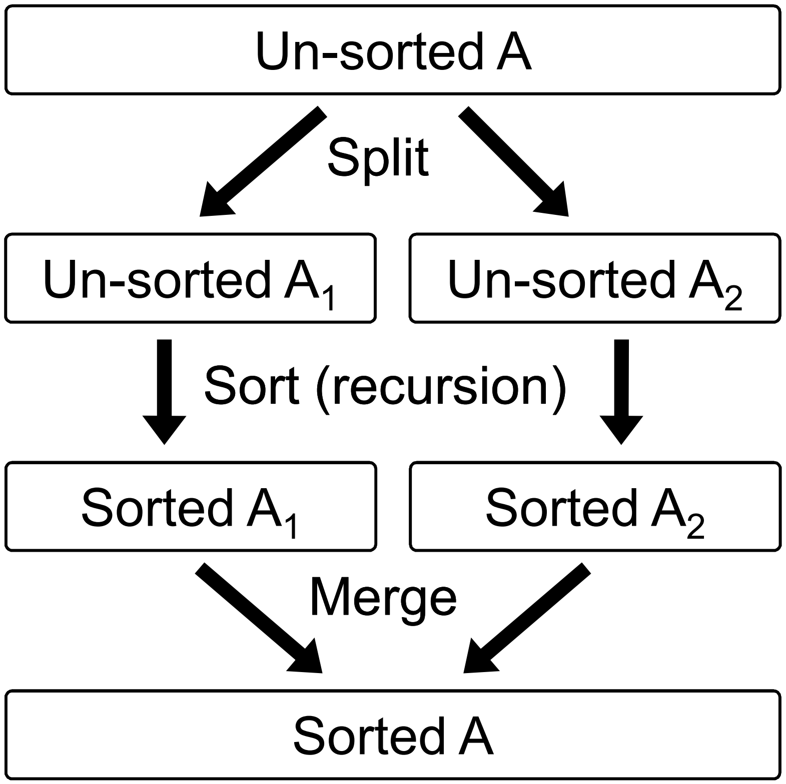
Logarithmic algorithms

- Merge sort idea:

// Divide-and-conquer technique.

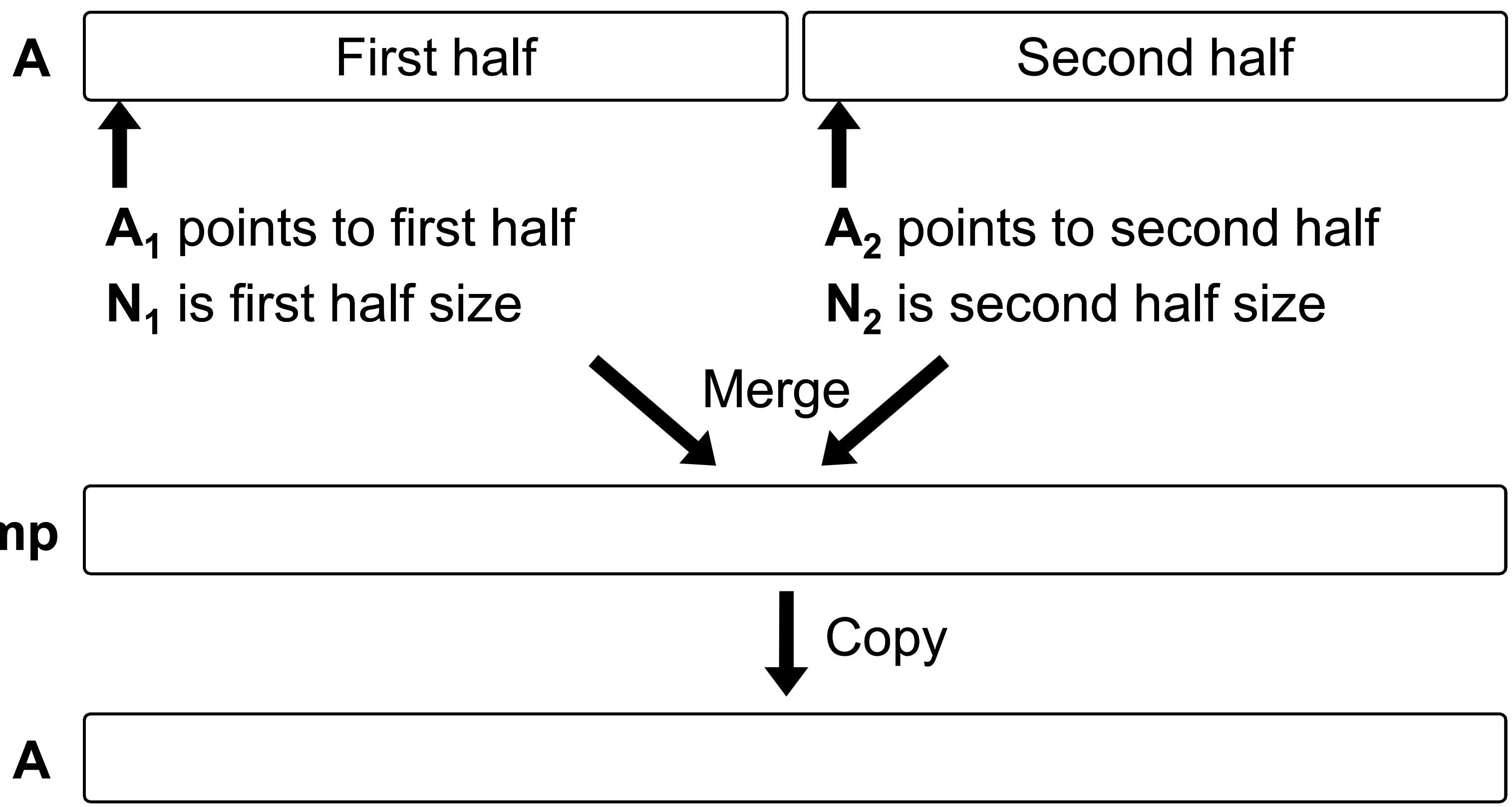
mergeSort(array **A**, size **N**)
{
    if (**A has one element**)
        Stop;
    else
    {
        Split **A** into $\mathbf{A_1}$ (size $\mathbf{N_1}$) and $\mathbf{A_2}$ (size $\mathbf{N_2}$);
        mergeSort($\mathbf{A_1}$, $\mathbf{N_1}$);
        mergeSort($\mathbf{A_2}$, $\mathbf{N_2}$);

        mergeArray($\mathbf{A_1}$, $\mathbf{N_1}$, $\mathbf{A_2}$, $\mathbf{N_2}$, **A**);
    }
}



Logarithmic algorithms

- Merge sort improvement:
  - Splitting in same array:



Logarithmic algorithms

- Merge sort improvement:
  - Cut off splitting & use insertion sort.

mergeSort2(array **A**, size **N**)
{
    if (**N small enough**)
        insertionSort(A, N);
    else
    {
        Split **A** into $\mathbf{A_1}$ (size $\mathbf{N_1}$) and $\mathbf{A_2}$ (size $\mathbf{N_2}$);
        mergeSort2($\mathbf{A_1}$, $\mathbf{N_1}$);
        mergeSort2($\mathbf{A_2}$, $\mathbf{N_2}$);

        mergeArray($\mathbf{A_1}$, $\mathbf{N_1}$, $\mathbf{A_2}$, $\mathbf{N_2}$, **temp**);
        copyArray(**temp**, **A**);
    }
}

insertion sort is efficient for small array!!

Logarithmic algorithms

- Merge sort analysis:

Scenario	When occur?	Complexity
Best-case	Array is already sorted	O(n*log(n))
Worst-case	Array is in reversed order	O(n*log(n))
Average-case	Array is in random order	O(n*log(n))

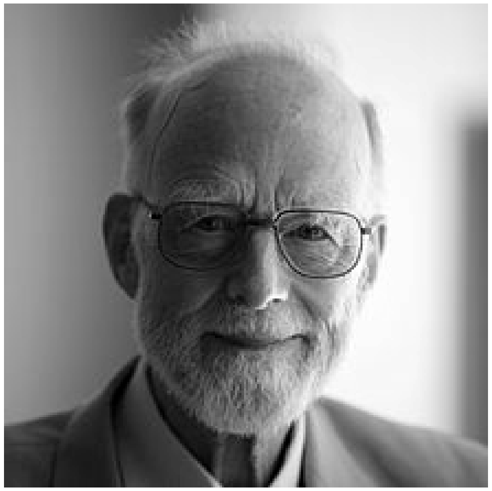
N	Insertion sort swap	Merge sort swap
10	~ 100	~ 40
1,000	~ 1,000,000	~ 10,000
100,000	~ 10,000,000,000	~ 1,700,000

  - Not an in-place sort algorithm.
    - ➔ Need a temporary array.

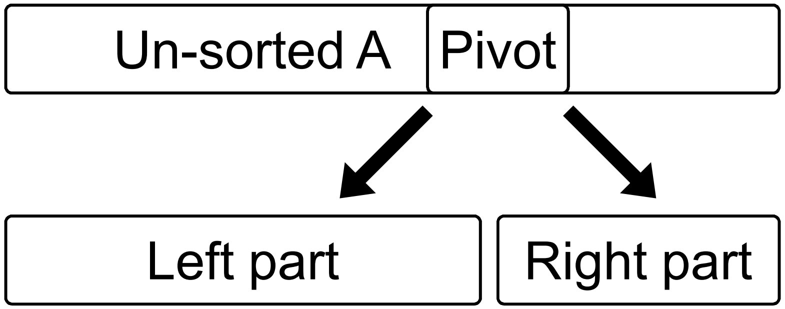
➔ Good for sorting **LARGE** array with **ENOUGH MEMORY**.

Logarithmic algorithms

- Quicksort overview:
  - Developed by Tony Hoare, 1961.
  - An O(n*log(n)) sort algorithm.
  - Three times faster than Merge sort.
  - Standard sort algorithm for libraries.



- Quicksort idea:
  - Partitioning array:
    - Given a pivot.
    - Left part <= pivot.
    - Right part >= pivot.

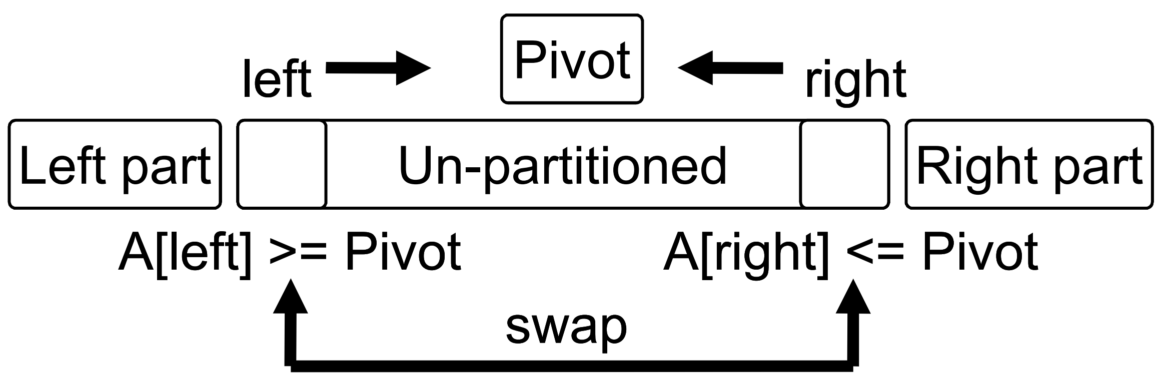


Logarithmic algorithms

- Partitioning array:

partitionArray(array **A**, from **l**, to **r**, pivot **P**): partition position **K**

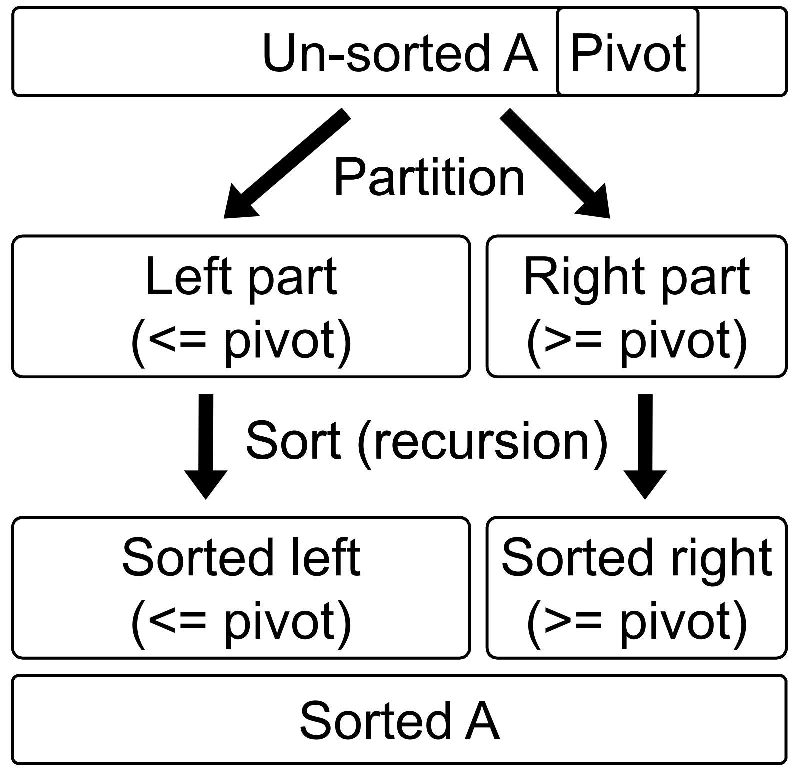
```
{
    loop ( l < r )
    {
        l = find from left, first element >= P
        r = find from right, first element <= P

        if ( l  < r ) {
            swap( A[ l ], A[ r ] );
            l++;
            r--;
        }
    }
    K = r;
}
```



Logarithmic algorithms

- Quicksort algorithm:

```
// Divide-and-conquer technique.
quickSort( array A, from l, to r )
{
    if ( sorting range is one )
        return;
    else
    {
        pivot = select pivot from A;
        pos = partitionArray( A, l, r, pivot );
        quickSort( A, l, pos );
        quickSort( A, pos + 1, r );
    }
}
```



Logarithmic algorithms

- Quicksort improvement:
  - Stop prematurely & use insertion sort:

```
// A is JUST NEARLY SORTED.
qSort( array A, from l, to r )
{
    if ( range is small )
        stop algorithm;
    else
    {
        pivot = select pivot from A;
        pos = partitionArray( A, l, r, pivot );
        qSort( A, l, pos );
        qSort( A, pos + 1, r );
    }
}
```

Stop prematurely!!

```
// Cover function.
quickSort3( array A, size N )
{
    qSort( A, 0, N - 1 );

    // A is nearly sorted.
    // Perform insertion sort.
    insertionSort( A, N );
}
```

Insertion sort is very efficient for nearly sorted array!!

Logarithmic algorithms

- Quicksort analysis:

Scenario	When occur?	Complexity
Best-case	Array is already sorted	O(n*log(n))
Worst-case	Array is in reversed order	O(n^2)
Average-case	Array is in random order	O(n*log(n))

  - An in-place sort algorithm but use recursion.
    ➔ Space complexity: O(logN).
  - Not a stable sort.
  - Average-case is closer to best-case than worst-case.
  ➔ Faster than most of O(NlogN) algorithms.
  ➔ Standard sort algorithm for libraries.

Summary

- Bubble sort:
  - Average-case complexity: O(n^2).
  - Storage space: in-place.
- Selection sort:
  - Average-case complexity: O(n^2).
  - Storage space: in-place.
  - Stable sort algorithm.
- Insertion sort:
  - Average-case complexity: O(n^2).
  - Storage space: in-place.
  - Efficient for small or nearly sorted array.

Summary

- Merge sort:
  - Average-case complexity: O(n*log(n)).
  - Storage space: need temporary array.
  - With enough memory, good for large array.
- Merge sort improvement:
  - Splitting in same array: use pointers.
  - Cut off & use selection sort.
  - Stop prematurely & use insertion sort.

Summary

- Quicksort:
  - Average-case complexity: O(n*log(n)).
  - Worst-case complexity (rarely): O(n^2).
  - Storage space: in-place.
  - Fast and commonly used in libraries.
- Quicksort improvement:
  - Stop prematurely & use insertion sort.

Practice

- ## Practice 5.1:

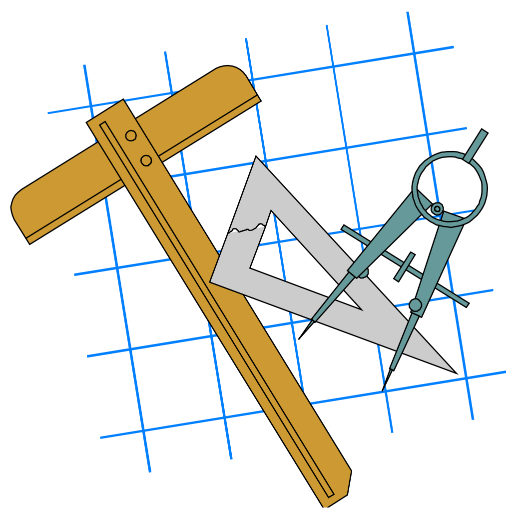
  Practice sort algorithms in this slides on the following arrays:

  a) A = { 8 7 2 3 1 4 6 5 }.

  b) A = { 3 5 1 2 8 7 4 6 }.

  c) A = { 8 7 6 5 4 3 2 1 }.

  d) A = { 1 2 3 4 5 6 7 8 }.

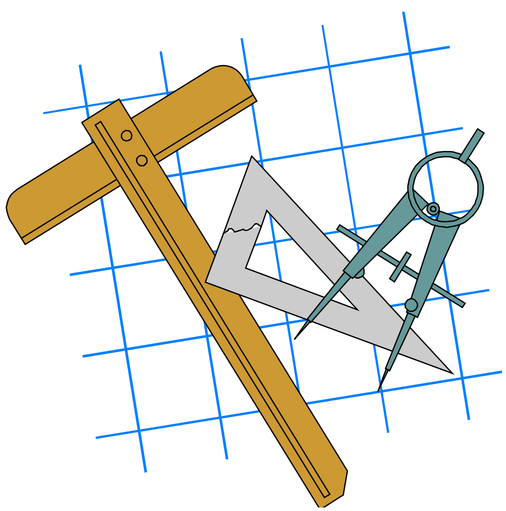


Practice

- Practice 5.2:

  Construct class **Array** (of integer) and provide it with the following sort methods:

  - Bubble sort.
  - Selection sort.
  - Insertion sort.
  - Merge sort.
  - Quick sort.



Practice

- Practice 5.3:
  a) Implement **Insertion sort** on **Singly Linked List**.
  b) Implement **Merge sort** on **Singly Linked List**.