Analysis of Algorithms

Coming four lectures:

- A (gentle) Introduction to Complexity Theory.
- Explore Sorting algorithms as an example.
- Explore Search algorithms
- Introduce Hashing as a way to make things *fast*.

$\Rightarrow$ All important for final coursework.

Motivation

There are often **many** algorithms to solve **one** problem.

- Which algorithm is best?
- Timing an algorithm gives us an answer for one specific computer, one specific data set, one specific compiler...

$\Rightarrow$ Need to reason about it from a more fundamental level

Aim:

- Learn how to compare algorithms.
- Develop an intuition for “O” notation. (formal intro in COMS21101)
- How to informally use it.
- What it means

Sorting Algorithms

Motivation

- **Problem:** Sorting an initially unordered collection of values
- **Result:** Ordered collection of values
- **Solution: Devise a suitable sorting algorithm**

Finding a sorting technique that matches a problem is not easy.
(more in Software Engineering course)

Aim here:

- Get intuitive grasp on how various sorting techniques work.
  - “naive”, and “divide and conquer”
- Familiarise with performance characteristics of them.

(• Concentrate on **comparison-based** sorting algorithms.)

BubbleSort

- Notoriously **slow**, but conceptually **simplest** sorting algorithm.

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

Repeat

1. Scan `A`, compare adjacent values `A[i]`, `A[i+1]`
2. If `A[i]` $>$ `A[i+1]` then exchange `A[i]` and `A[i+1]`

Until no further exchange was necessary

- In each pass:
  - Biggest value “bubbles up” to the “top” of `A`
  - Item(s) at the end of `A` are sorted (one per pass)
  - Smallest value moves one position towards the “bottom” of `A`

BubbleSort: In Action (I)

Let `A = 18 26 3 9 34 5 21 14 6`

Pass 1:

```
A = (18 26) 3 9 34 5 21 14 6     ok
A = 18 (26 3) 9 34 5 21 14 6     ex
A = 18 3 (26 9) 34 5 21 14 6     ex
A = 18 3 9 (26 34) 5 21 14 6     ok
A = 18 3 9 26 (34 5) 21 14 6     ex
A = 18 3 9 26 5 (34 21) 14 6     ex
A = 18 3 9 26 5 21 (34 14) 6     ex
A = 18 3 9 26 5 21 14 (34 6)     ex
A = 18 3 9 26 5 21 14 6  34
```

Comparisons are marked in brackets.

Sublist containing last value `34` is now sorted.

BubbleSort: In Action (II)

Pass 2:

```
A = (18 3) 9 26 5 21 14 6  34   ex
A = 3 (18 9) 26 5 21 14 6  34   ex
A = 3 9 (18 26) 5 21 14 6  34   ok
A = 3 9 18 (26 5) 21 14 6  34   ex
A = 3 9 18 5 (26 21) 14 6  34   ex
A = 3 9 18 5 21 (26 14) 6  34   ex
A = 3 9 18 5 21 14 (26 6)  34   ex
A = 3 9 18 5 21 14 6 (26  34)   ok
A = 3 9 18 5 21 14 6  26 34
```

Sublist containing last two values `26 34` is now sorted.

BubbleSort: In Action (III)

Pass 3:

```
A = (3  9)  18  5  21  14  6     26  34    ok
A = 3  (9  18)  5  21  14  6     26  34    ex
A = 3  9  (18  5)  21  14  6     26  34    ex
A = 3  9  5  (18  21)  14  6     26  34    ok
A = 3  9  5  18  (21  14)  6     26  34    ex
A = 3  9  5  18  14  (21  6)     26  34    ex
A = 3  9  5  18  14  6  (21      26)  34   ok
A = 3  9  5  18  14  6  21      (26  34)   ok
A = 3  9  5  18  14  6   21  26  34
```

Sublist containing last three values **21 26 34** is now sorted.

$\Rightarrow$ After pass $\mathbf{i}$, the last $\mathbf{i}$ values are sorted. Save comparisons!

BubbleSort: In Action (III)

(After) Pass 4:

`A = 3 5 9 14 6 | 18 21 26 34`

(After) Pass 5:

`A = 3 5 9 6 | 14 18 21 26 34`

(After) Pass 6:

`A = 3 5 6 | 9 14 18 21 26 34`

(After) Pass 7:

`A = 3 5 6 | 9 14 18 21 26 34`

No exchange made $\Rightarrow$ done!

BubbleSort: Analysis (I)

Comparisons (on average)

- For 9 values: 8 in first pass, 7 in second pass, 6 in third pass, ...
  8+7+6+5+4+3+2+1 = 36 (comparisons)

- For $\mathbf{n}$ values: $(\mathbf{n}-\mathbf{1})$ in first pass, $(\mathbf{n}-\mathbf{2})$ in second pass, ...
  $(\mathbf{n}-\mathbf{1}) + (\mathbf{n}-\mathbf{2}) + ... + 1 = \mathbf{n} * (\mathbf{n}-\mathbf{1})/\mathbf{2}$ (comparisons)

- Number of comparisons: $\mathbf{n} * (\mathbf{n}-\mathbf{1})/\mathbf{2} = (\mathbf{n}^{\mathbf{2}} - \mathbf{n})/\mathbf{2}$

Best case: Input already sorted

$\Rightarrow$ one pass, $\mathbf{n}-\mathbf{1}$ comparisons, no exchange

BubbleSort: Analysis (II)

Exchanges (on average)

- Fewer exchanges than comparisons. (Only when needed.)
- On random input data, exchange is needed about half the time.
- **Number of exchanges:** $(n^2 - n)/4 \Rightarrow$ (also) proportional to n^2

BubbleSort runs in time $O(n^2)$ (on average).

Looking at the size of the problem

Variable n represents size of a problem:

- n length of a list, number of nodes in a tree, or items in an array

Summarise number of steps with big-O, assume n is big (very big)

$\Rightarrow O(a\mathbf{n}^2 + b\mathbf{n} + c)$

For big problems, the dominant term in f($\mathbf{n}$) usually accounts for most of f($\mathbf{n}$)'s value, ignore the lesser terms by throwing them out

$\Rightarrow O(a\mathbf{n}^2)$

Neglect differences from measuring in different environments, by throwing out any constant factors

$\Rightarrow O(\mathbf{n}^2)$

What does mean when we say "the running time of alg X is $O(n^2)$"?

- X runs in an amount of time **no greater than** n^2 times some coefficient, provided the problem size is big enough.

Seven Common Complexity Classes

Characterised by an adjective name and their O-notation:

constant	$O(1)$
logarithmic	$O(\log \mathbf{n})$
linear	$O(\mathbf{n})$
n log n	$O(\mathbf{n} \log \mathbf{n})$
quadratic	$O(\mathbf{n}^2)$
cubic	$O(\mathbf{n}^3)$
exponential	$O(2^{\mathbf{n}})$ or $O(10^{\mathbf{n}})$

Note:

- In $O(g(\mathbf{n}))$, we make $g(\mathbf{n})$ as simple as possible.
- Always use the "tightest fit", O-notation gives an upper bound
  - If f($\mathbf{n}$) is O($\mathbf{n}$), then it is also O($\mathbf{n}^2$), O($\mathbf{n}^3$), etc, but **not** O(log $\mathbf{n}$).
- The further down in the table the more dominant the term.

A (hypothetical) Example (I)

Algorithm X, executed on computer that performs 1 step of X in 1 μs.
(Computer executes at 1 Million instructions per second, 1 MIPs.)

Algorithm X stops in f($\mathbf{n}$) microseconds

$f(\mathbf{n})$	$\mathbf{n}=2$	$\mathbf{n}=16$	$\mathbf{n}=256$	$\mathbf{n}=1024$	$\mathbf{n}=1048576$
1	1	1	1	$1.00\cdot10^{0}$	$1.00\cdot10^{0}$
$log_2\mathbf{n}$	1	4	8	$1.00\cdot10^{1}$	$2.00\cdot10^{1}$
$\mathbf{n}$	2	$1.6\cdot10^{1}$	$2.56\cdot10^{2}$	$1.02\cdot10x^{3}$	$1.05\cdot10^{6}$
$\mathbf{n}log_2\mathbf{n}$	2	$6.4\cdot10^{1}$	$2.05\cdot10^{3}$	$1.02\cdot10^{4}$	$2.10\cdot10^{7}$
$\mathbf{n}^2$	4	$2.56\cdot10^{2}$	$6.55\cdot10^{4}$	$1.05\cdot10^{6}$	$1.10\cdot10^{12}$
$\mathbf{n}^3$	8	$4.10\cdot10^{3}$	$1.68\cdot10^{7}$	$1.07\cdot10^{9}$	$1.15\cdot10^{18}$
$2^{\mathbf{n}}$	4	$6.55\cdot10^{4}$	$1.16\cdot10^{77}$	$1.80\cdot10^{308}$	$6.74\cdot10^{315652}$

= There are $3.15\cdot10^{13}$ μs in a year. $(365\cdot24\cdot60\cdot60\cdot1000000)$

Alg complexity $\mathbf{2^n}$, problem size $\mathbf{n}$=256, takes $\mathbf{2^{256}}=3.7\cdot10^{63}$ years!

A (hypothetical) Example (I)

Algorithm X, executed on computer that performs 1 step of X in 1 μs.
(Computer executes at 1 Million instructions per second, 1 MIPs.)

Algorithm X stops in f($\mathbf{n}$) microseconds

$f(\mathbf{n})$	$\mathbf{n} = 2$	$\mathbf{n} = 16$	$\mathbf{n} = 256$	$\mathbf{n} = 1024$	$\mathbf{n} = 1048576$
1	1	1	1	$1.00 \cdot 10^0$	$1.00 \cdot 10^0$
$log_2\mathbf{n}$	1	4	8	$1.00 \cdot 10^1$	$2.00 \cdot 10^1$
$\mathbf{n}$	2	$1.6 \cdot 10^1$	$2.56 \cdot 10^2$	$1.02 \cdot 10x^3$	$1.05 \cdot 10^6$
$\mathbf{n}log_2\mathbf{n}$	2	$6.4 \cdot 10^1$	$2.05 \cdot 10^3$	$1.02 \cdot 10^4$	$2.10 \cdot 10^7$
$\mathbf{n}^2$	4	$2.56 \cdot 10^2$	$6.55 \cdot 10^4$	$1.05 \cdot 10^6$	$1.10 \cdot 10^{12}$
$\mathbf{n}^3$	8	$4.10 \cdot 10^3$	$1.68 \cdot 10^7$	$1.07 \cdot 10^9$	$1.15 \cdot 10^{18}$
$2^\mathbf{n}$	4	$6.55 \cdot 10^4$	$1.16 \cdot 10^{77}$	$1.80 \cdot 10^{308}$	$6.74 \cdot 10^{315652}$

= There are $3.15 \cdot 10^{13}$ μs in a year. $(365 \cdot 24 \cdot 60 \cdot 60 \cdot 1000000)$

Alg complexity $\mathbf{2^n}$, problem size $\mathbf{n}$=256, takes $\mathbf{2^{256}} = 3.7 \cdot 10^{63}$ years!

A (hypothetical) Example (II)

Algorithm X, executed on computer that performs 1 step of X in 1 μ.
(Computer executes at 1 Million instructions per second, 1 MIPs.)

Algorithm X stops in f($\mathbf{n}$) more familiar time units

f($\mathbf{n}$)	$\mathbf{n}$=2	$\mathbf{n}$=16	$\mathbf{n}$=256	$\mathbf{n}$=1024	$\mathbf{n}$=1048576
1	1 μs	1 μs	1 μs	1 μs	1 μs
$\log_2 \mathbf{n}$	1 μs	4 μs	8 μs	10 μs	20 μs
$\mathbf{n}$	2 μs	16 μs	256 μs	1 ms	1 s
$\mathbf{n} \log_2 \mathbf{n}$	2 μs	64 μs	2 ms	10 ms	21 s
$\mathbf{n}^2$	4 μs	26 μs	66 ms	1 s	12 days
$\mathbf{n}^3$	8 μs	4.1 ms	17 s	18 min	366 cntrs
$\mathbf{2}^{\mathbf{n}}$	4 μs	66 ms	10^{63} yrs	10^{295} yrs	10^{315639} yrs

- Sun will burn out in approx. $5 \cdot 10^9$ years!

$\Rightarrow$ The sun will burn out *before your computation finishes...*

Conclusions (so far)

If problem size is small, $\mathbf{n \leq 16}$

- all finish in under a 10th of a second

$\Rightarrow$ complexity class does not matter (too much)

On medium large problems, $\mathbf{n \leq 1024}$

- algs that are no more complex than $\mathbf{n^2}$ finish in 1s or less
- algs as complex as $\mathbf{n^3}$ and above take inconveniently long

For very large problems, $\mathbf{n \leq 1048576}$

- surprising difference between algs of complexity $\mathbf{n \log n}$ and $\mathbf{n^2}$

Exponential algorithms take very long for all but small problems.

Some Algorithms and their Complexity

Average complexity	Sort
$O(\mathbf{n}^2)$	Selection-sort, Insertion-sort, Bubble-sort
$O(\mathbf{n}\log\mathbf{n})$	Quick-Sort Merge-sort, Heap-sort
$O(\log\mathbf{n})$	Binary Chop
$O(\mathbf{1})$	Array Indexing, Hashing
$O(2^n)$	Travelling salesman

Exponential: Not practical unless problem size is small.

- There exist algorithmic problems for which the best solutions *known to date* take exponential running time!
- Travelling salesman problem
- Algorithms for computing moves in games

Constant and Linear Time Algorithms

Constant characterised by O(1) - Why?

- Take no more than a fixed amount of time, no matter how big the problem size.
- Eg, select and print a single random array item `A[i]` in an array `A[0:n-1]`
- Fixed amount of time:
  - for computing a random number `i`
  - to access `A[i]`
  - to print a number

Linear characterised by O($\mathbf{n}$)

- Parsing of a C program: Parsing runs in time proportional to program length (measured in characters)

What O-Notation does NOT tell you

O-notation applies only to problem sizes that are sufficiently large.

Conclusions that hold for large problem sizes, based on O-notation, may not hold for small problem sizes.

For small problems:

- **Don't rely on conclusions based on O-notation!**
- Test several algorithms, i.e. run, measure and select.

In general:

- Avoid exponential algorithms.
- Many *intuitive algs* in high complexity class can be *re-implemented more cleverly* in a lower complexity class.

$\Rightarrow$ **Aim to be well informed about computational complexity classes of algorithms.**

Where to get more information?

Thomas A. Standish

“Data Structures, Algorithms & Software Principles in C”

Addison-Wesley, 1995

Chapter 6: Introduction to Analysis of Algorithms (p. 205 ff)

J. Glenn Brookshear

“Computer Science - An Overview” (sixth edition)

Addison-Wesley, 2000

11.5 Complexity of Problems (p. 513 ff)

A. D. Dewdney **“The New Turing Omnibus”**

Computer Science Press, 1997

Chapter 15: Time and Space Complexity (p. 96 ff)

Taking another point of view

So far: Looked at how long it takes for an alg to produce a solution for a given problem size.

Now inverse question:

How big a problem can we solve within a year (week, day)?

- alg X takes exactly f($\mathbf{n}$) steps to finish a task
- one step takes 1 μs
- $\mathbf{n}$ problem size

How big can $\mathbf{n}$ be if we expect the computation of alg X to terminate in a certain time limit?

A Detailed Example

Say we want to know for 1 year:

1 year = 31,536,000 s = 3.15 * 10^{13} μs

Find the largest $\mathbf{n}$ s.t. f($\mathbf{n}$) $\leq$ 3.15 * 10^{13} μs

Assume f($\mathbf{n}$) = $\mathbf{2^n}$

For X to finish within a year, the largest $\mathbf{n}$ we could handle using one year of computing effort is .....

A Detailed Example

Say we want to know for 1 year:

1 year = 31,536,000 s = 3.15 * 10^{13} μs

Find the largest $\mathbf{n}$ s.t. f($\mathbf{n}$) $\leq$ 3.15 * 10^{13} μs

Assume f($\mathbf{n}$) = $\mathbf{2^n}$

For X to finish within a year, the largest $\mathbf{n}$ we could handle using one year of computing effort is $\mathbf{n}$=44.

A Detailed Example

Say we want to know for 1 year:

1 year = 31,536,000 s = $3.15 * 10^{13}\ \mu$s

Find the largest $\mathbf{n}$ s.t. $f(\mathbf{n}) \leq 3.15 * 10^{13}\ \mu$s

Assume $f(\mathbf{n}) = \mathbf{2^n}$

For X to finish within a year, the largest $\mathbf{n}$ we could handle using one year of computing effort is $\mathbf{n}$=44.

If $f(\mathbf{n}) = \mathbf{10^n}$, then $\mathbf{n} \leq 13$ (in a year)!

- Unrealistic to expect a computer to work reliably for a whole year!

A (hypothetical) Example (III)

Algorithm X, executed on computer that performs one step of X in 1 microsecond.

(Computer executes at 1 Million instructions per second, 1 MIPs.)

Size of largest problem that algorithm X can solve if solution is computed in time $\leq$ t at 1 microsecond per step

# steps	t = 1 min	t = 1 h	t = 1 day	t = 1 wk	t = 1 yr
n	10^7	10^9	10^{10}	10^{11}	10^{13}
$n \log_2 n$	10^6	10^8	10^9	10^{10}	10^{11}
n^2	10^3	10^4	10^5	10^5	10^6
n^3	10^2	10^3	10^3	10^3	10^4
2^n	25	31	36	39	44
10^n	7	9	10	11	13

$\Rightarrow$ Algs with exponential complexity:

- Not practical for very large problems!

Divide and Conquer Sorting Methods (I)

More efficient than BubbleSort (on average).

- **MergeSort** and **QuickSort** are "divide and conquer" methods

Abstract strategy:

1. Divide a sequence of n values into two subsequences.
2. Sort these two subsequences.
3. Combine two sorted subsequences into single sorted result.

Divide and Conquer Methods (II)

Draft for code to sort an array `A[m:n]`:

```
void Sort(Array A, int m, int n) {
    if (Array A[m:n] contains more than one item) {
        Divide A[m:n] into subarrays A[m:i] and A[i+1:n]
        Sort subarray A[m:i]
        Sort subarray A[i+1:n]
        Combine sorted subarrays to sorted version of original
    } else { /* one-element array */
        do nothing because the array is already sorted
    }
}
```

MergeSort

Refinement of general *divide and conquer* strategy

To sort **`n`** values, held in array **`A[0:n-1]`**, in ascending order:

1. Divide sequence **`A`** into two halves, left **`L`** and right **`R`**.
2. MergeSort the left sequence **`L`**.
3. MergeSort the right sequence **`R`**.
4. Merge sorted left and right sequence into sorted result **`S`**.

- Use MergeSort (recursively) to sort both halves.
- One element sequences are already sorted.
- The heart of MergeSort is the **Merge** step.

The Merge Step in MergeSort

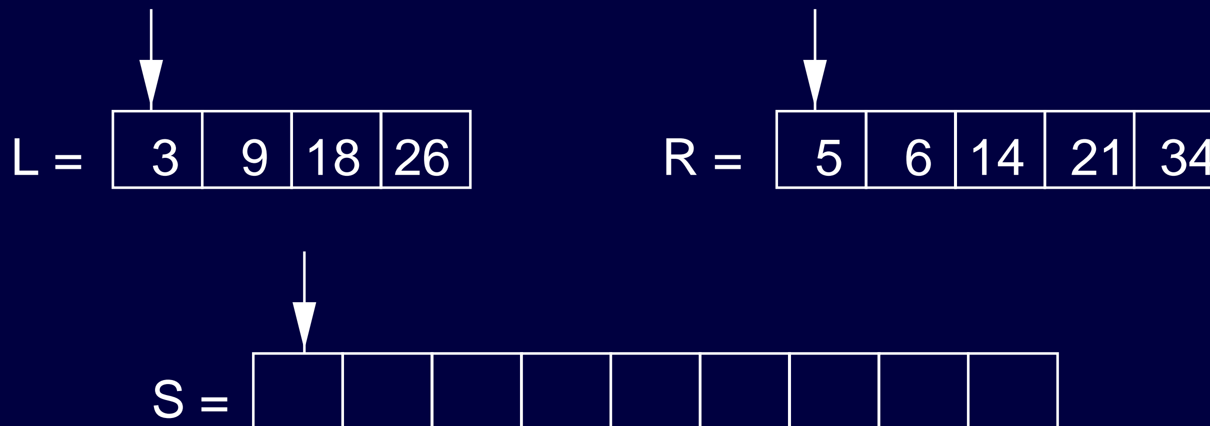
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

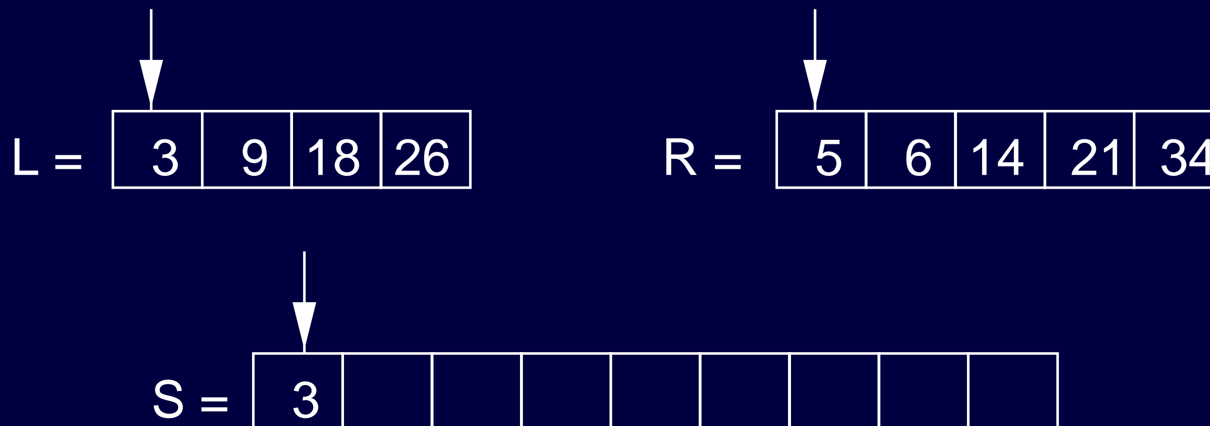
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

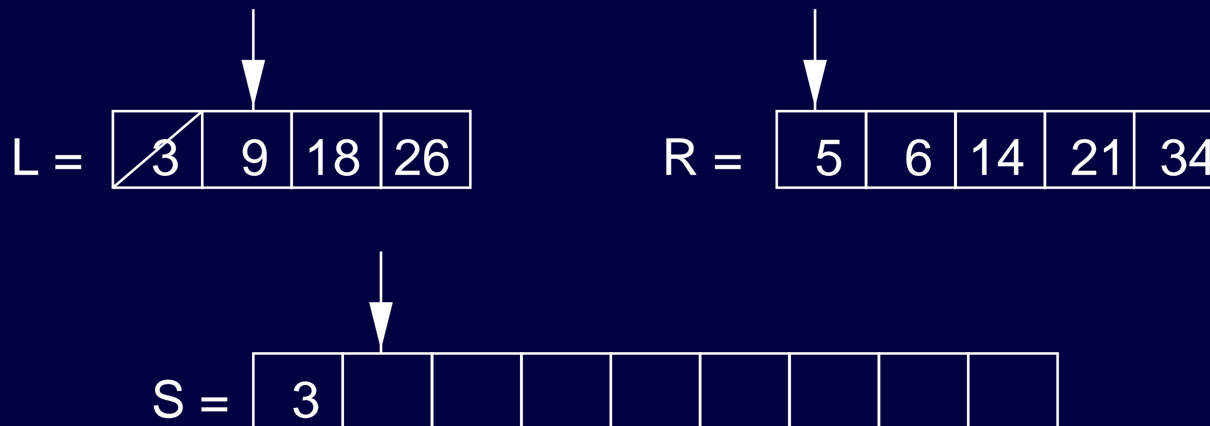
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

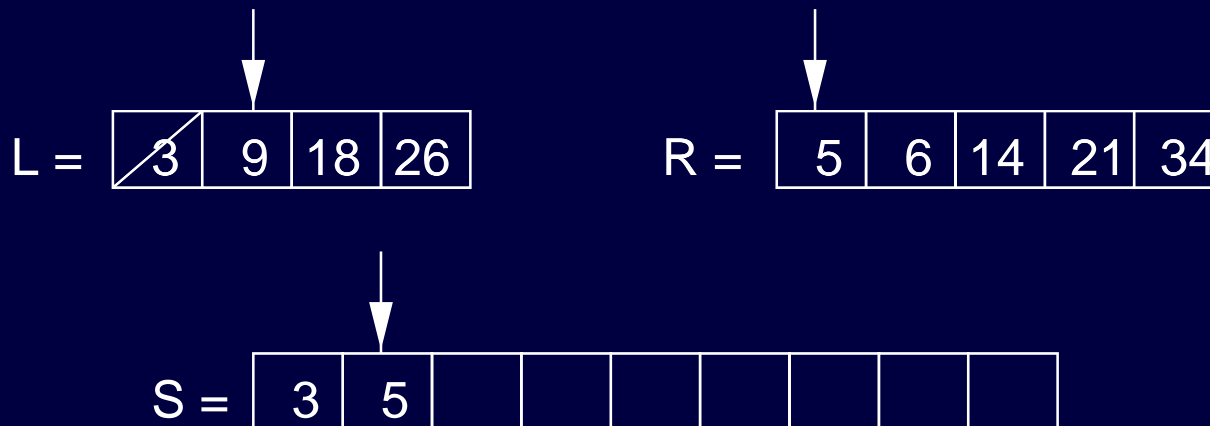
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

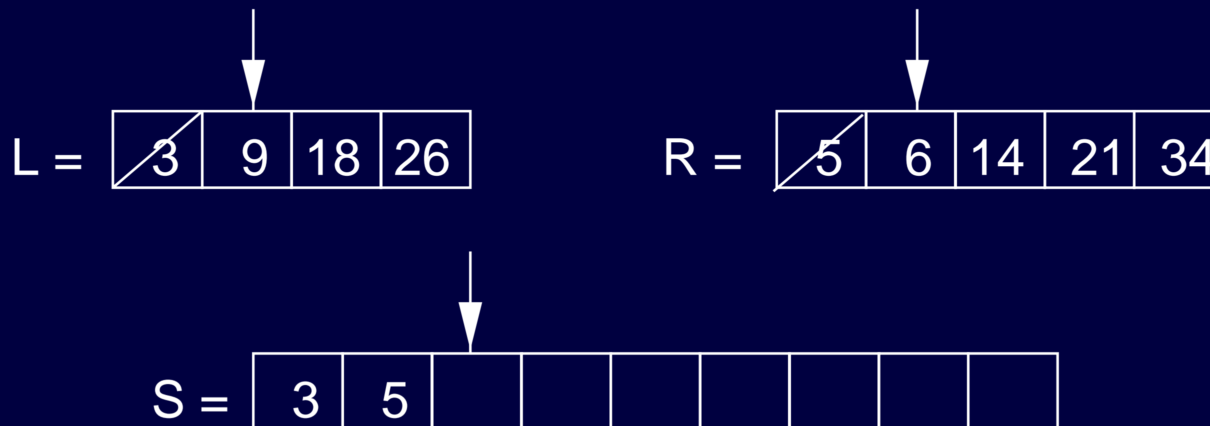
Assume two sorted sequences **`L`** and **`R`**

To merge values from **`L`** and **`R`**, into a new sorted sequence **`S`**:

Repeatedly

1. Compare the leading values of **`L`** and **`R`**
2. Write the smallest one to **`S`**
3. (If one sequence is empty, copy remaining elements of other into **`S`**)

Result: **`S`** contains all elements in **`L`** and **`R`** arranged in sorted order



The Merge Step in MergeSort

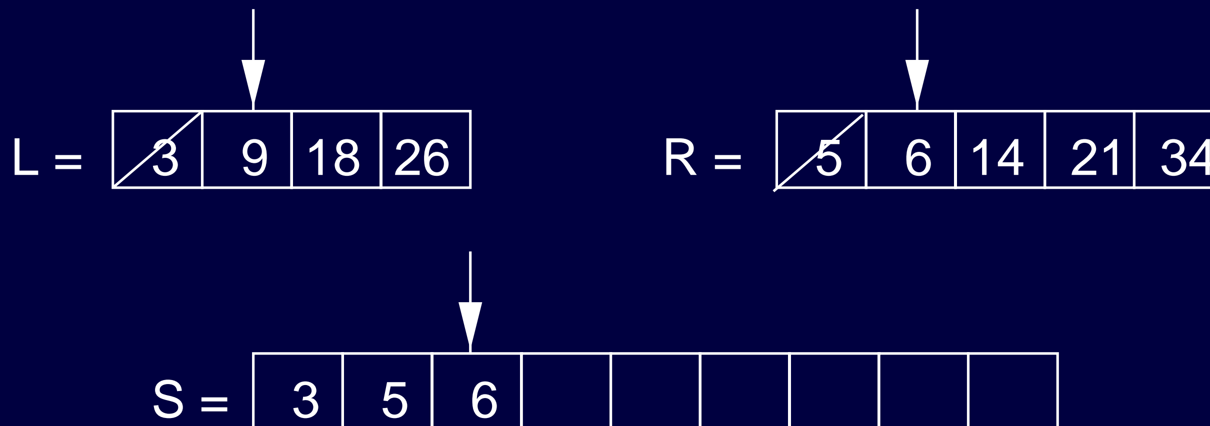
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

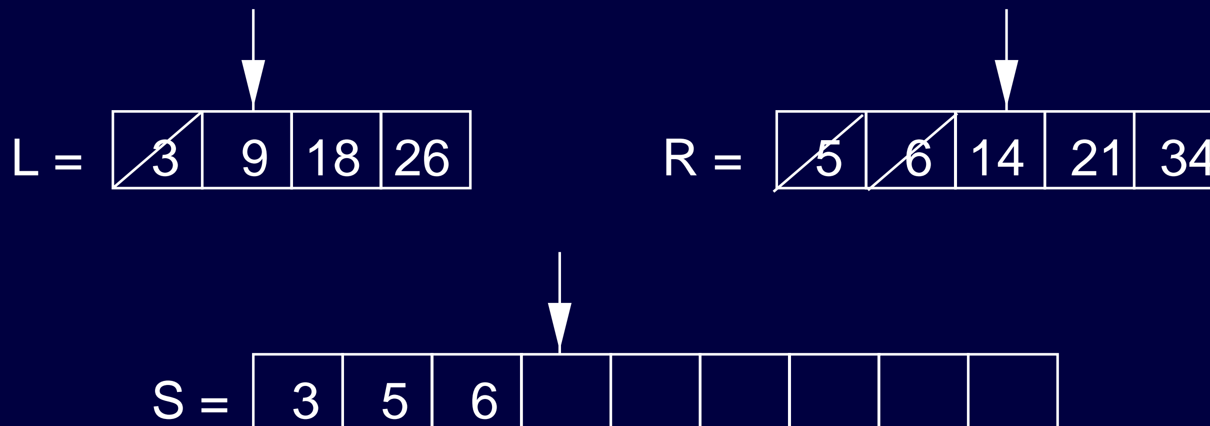
Assume two sorted sequences **`L`** and **`R`**

To merge values from **`L`** and **`R`**, into a new sorted sequence **`S`**:

Repeatedly

1. Compare the leading values of **`L`** and **`R`**
2. Write the smallest one to **`S`**
3. (If one sequence is empty, copy remaining elements of other into **`S`**)

Result: **`S`** contains all elements in **`L`** and **`R`** arranged in sorted order



The Merge Step in MergeSort

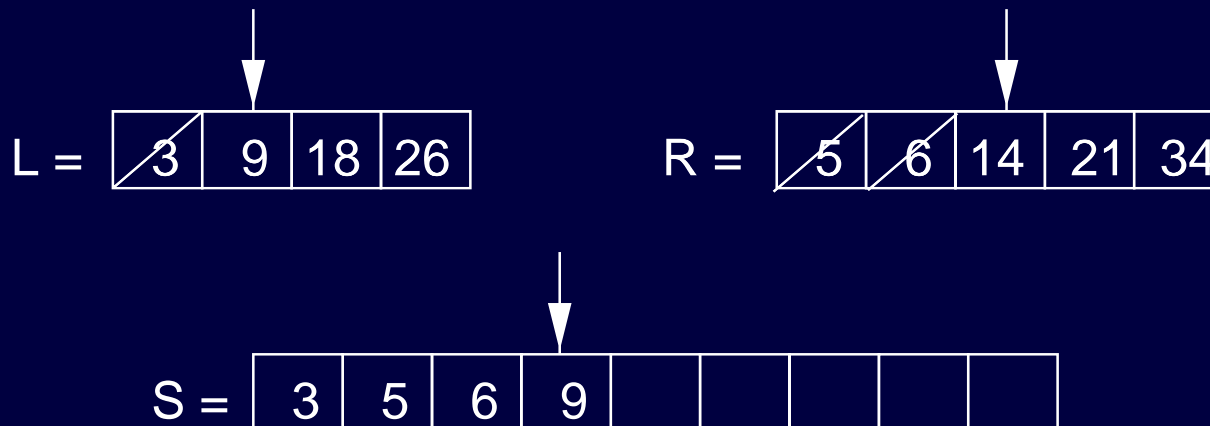
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

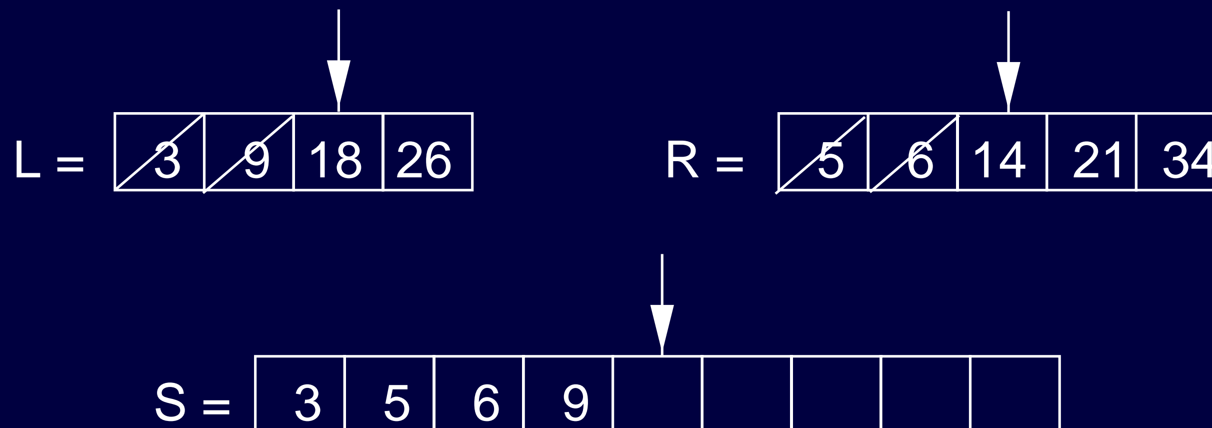
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

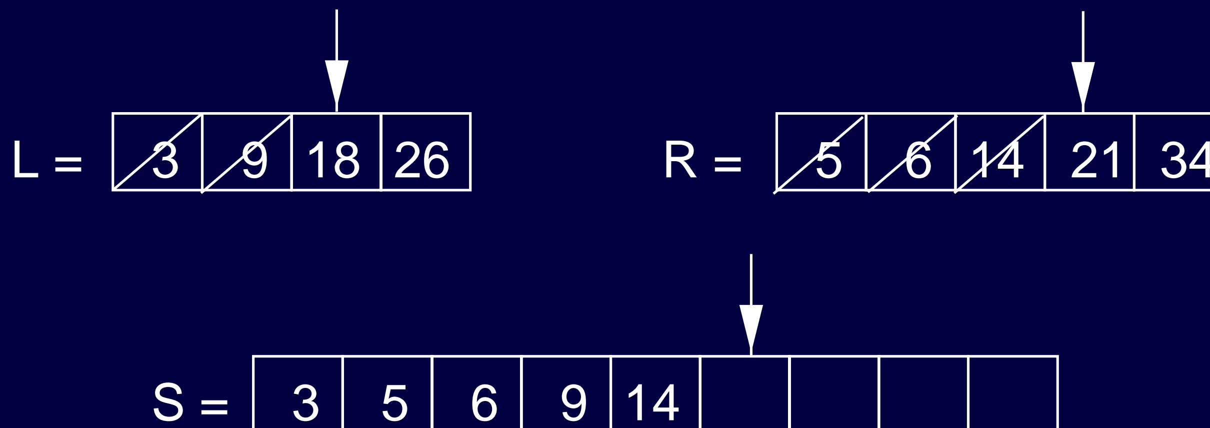
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

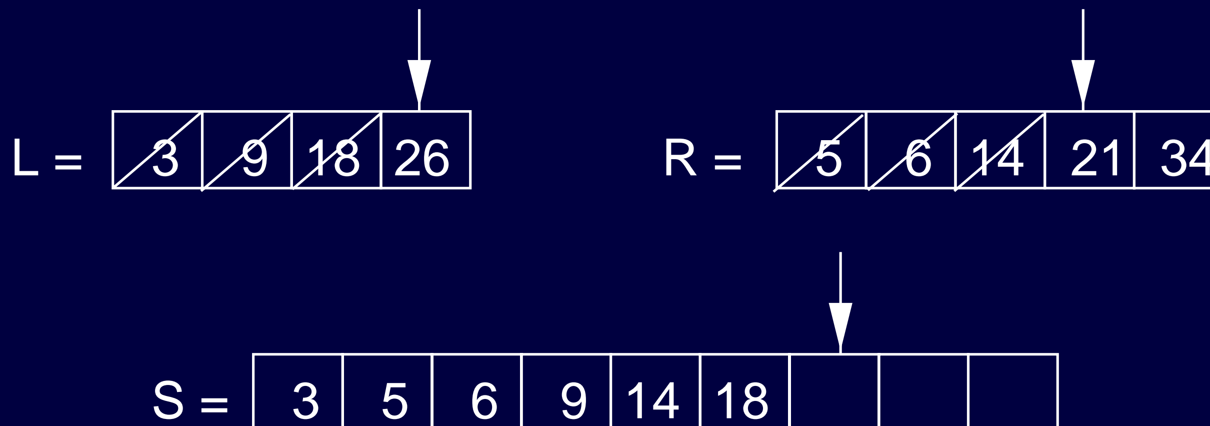
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

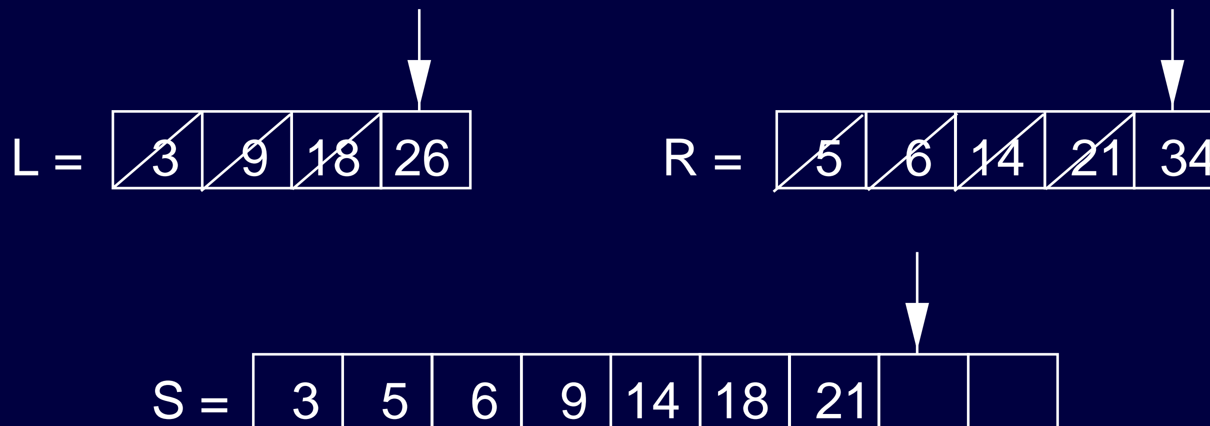
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

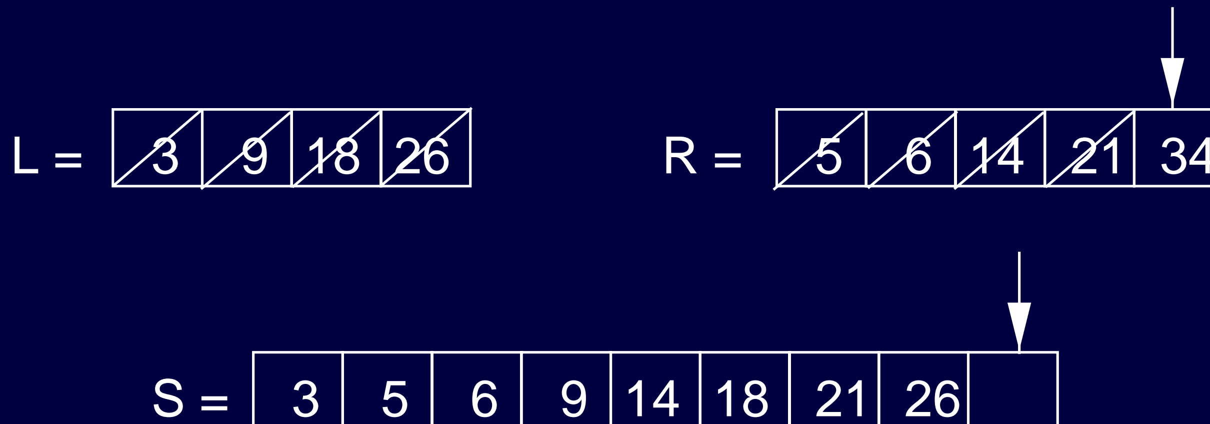
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



The Merge Step in MergeSort

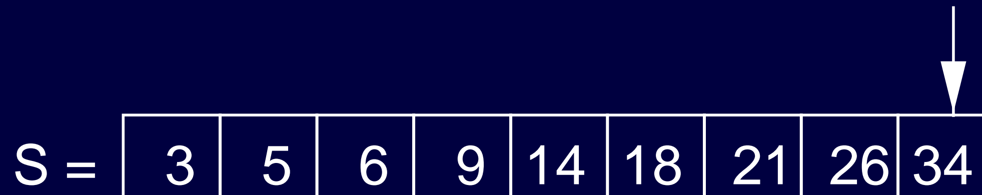
Assume two sorted sequences `L` and `R`

To merge values from `L` and `R`, into a new sorted sequence `S`:

Repeatedly

1. Compare the leading values of `L` and `R`
2. Write the smallest one to `S`
3. (If one sequence is empty, copy remaining elements of other into `S`)

Result: `S` contains all elements in `L` and `R` arranged in sorted order



MergeSort: In Action

MergeSort **A** = 18 26 3 9 34 5 21 14 6

- Divide **A**: **L** = 18 26 3 9 and **R** = 34 5 21 14 6
- MergeSort **L**:
  - Divide **L**: **LL** = 18 26 and **LR** = 3 9
  - MergeSort **LL**:
    - Divide **LL**: **LLL** = 18 and **LLR** = 26
    - MergeSort **LLL**: ⇐ is already sorted
    - MergeSort **LLR**: ⇐ is already sorted
    - Merge 18 (sorted **LLL**) and 26 (sorted **LLR**): Result: 18 26
  - MergeSort **LR**: ... Result: 3 9
  - Merge 18 26 (sorted **LL**) and 3 9 (sorted **LR**): Result: 3 9 18 26
- MergeSort **R**: ... Result: 5 6 14 21 34

Merge 3 9 18 26 (sorted **L**) and 5 6 14 21 34 (sorted **R**):

Result: 3 5 6 9 14 18 21 26 34

MergeSort: Summary

Strategy:

- Divide an array `A` with `n` elements into two half-arrays `L[0:middle]` and `R[middle+1:n-1]`
- Combine sorted versions of `L` and `R` by merging them into a single sorted result

Requires an additional array of equal size to the one being sorted.

- If original array just about fits into memory, MergeSort won't work!
- If you have enough space, MergeSort is a good choice.

Complexity class:

- Average case: $O(\mathbf{n} \log \mathbf{n})$
- Worst case: $O(\mathbf{n} \log \mathbf{n})$

Other sorting algorithms

- Quicksort: Merge-sort, but partitions the array cleverly.
- Insertion sort: Bubble sort but add one value at a time.
- ...
- There is another lecture worth
  - Not taught, you can read it for interest.

Where to get more information?

Thomas A. Standish
"Data Structures, Algorithms & Software Principles in C"
Addison-Wesley, 1995
Chapter 13: Sorting (p. 524 ff)

A. D. Dewdney **"The New Turing Omnibus"**
Computer Science Press, 1997
Chapters 35, 40 on Sorting Problems and Algorithms

Next: QuickSort, InsertionSort

Sorting Algorithms II

QuickSort

QuickSort was discovered in 1962.

Refinement of general *divide and conquer* strategy

To sort **n** values, held in array **A[0:n-1]**, in ascending order:

1. Partition sequence **A** into left **L** and right **R** partion using one of the values from **A** as a pivot value.
2. Sort the left partition **L**.
3. Sort the right partition **R**.

- Use QuickSort (recursively) to sort both partitions.
- One element sequences are already sorted.
- The heart of QuickSort is the **Partition** step.

QuickSort

QuickSort was discovered in 1962.

Refinement of general *divide and conquer* strategy

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

1. Partition sequence `A` into left `L` and right `R` partion using one of the values from `A` as a pivot value.
2. QuickSort the left partition `L`.
3. Sort the right partition `R`.

- Use QuickSort (recursively) to sort both partitions.
- One element sequences are already sorted.
- The heart of QuickSort is the **Partition** step.

QuickSort

QuickSort was discovered in 1962.

Refinement of general *divide and conquer* strategy

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

1. Partition sequence `A` into left `L` and right `R` partion using one of the values from `A` as a pivot value.
2. QuickSort the left partition `L`.
3. QuickSort the right partition `R`.

- Use QuickSort (recursively) to sort both partitions.
- One element sequences are already sorted.
- The heart of QuickSort is the **Partition** step.

QuickSort

QuickSort was discovered in 1962.

Refinement of general *divide and conquer* strategy

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

1. **Partition** sequence `A` into left `L` and right `R` partion using one of the values from `A` as a pivot value.
2. QuickSort the left partition `L`.
3. QuickSort the right partition `R`.

- Use QuickSort (recursively) to sort both partitions.
- One element sequences are already sorted.
- The heart of QuickSort is the **Partition** step.

Data Partition

To *partition* data is to divide it into two groups:

- select discrimination criteria (pivot value)
- put all items with value higher than pivot in one group
- put all items with value lower than pivot in the other group

Choose any value for the pivot value (depending on purpose of partition)

After partitioning:

- data has simply been divided into two groups
- data is not yet sorted, but more sorted than before
- From here, it is not difficult to completely sort the data.

The Partition Step in QuickSort

Assume a sequence `A`

To partition `A[m:n]` into a left partition `L` and a right partition `R`

Choose the middle value as the pivot `P`

Repeat

1. From left find first `i` s.t. `A[i]` $\geq$ `P` (or `A[i]` $\not<$ `P`)
2. From right find first `j` s.t. `A[j]` $\leq$ `P` (or `A[i]` $\not>$ `P`)
3. If `i` and `j` did not cross over, swap `A[i]` and `A[j]`, afterwards increment `i` and decrement `j`

Until `i` and `j` cross over

- Values left (right) of cross-over point are less (greater) than `P`

Result: left partition `L` = `A[m:j]` and right partition `R` = `A[i:n]`

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

A =	18	26	3	9	6	5	21	14

Choose pivot value `P`

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

`i`=left end of `A`

`j`=right end of `A`

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

From left: `18` $\not<$ `6`, from right: `14` $>$ `6`

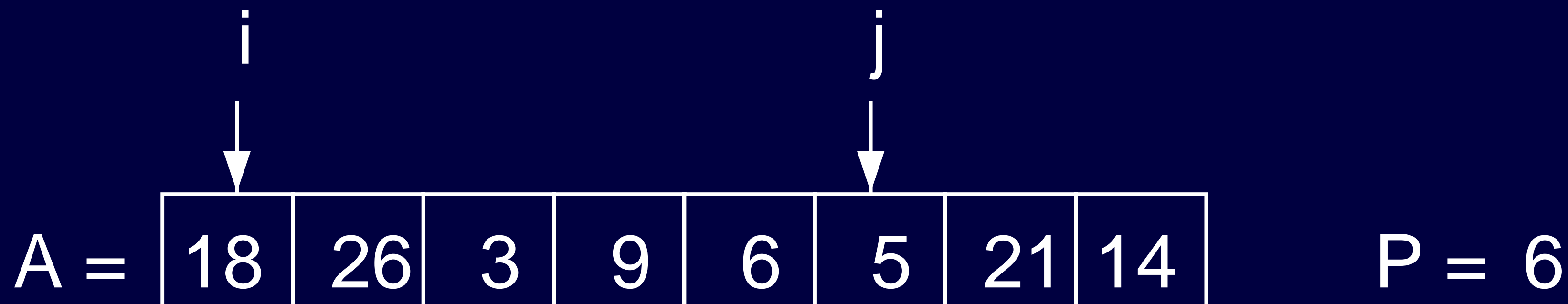
Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

From left: `18` $\not<$ `6`, from right: `21` $>$ `6`

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

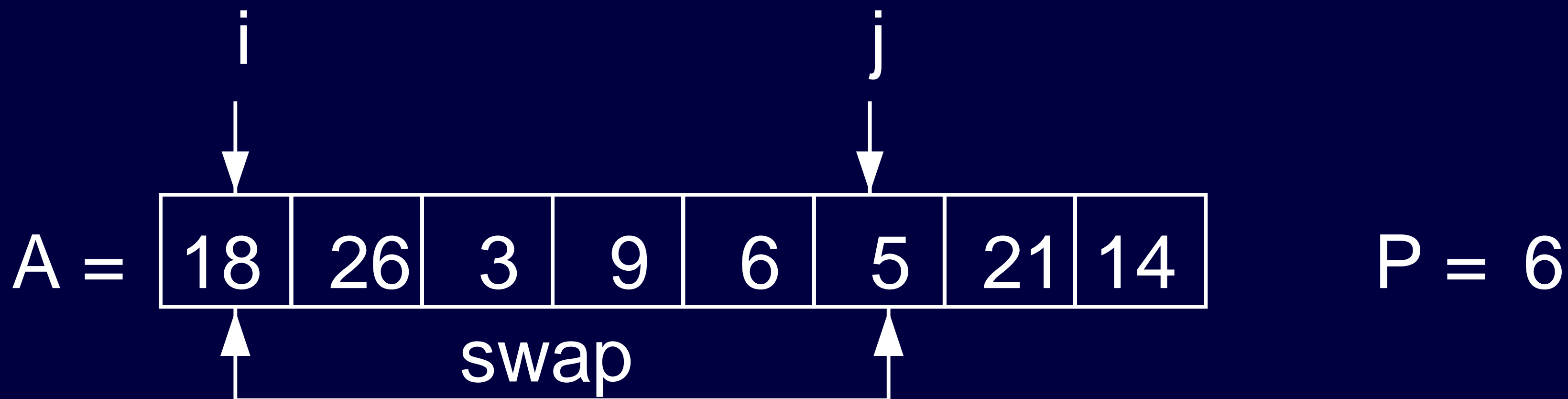


From left: `18` $\not<$ `6`, from right: `5` $\not>$ `6`

Swap `A[i]` and `A[j]`

Partitioning: In Action

Aim: Divide **`A = 18 26 3 9 6 5 21 14`** into a left partition **`L`** and a right partition **`R`**

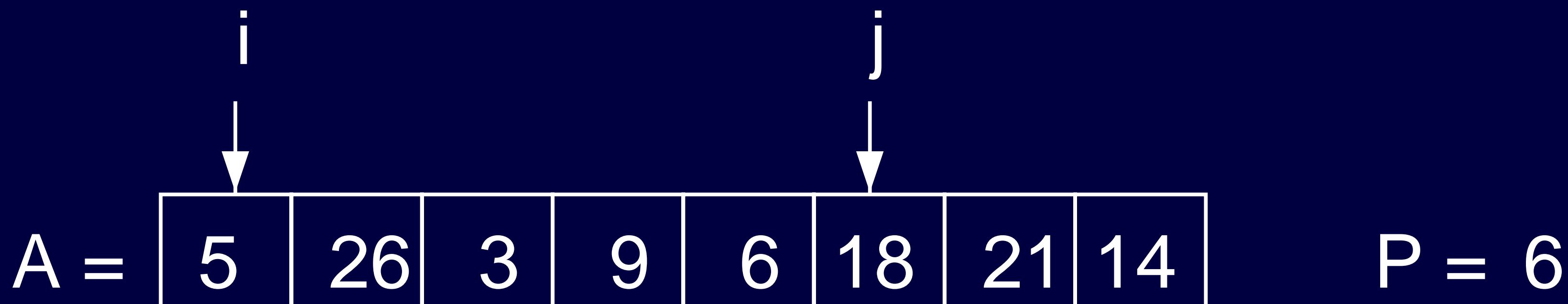


From left: **`18`** $\not<$ **`6`**, from right: **`5`** $\not>$ **`6`**

Swap **`A[i]`** and **`A[j]`**

Partitioning: In Action

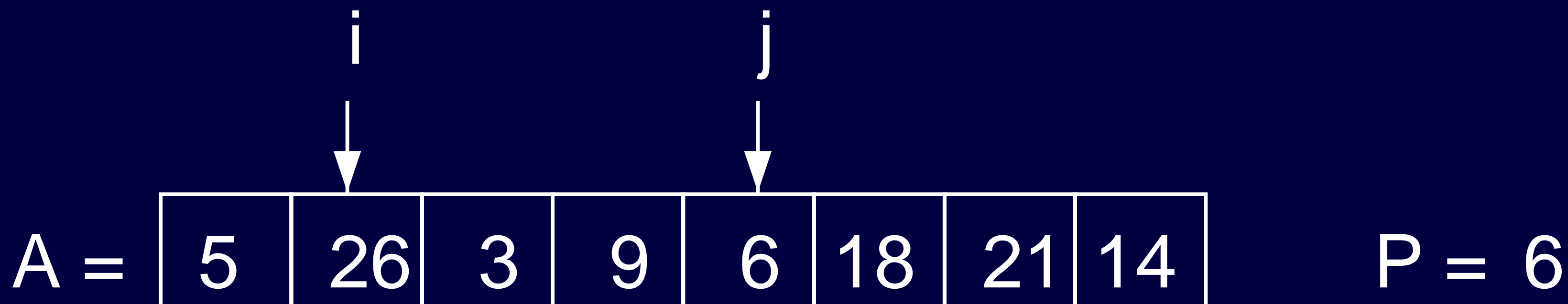
Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`



Increment `i`, decrement `j`

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`



From left: `26` $\not<$ `6`, from right: `6` $\not>$ `6`

Swap `A[i]` and `A[j]`

Partitioning: In Action

Aim: Divide **`A = 18 26 3 9 6 5 21 14`** into a left partition **`L`** and a right partition **`R`**

A =	5	26	3	9	6	18	21	14

i points to 26, j points to 6; swap

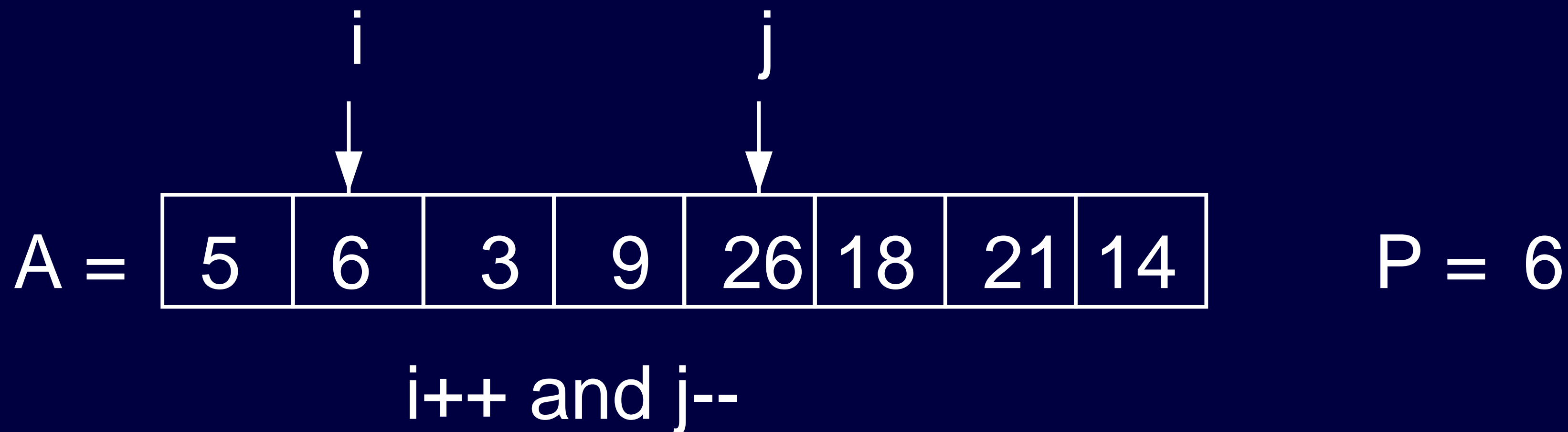
P = 6

From left: **`26`** $\not<$ **`6`**, from right: **`6`** $\not>$ **`6`**

Swap **`A[i]`** and **`A[j]`**

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`



Increment `i`, decrement `j`

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

From left: `3` $<$ `6`

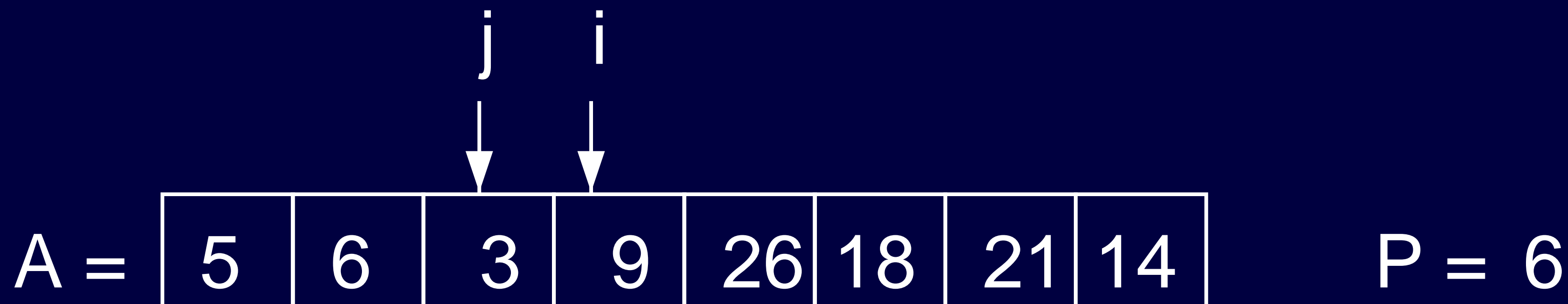
Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

From left: **9** $\not<$ **6**, from right: **9** $>$ **6**

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`



From left: **9** $\not<$ **6**, from right: **3** $\not>$ **6**

Partitioning: In Action

Aim: Divide `A = 18 26 3 9 6 5 21 14` into a left partition `L` and a right partition `R`

Index cross-over!

Cross-over point is border of partition.

Partition Algorithm (C code)

```
void Partition(int A[], int *i, int *j) {
  int temp, pivot;

  pivot = A[(*i + *j)/2];            /* take middle as pivot */

  do {
    while (A[*i] < pivot) (*i)++;  /* find leftmost >= pivot */
    while (A[*j] > pivot) (*j)--; /* find rightmost <= pivot */

    if (*i <= *j) {              /* i and j did not cross over */
      if (*i < *j) {                 /* i and j are different */
        temp = A[*i]; A[*i] = A[*j]; A[*j] = temp;   /* swap */
      }
      (*i)++;                         /* move i one space right */
      (*j)--;                          /* move j one space left */
    }

  } while (*i <= *j); /* while i and j have not crossed over */
}
```

QuickSort Algorithm (C Code)

To sort subarray **A[m:n]** of array **A** of integers into ascending order,

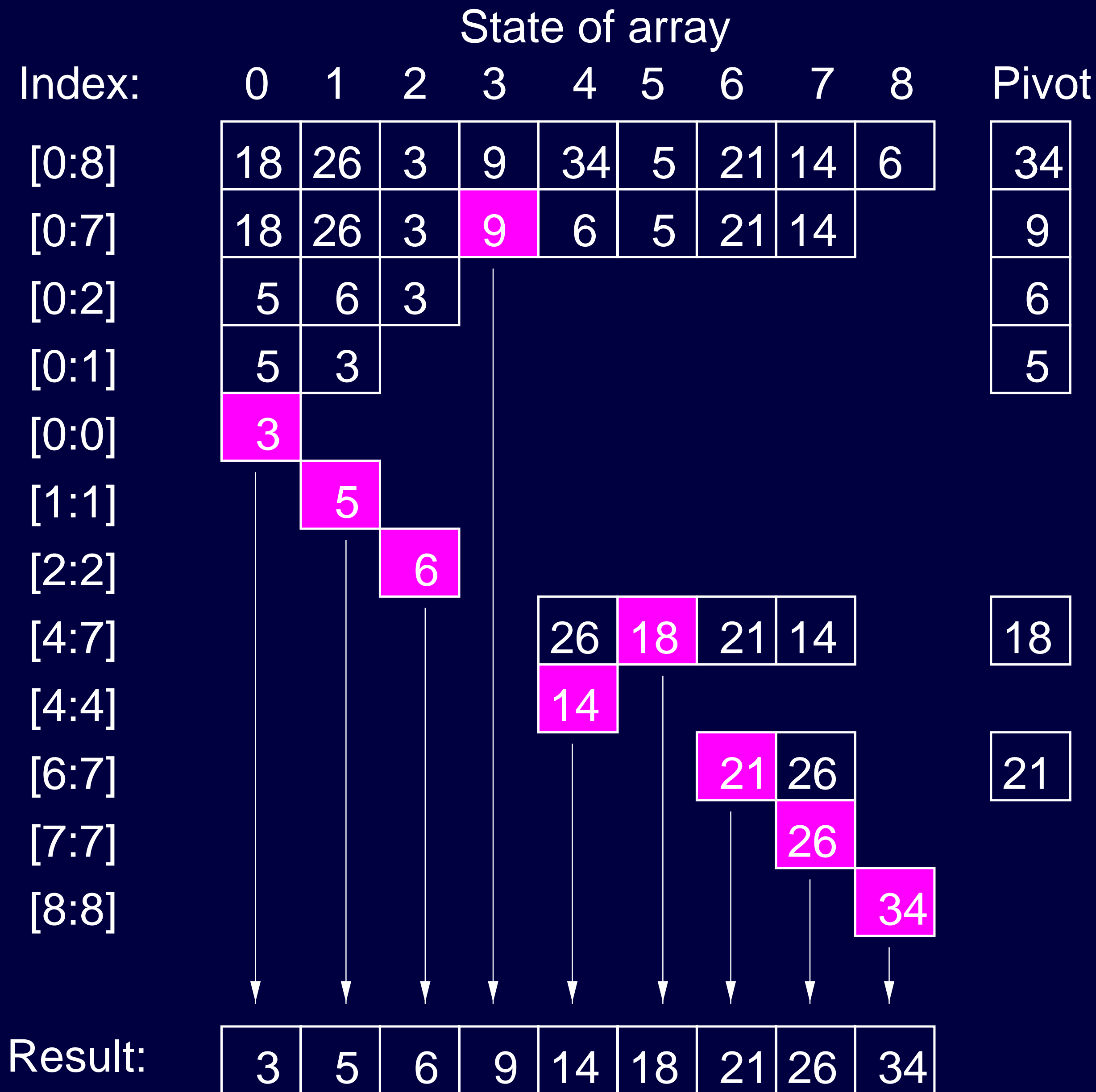
```
void QuickSort(int A[], int m, int n) {
  int i, j;

  if (m<n) {
    i = m;                        /* i is index of first item */
    j = n;                         /* j is index of last item */
    Partition(A, &i, &j);     /* partition A[m:n] into */
    QuickSort(A,m,j);                          /* A[m:j] and */
    QuickSort(A,i,n);                              /* A[i:n] */
  }
}
```

recursively call the partition algorithm.

- No extra memory necessary (as is in MergeSort), result is in **A**.
- Early passes put values close to final position, later passes make minor changes.

QuickSort: In Action



QuickSort: Summary

Strategy:

- Choose a value from array `A` with $\mathbf{n}$ elements as pivot
- Divide `A` into two partitions `L` and `R` s.t.

  `L` contains values $\leq$ pivot, `R` contains values $\geq$ pivot
- Sort `L` and `R` by recursively applying QuickSort
- Once `L` and `R` are sorted, no further action is necessary.

Complexity class:

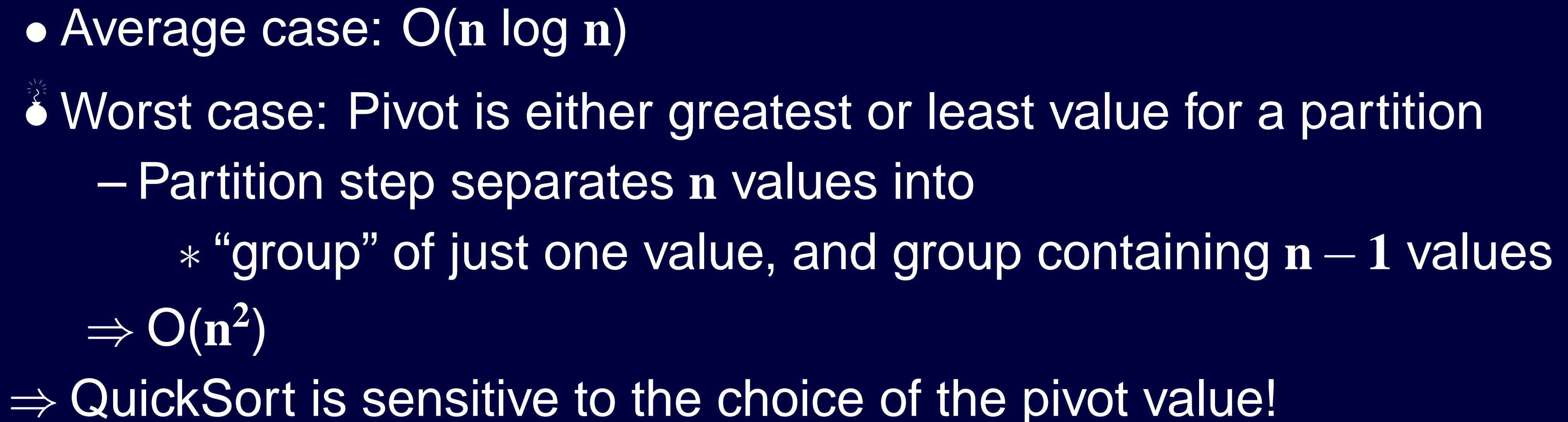
- Average case: O($\mathbf{n}$ log $\mathbf{n}$)
- Worst case:

QuickSort: Summary

Strategy:

- Choose a value from array $\mathtt{A}$ with $\mathbf{n}$ elements as pivot
- Divide $\mathtt{A}$ into two partitions $\mathtt{L}$ and $\mathtt{R}$ s.t.

  $\mathtt{L}$ contains values $\leq$ pivot, $\mathtt{R}$ contains values $\geq$ pivot
- Sort $\mathtt{L}$ and $\mathtt{R}$ by recursively applying QuickSort
- Once $\mathtt{L}$ and $\mathtt{R}$ are sorted, no further action is necessary.



Complexity class:

- Average case: O($\mathbf{n}$ log $\mathbf{n}$)
- Worst case: Pivot is either greatest or least value for a partition
  - Partition step separates $\mathbf{n}$ values into
    - “group” of just one value, and group containing $\mathbf{n-1}$ values

  $\Rightarrow$ O($\mathbf{n}^2$)

$\Rightarrow$ QuickSort is sensitive to the choice of the pivot value!

Insert and Keep Sorted Method(s)

- **InsertionSort** and TreeSort are “insert and keep sorted” methods

Strategy:

- Divide values to be sorted into
  - collection `U` of unsorted values (to be sorted)
  - collection `S` of sorted values (to be kept in sorted order)
- Remove values (one-at-a-time) from `U` and insert into `S`
  - Maintain `S` in sorted order.

Initialization:

- `U` = ... , and `S` = ...

Termination:

- `U` = ... , and `S` = ...

Insert and Keep Sorted Method(s)

- **InsertionSort** and TreeSort are “insert and keep sorted” methods

Strategy:

- Divide values to be sorted into
  - collection `U` of unsorted values (to be sorted)
  - collection `S` of sorted values (to be kept in sorted order)
- Remove values (one-at-a-time) from `U` and insert into `S`
  - Maintain `S` in sorted order.

Initialization:

- `U` = entire set of unsorted values, and `S` = ...

Termination:

- `U` = ... , and `S` = ...

Insert and Keep Sorted Method(s)

- **InsertionSort** and TreeSort are “insert and keep sorted” methods

Strategy:

- Divide values to be sorted into
  - collection `U` of unsorted values (to be sorted)
  - collection `S` of sorted values (to be kept in sorted order)
- Remove values (one-at-a-time) from `U` and insert into `S`
  - Maintain `S` in sorted order.

Initialization:

- `U` = entire set of unsorted values, and `S` = empty

Termination:

- `U` = ... , and `S` = ...

Insert and Keep Sorted Method(s)

- **InsertionSort** and TreeSort are “insert and keep sorted” methods

Strategy:

- Divide values to be sorted into
  - collection `U` of unsorted values (to be sorted)
  - collection `S` of sorted values (to be kept in sorted order)
- Remove values (one-at-a-time) from `U` and insert into `S`
  - Maintain `S` in sorted order.

Initialization:

- `U` = entire set of unsorted values, and `S` = empty

Termination:

- `U` = empty, and `S` = ...

Insert and Keep Sorted Method(s)

- **InsertionSort** and TreeSort are “insert and keep sorted” methods

Strategy:

- Divide values to be sorted into
  - collection `U` of unsorted values (to be sorted)
  - collection `S` of sorted values (to be kept in sorted order)
- Remove values (one-at-a-time) from `U` and insert into `S`
  - Maintain `S` in sorted order.

Initialization:

- `U` = entire set of unsorted values, and `S` = empty

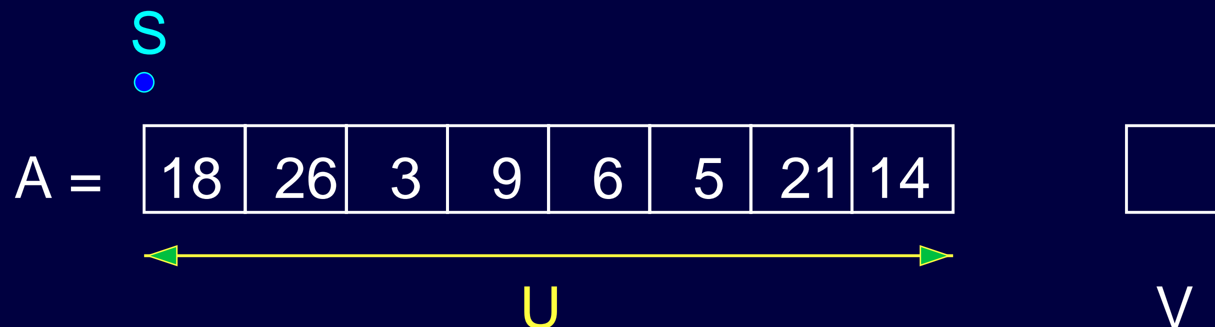
Termination:

- `U` = empty, and `S` = entire set of sorted values

InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

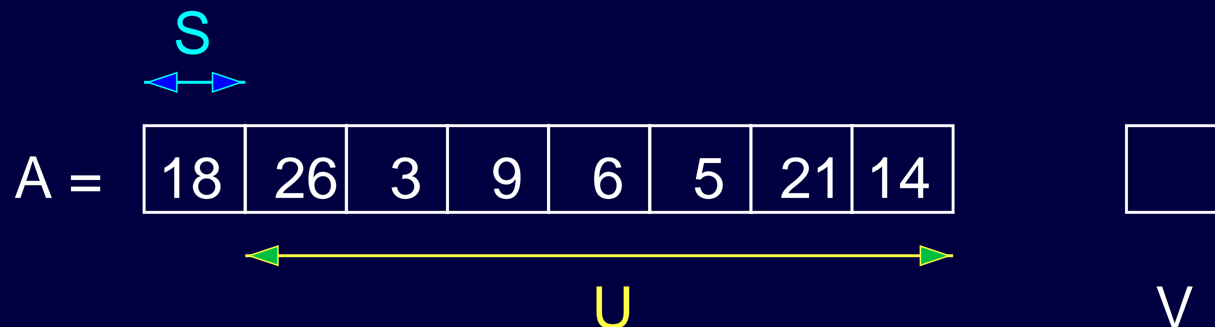
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - ∗ This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

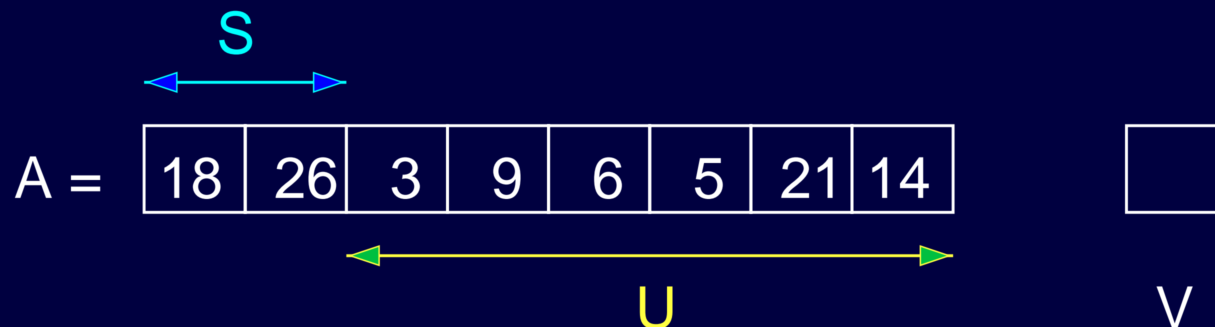
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

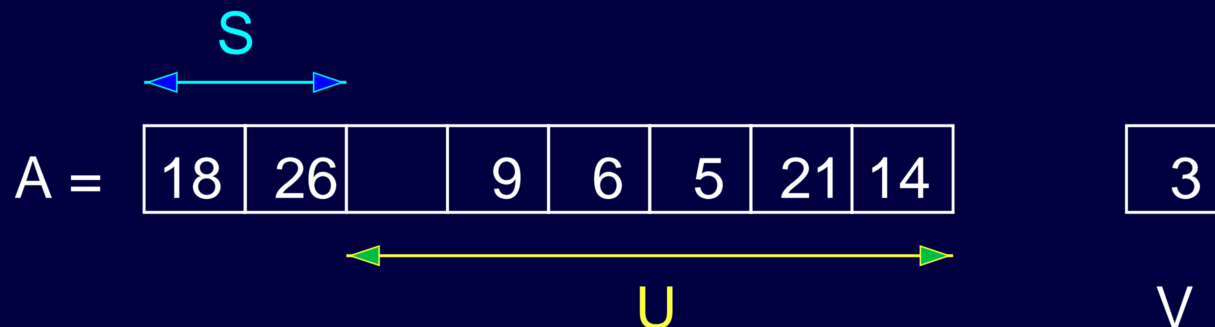
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

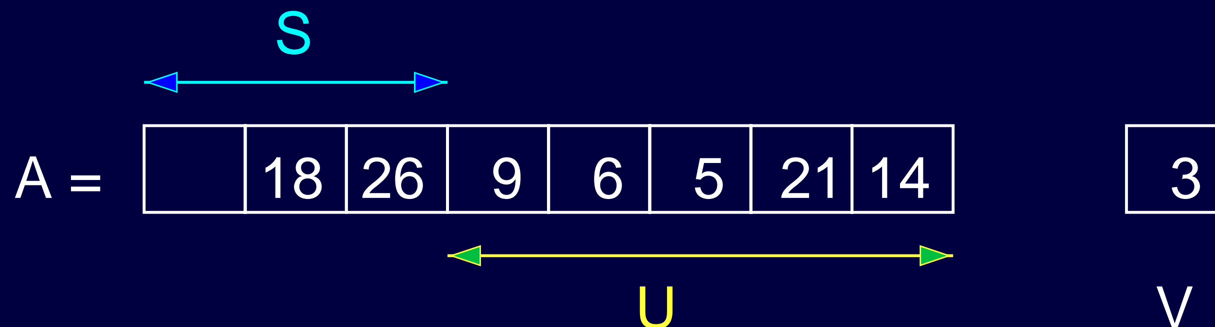
- Use `A` to contain subarrays `S` and `U` “side-by-side”
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

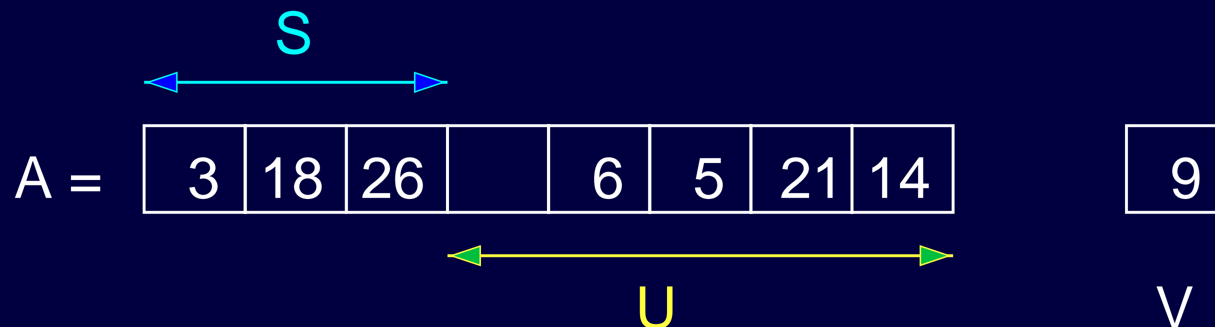
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

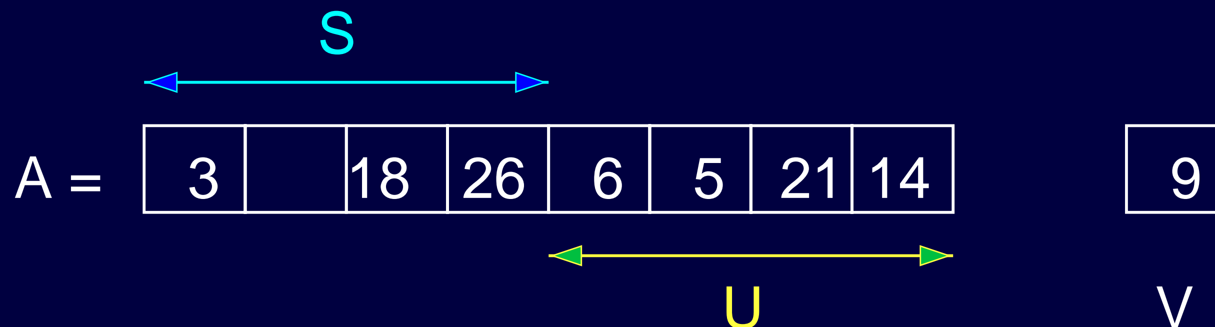
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - ∗ This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

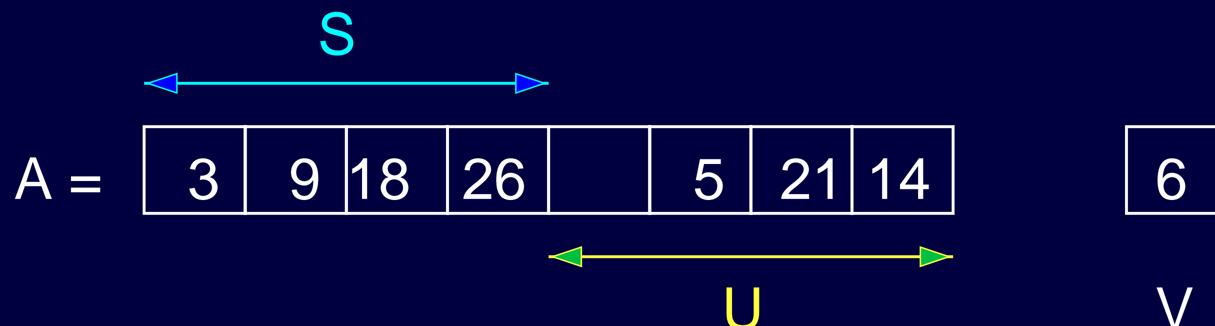
- Use `A` to contain subarrays `S` and `U` “side-by-side”
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - ∗ This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

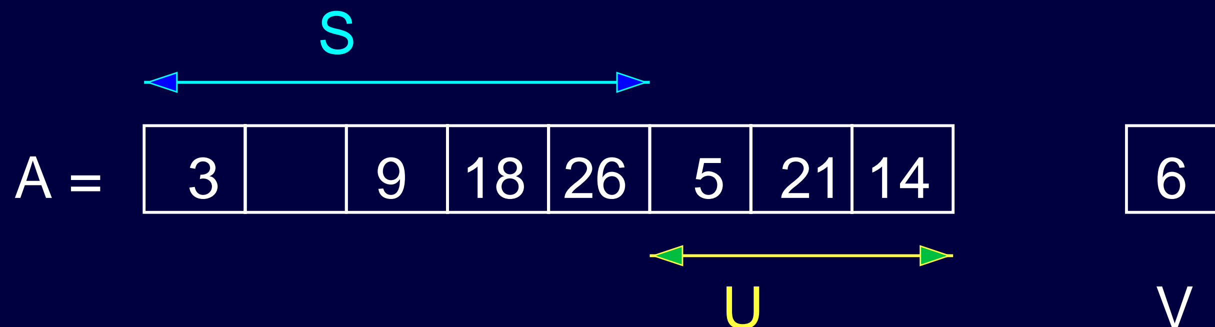
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

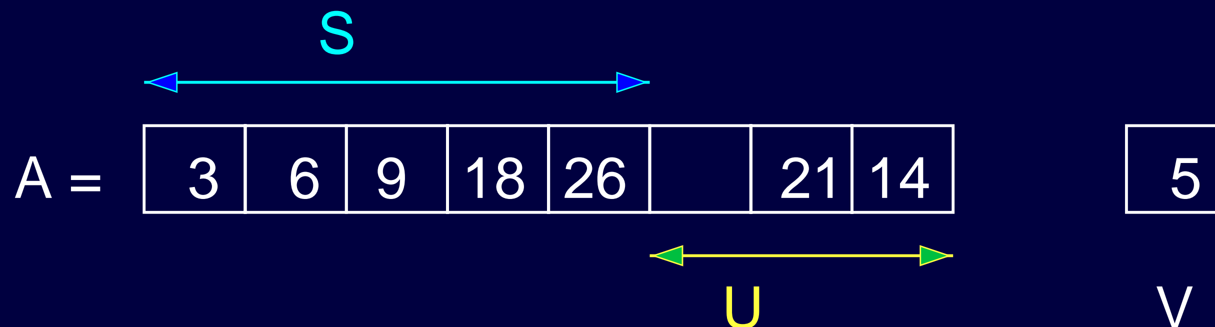
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - ∗ This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

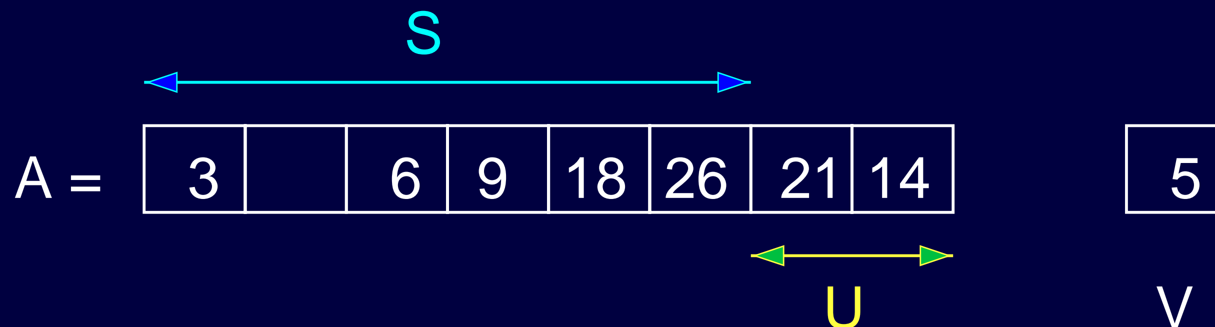
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - ∗ This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

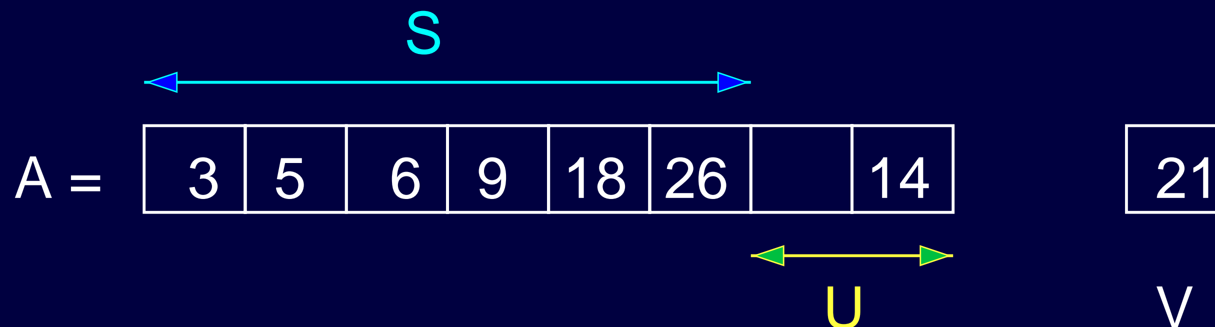
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

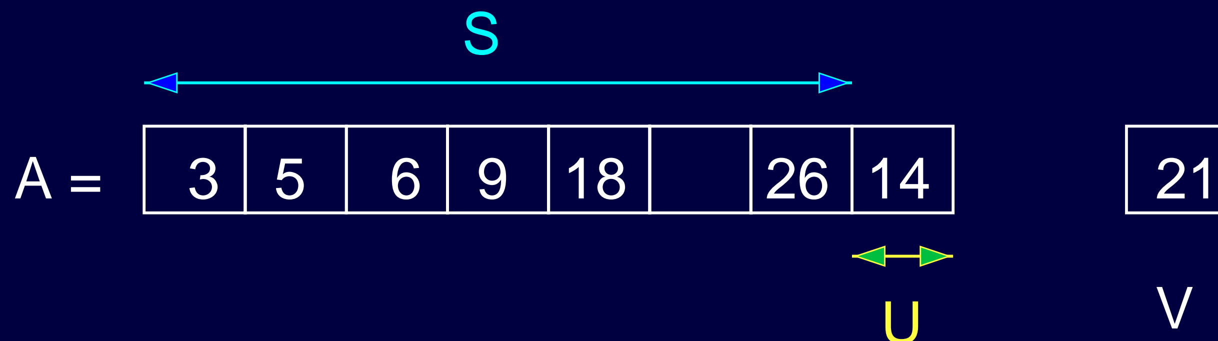
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

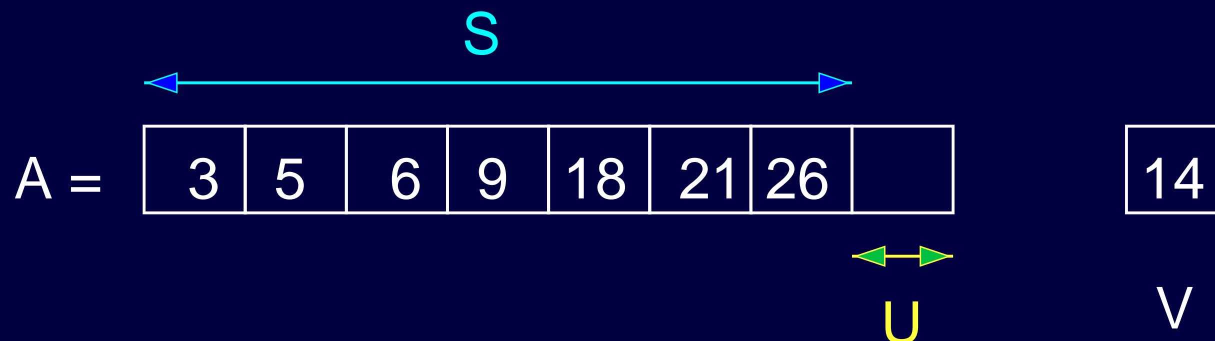
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort **n** values, held in array **A[0:n-1]**, in ascending order:

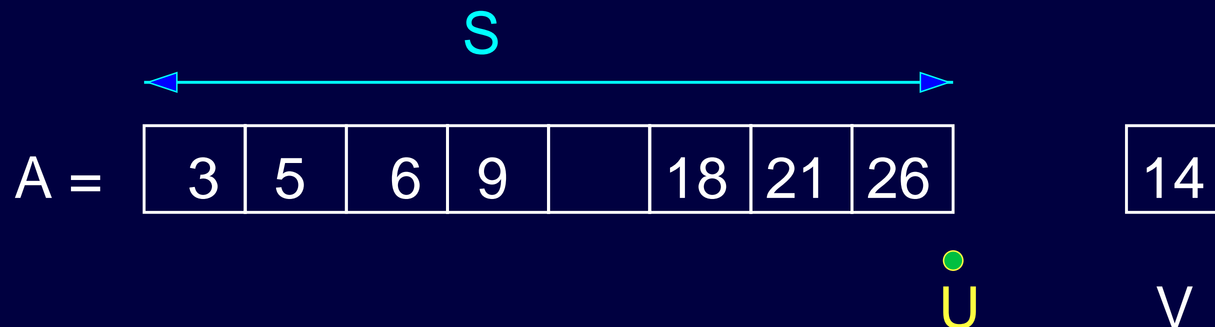
- Use **A** to contain subarrays **S** and **U** "side-by-side"
  - **S** initially at left end of **A**, and **U** initially at right end of **A**
  - Detach a value **V** at left end of **U**
    - This creates a hole at the border between **S** and **U**
  - Move all values in **S** that are bigger than **V** one space to right
  - Insert **V** into the remaining hole in **S**.
- Repeat until all values in **U** are inserted into **S**
- At the end **S** occupies same space as **A**



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

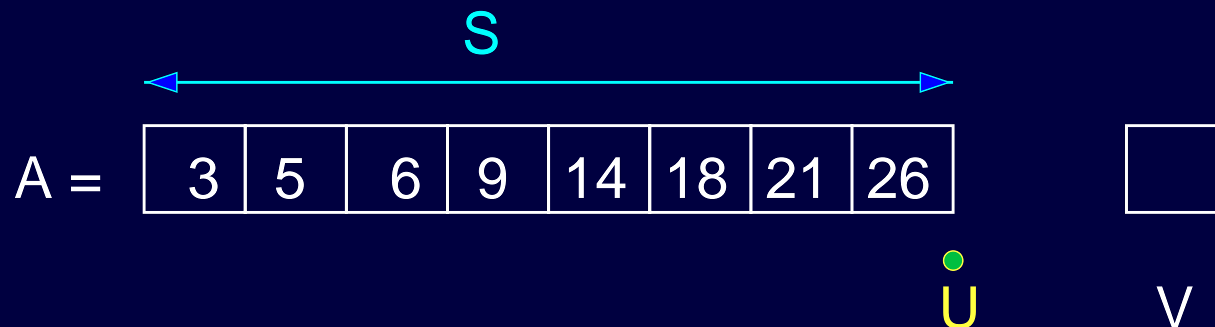
- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - ∗ This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: In Action

To sort `n` values, held in array `A[0:n-1]`, in ascending order:

- Use `A` to contain subarrays `S` and `U` "side-by-side"
  - `S` initially at left end of `A`, and `U` initially at right end of `A`
  - Detach a value `V` at left end of `U`
    - This creates a hole at the border between `S` and `U`
  - Move all values in `S` that are bigger than `V` one space to right
  - Insert `V` into the remaining hole in `S`.
- Repeat until all values in `U` are inserted into `S`
- At the end `S` occupies same space as `A`



InsertionSort: Summary

Strategy:

- Start with empty “sorted” storage container `s`
- Successively insert values into `s` from unsorted container `u` until `u` is empty

Complexity class:

- On average: $O(\mathbf{n}^2)$
- Best case:
- Worst case:

InsertionSort: Summary

Strategy:

- Start with empty “sorted” storage container `s`
- Successively insert values into `s` from unsorted container `u` until `u` is empty

Complexity class:

- On average: $O(\mathbf{n}^2)$
- Best case: input already sorted $\Rightarrow$ $O(\mathbf{n})$
- Worst case:

InsertionSort: Summary

Strategy:

- Start with empty “sorted” storage container `s`
- Successively insert values into `s` from unsorted container `u` until `u` is empty

Complexity class:

- On average: $O(\mathbf{n}^2)$
- Best case: input already sorted $\Rightarrow$ $O(\mathbf{n})$
- Worst case: input in reverse order $\Rightarrow$ $O(\mathbf{n}^2)$

InsertionSort: Summary

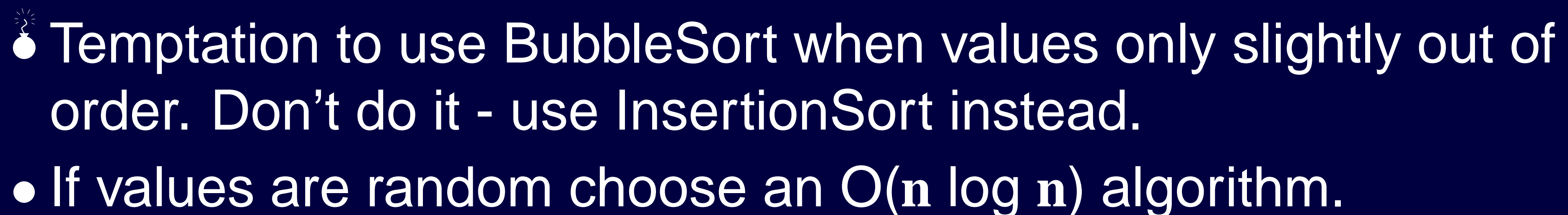
Strategy:

- Start with empty “sorted” storage container `S`
- Successively insert values into `S` from unsorted container `U` until `U` is empty

Complexity class:

- On average: $O(n^2)$
- Best case: input already sorted $\Rightarrow$ $O(n)$
- Worst case: input in reverse order $\Rightarrow$ $O(n^2)$

Pitfalls:



- Temptation to use BubbleSort when values only slightly out of order. Don’t do it - use InsertionSort instead.
- If values are random choose an $O(n \log n)$ algorithm.

Sorting Algorithms: A Summary

- Examine sorting problem carefully, look for a **simpler** solution.

- Avoid the pitifully inefficient methods.
- Know the properties of the most common sorting algorithms.
  - You now know: BubbleSort, MergeSort, QuickSort, InsertionSort

- **If you absolutely need the most efficient method:**
  - Test various methods, consider their worst case behaviour,
  - measure and tune, and
  - select the “best” for the problem.

Conclusion: There is no single best sorting algorithm that beats all the others all the time.

Sorting Algorithms: A Summary

- Examine sorting problem carefully, look for a **simpler** solution.
  - **Clever** “index mapping” might be good enough!
- Avoid the pitifully inefficient methods.
- Know the properties of the most common sorting algorithms.
  - You now know: BubbleSort, MergeSort, QuickSort, InsertionSort

- **If you absolutely need the most efficient method:**
  - Test various methods, consider their worst case behaviour,
  - measure and tune, and
  - select the “best” for the problem.

Conclusion: There is no single best sorting algorithm that beats all the others all the time.

Sorting Algorithms and their Complexity

Average complexity	Sort
$O(\mathbf{n}^2)$	Selection, **Insertion**, **Bubble**
$O(\mathbf{n} \log \mathbf{n})$	**Quick**, **Merge**, Heap
$O(\mathbf{n})$	Address calculation (not in this course)

Choice of “right” algorithm can make a big difference for large problems.

One can prove:

Comparison-based sorting algorithms (1st two lines in table above) can’t sort any faster than $O(\mathbf{n} \log \mathbf{n})$ in the average case!

A Lower Bound on Speed

Given a list of $\mathbf{n}$ elements, what is the minimum amount of time a computer may take to sort it?

MergeSort average complexity is $O(\mathbf{n} \log \mathbf{n})$.

A Lower Bound on Speed

Given a list of $\mathbf{n}$ elements, what is the minimum amount of time a computer may take to sort it?

MergeSort average complexity is O($\mathbf{n}$ log $\mathbf{n}$).

Means, that for large enough $\mathbf{n}$ the number of steps this algorithm takes to sort $\mathbf{n}$ elements is never greater than c * $\mathbf{n}$ log $\mathbf{n}$, for some constant c.

$\Rightarrow$ *Upper Bound*

A Lower Bound on Speed

Given a list of $\mathbf{n}$ elements, what is the minimum amount of time a computer may take to sort it?

MergeSort average complexity is O($\mathbf{n}$ log $\mathbf{n}$).

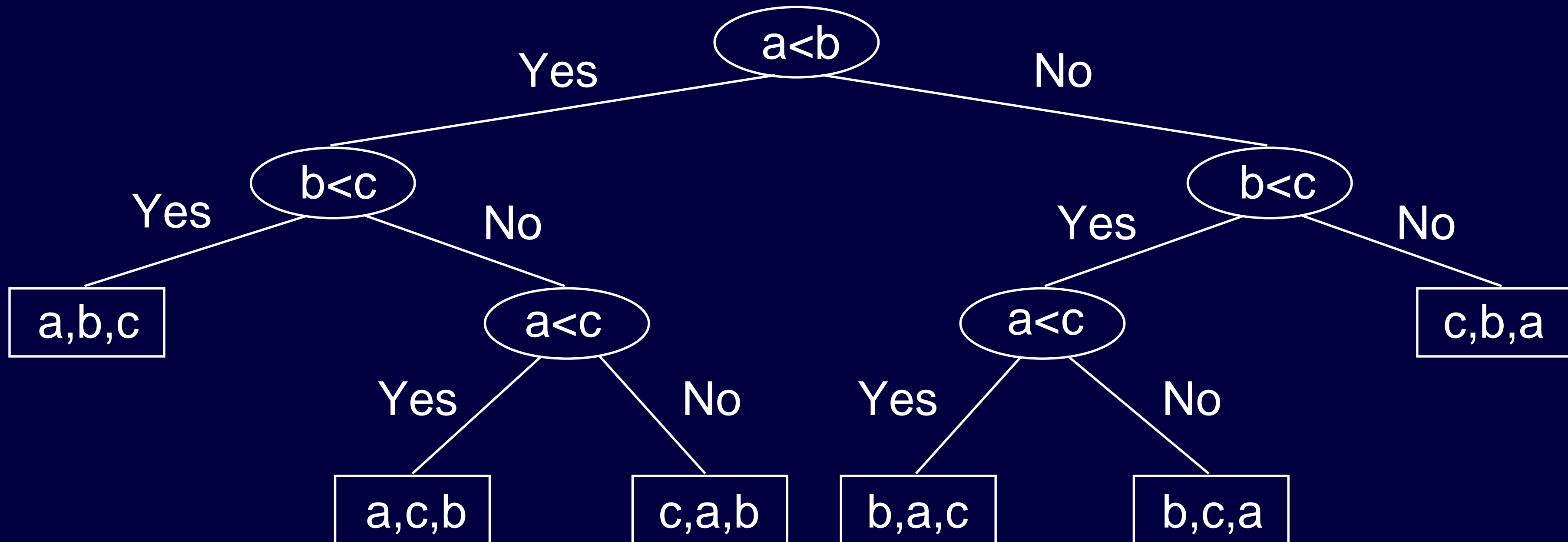
Means, that for large enough $\mathbf{n}$ the number of steps this algorithm takes to sort $\mathbf{n}$ elements is never greater than c * $\mathbf{n}$ log $\mathbf{n}$, for some constant c.

$\Rightarrow$ *Upper Bound*

= This is also the *Lower Bound* on the number of steps necessary for *comparison-based* sorting algorithms.

Comparison Trees

Comparison-based sorting algorithms perform comparisons between pairs of values. $\Rightarrow$ Results in a **comparison tree**!



Comparisons interleaved with re-arrangements.

- Number of comparisons needed is identical to **depth** of comparison tree.

Min Average Comparisons for Sorting

Sorting 3 values results in 6 (= 3!) distinct sorted orders.

In general: $\mathbf{n}!$ distinct orders for $\mathbf{n}$ values

$\Rightarrow$ Number of distinct terminal nodes in comparison tree is $\mathbf{n}!$

A binary tree with $\mathbf{m}$ terminal nodes has a depth of at least log $\mathbf{m}$.

Depth $\mathbf{D}$ of comparison tree (which is a binary tree) must satisfy:

$$\begin{aligned} \mathbf{D} &> \log \mathbf{n}! \\ &> \log \mathbf{n} * (\mathbf{n}-\mathbf{1}) * (\mathbf{n}-\mathbf{2}) * \ldots * \mathbf{2} * \mathbf{1} \\ &> \mathbf{n}/\mathbf{2} \log \mathbf{n} \qquad \text{by approximation} \\ &\Rightarrow O(\mathbf{n} \log \mathbf{n}) \end{aligned}$$

Means: Minimum average number of comparisons required to sort $\mathbf{n}$ values using a comparison-based sorting method is $\mathbf{n}$ log $\mathbf{n}$.

Where to get more information?

Thomas A. Standish
“Data Structures, Algorithms & Software Principles in C”
Addison-Wesley, 1995
Chapter 13: Sorting (p. 524 ff)

A. D. Dewdney **“The New Turing Omnibus”**
Computer Science Press, 1997
Chapters 35, 40 on Sorting Problems and Algorithms