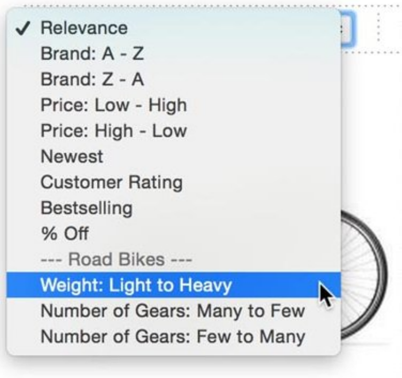
Sorting algorithms

A Ammar, A Scemama, P Reinhardt, Y Damour

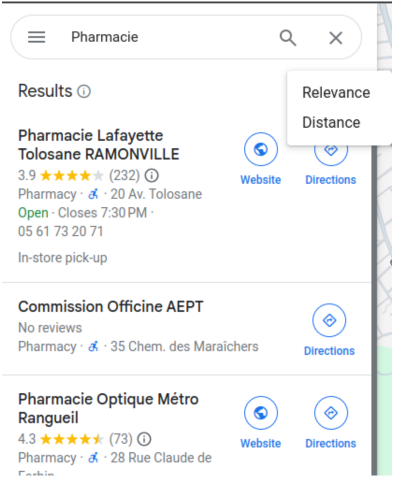
November 21, 2024

Sorting for Sorting

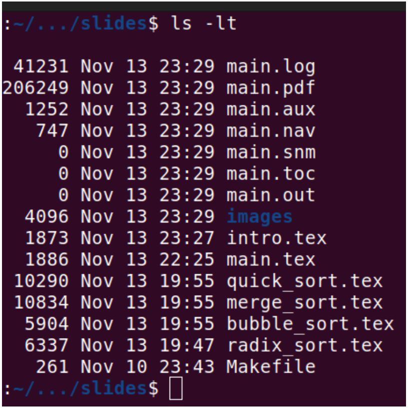
- Sorting algorithms are used in everyday applications



Sorting for Sorting

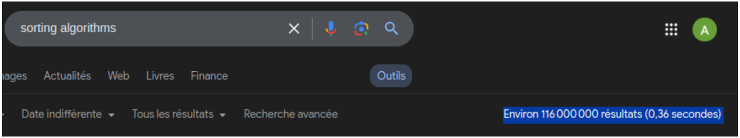


Sorting for Sorting



Sorting for Efficiency

- Sorting algorithms are employed for more than just sorting



Sorting for Efficiency

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

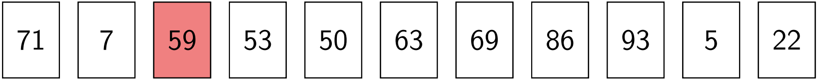
Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

Linear vs Binary search

- Find 69 ?



Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

➥ **time scaling** $\mathcal{O}(N)$

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

➥ **time scaling** $\mathcal{O}(N)$

5	7	50	22	53	59	63	69	73	86	93

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

➥ **time scaling** $\mathcal{O}(N)$

5	7	50	22	53	59	63	69	73	86	93

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

➥ **time scaling** $\mathcal{O}(N)$

5	7	50	22	53	59	63	69	73	86	93

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

➥ **time scaling** $\mathcal{O}(N)$

5	7	50	22	53	59	63	69	73	86	93

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

➥ **time scaling** $\mathcal{O}(N)$

5	7	50	22	53	59	63	69	73	86	93

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

➥ **time scaling** $\mathcal{O}(N)$

5	7	50	22	53	59	63	69	73	86	93

Linear vs Binary search

- Find 69 ?

71	7	59	53	50	63	69	86	93	5	22

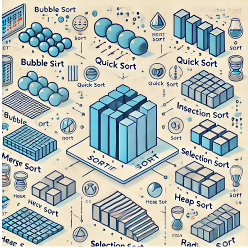
➥ **time scaling** $\mathcal{O}(N)$

5	7	50	22	53	59	63	69	73	86	93

➥ **time scaling** $\mathcal{O}(\log N)$

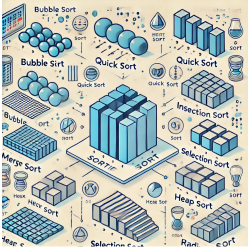
🧐 $\log(116\,000\,000) \approx 18$ ‼️

Best Sorting Algorithm ?



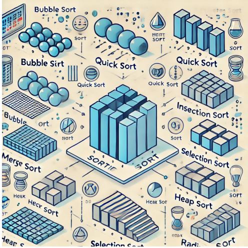
- Key considerations for sorting algorithms:

Best Sorting Algorithm ?



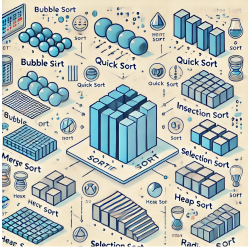
- Key considerations for sorting algorithms:
  - Time complexity: $\mathcal{O}(N^2)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(kN)$, ...

Best Sorting Algorithm ?



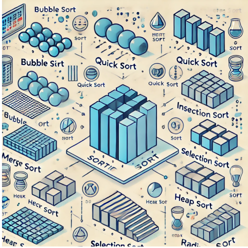
- Key considerations for sorting algorithms:
  - Time complexity: $\mathcal{O}(N^2)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(kN)$, ...
  - Space complexity (memory usage): $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, ...

Best Sorting Algorithm ?



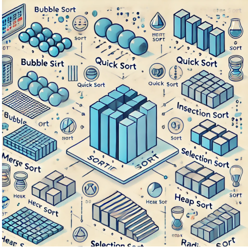
- Key considerations for sorting algorithms:
  - Time complexity: $\mathcal{O}(N^2)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(kN)$, ...
  - Space complexity (memory usage): $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, ...
  - Stability: ❷ ② $\neq$ ② ❷

Best Sorting Algorithm ?



- Key considerations for sorting algorithms:
  - Time complexity: $\mathcal{O}(N^2)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(kN)$, ...
  - Space complexity (memory usage): $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, ...
  - Stability: ❷ ② $\neq$ ② ❷
  - Adaptiveness (best, worst, and average cases)

Best Sorting Algorithm ?



- Key considerations for sorting algorithms:
  - Time complexity: $\mathcal{O}(N^2)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(kN)$, ...
  - Space complexity (memory usage): $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, ...
  - Stability: ❷ ② $\neq$ ② ❷
  - Adaptiveness (best, worst, and average cases)
  - Online vs. Offline sorting

1. Bubble Sort
2. Merge sort
3. Quick sort
4. Radix sort

Bubble Sort

Bubble Sort Algorithm

- Goal: Sort an array of n items
- Algorithm:
    1. Compare first pair of adjacent items
    2. Swap if they are in the wrong order

Bubble Sort Algorithm

- Goal: Sort an array of n items
- Algorithm:
  1. Compare first pair of adjacent items
  2. Swap if they are in the wrong order
  3. Iterate over next pairs
     - Largest item "bubbles" to the end

Bubble Sort Algorithm

- Goal: Sort an array of n items
- Algorithm:
  1. Compare first pair of adjacent items
  2. Swap if they are in the wrong order
  3. Iterate over next pairs
     - Largest item "bubbles" to the end
  4. Reduce n by 1 and go to step 1

Illustration

First pass

25	13	4	7	16

Illustration

First pass

25	13	4	7	16

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

13	4	7	25	16

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

13	4	7	25	16

Illustration

First pass

25	13	4	7	16
13	25	4	7	16
13	4	25	7	16
13	4	7	25	16
13	4	7	16	25

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

13	4	7	25	16

13	4	7	16	25

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

13	4	7	25	16

13	4	7	16	25

Second pass

13	4	7	16	25

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

13	4	7	25	16

13	4	7	16	25

Second pass

13	4	7	16	25

4	13	7	16	25

Illustration

First pass

25	13	4	7	16

13	25	4	7	16

13	4	25	7	16

13	4	7	25	16

13	4	7	16	25

Second pass

13	4	7	16	25

4	13	7	16	25

4	7	13	16	25

Illustration

First pass

25	13	4	7	16
13	25	4	7	16
13	4	25	7	16
13	4	7	25	16
13	4	7	16	25

Second pass

13	4	7	16	25
4	13	7	16	25
4	7	13	16	25
4	7	13	16	25

Naive implementation

Data: Array A of n elements
Result: Sorted array A

Naive implementation

Data: Array A of n elements
Result: Sorted array A
loop over passes

Naive implementation

Data: Array A of n elements
Result: Sorted array A
# loop over passes
for $i = 0$ **to** $n - 2$ **do**

Naive implementation

Data: Array A of n elements
Result: Sorted array A
# loop over passes
for $i = 0$ **to** $n - 2$ **do**
    # compares adjacent elements

Naive implementation

Data: Array A of n elements
Result: Sorted array A

```
# loop over passes
for i = 0 to n − 2 do
    # compares adjacent elements
    for j = 0 to n − 2 − i do
```

Naive implementation

Data: Array A of n elements
Result: Sorted array A

```
# loop over passes
for i = 0 to n − 2 do
    # compares adjacent elements
    for j = 0 to n − 2 − i do
        # bubble largest element
```

Naive implementation

Data: Array A of n elements
Result: Sorted array A

```
# loop over passes
for i = 0 to n − 2 do
    # compares adjacent elements
    for j = 0 to n − 2 − i do
        # bubble largest element
        if A[j] > A[j + 1] then
            Swap A[j] and A[j + 1];
```

Illustration

First pass

25	13	4	7	16
13	25	4	7	16
13	4	25	7	16
13	4	7	25	16
13	4	7	16	25

Second pass

13	4	7	16	25
4	13	7	16	25
4	7	13	16	25
4	7	13	16	25

Illustration

First pass

25	13	4	7	16
13	25	4	7	16
13	4	25	7	16
13	4	7	25	16
13	4	7	16	25

Second pass

13	4	7	16	25
4	13	7	16	25
4	7	13	16	25
4	7	13	16	25

Third pass

4	7	13	16	25

Illustration

First pass

25	13	4	7	16
13	25	4	7	16
13	4	25	7	16
13	4	7	25	16
13	4	7	16	25

Second pass

13	4	7	16	25
4	13	7	16	25
4	7	13	16	25
4	7	13	16	25

Third pass

4	7	13	16	25
4	7	13	16	25

Illustration

First pass

25	13	4	7	16
13	25	4	7	16
13	4	25	7	16
13	4	7	25	16
13	4	7	16	25

Second pass

13	4	7	16	25
4	13	7	16	25
4	7	13	16	25
4	7	13	16	25

Third pass

4	7	13	16	25
4	7	13	16	25
4	7	13	16	25

Improved implementation

```
Data: Array A of n elements
Result: Sorted array A
for i = 0 to n − 2 do
    is_sorted = true;
    for j = 0 to n − 2 − i do
        if A[j] > A[j + 1] then
            Swap A[j] and A[j + 1];
            is_sorted = false;
    if is_sorted then
        return;
```

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations
    - for $i = 1$: $n-2$ iterations

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations
    - for $i = 1$: $n-2$ iterations
    - $\ldots$
    - for $i = n-2$: 1 iteration

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations
    - for $i = 1$: $n-2$ iterations
    - $\ldots$
    - for $i = n-2$: 1 iteration
  - Total number of iterations: $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations
    - for $i = 1$: $n-2$ iterations
    - $\ldots$
    - for $i = n-2$: 1 iteration
  - Total number of iterations: $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$
- Total time: $\mathcal{O}(c\, n(n-1))$

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations
    - for $i = 1$: $n-2$ iterations
    - $\ldots$
    - for $i = n-2$: 1 iteration
  - Total number of iterations: $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$
- Total time: $\mathcal{O}(c\, n(n-1)) = \mathcal{O}(n^2 - n)$

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations
    - for $i = 1$: $n-2$ iterations
    - $\ldots$
    - for $i = n-2$: 1 iteration
  - Total number of iterations: $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$
- Total time: $\mathcal{O}(c\, n(n-1)) = \mathcal{O}(n^2 - n) = \mathcal{O}(n^2)$

Analysis

- Time complexity:
  - Outer loop (i): $0, 1, \ldots, n-2$
  - Inner loop (j):
    - for $i = 0$: $n-1$ iterations
    - for $i = 1$: $n-2$ iterations
    - $\ldots$
    - for $i = n-2$: 1 iteration
  - Total number of iterations: $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$
- Total time: $\mathcal{O}(c\, n(n-1)) = \mathcal{O}(n^2 - n) = \mathcal{O}(n^2)$
- We need a constant amount of memory: $\mathcal{O}(1)$

Merge sort

Idea

- Merge Sort is a divide-and-conquer algorithm
  1. Divide: Recursively divide the array into two halves
  2. Conquer: Merge the two halves

Idea

- Merge Sort is a divide-and-conquer algorithm
  1. Divide: Recursively divide the array into two halves
  2. Conquer: Merge the two halves
- Suppose time scaling is $\mathcal{O}(n^2)$
  - for n: $\rightarrow$ 1 day
  - for $n/2$: $\rightarrow 1/\mathbf{4}$ day

Idea

- Merge Sort is a divide-and-conquer algorithm
  1. Divide: Recursively divide the array into two halves
  2. Conquer: Merge the two halves
- Suppose time scaling is $\mathcal{O}(n^2)$
  - for n: $\rightarrow$ 1 day
  - for $n/2$: $\rightarrow 1/\mathbf{4}$ day
- Analysis (homework 😉):
  - Time Complexity: $\mathcal{O}(n \log n)$
  - Space Complexity: $\mathcal{O}(n)$

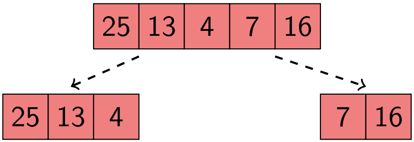
Illustration

25	13	4	7	16

Divide

Conquer

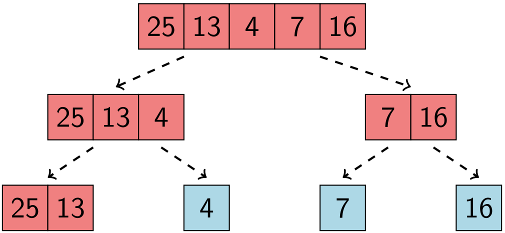
Illustration



Divide

Conquer

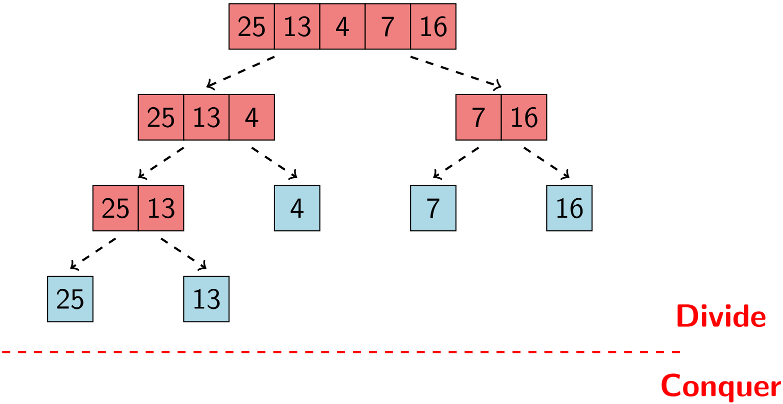
Illustration



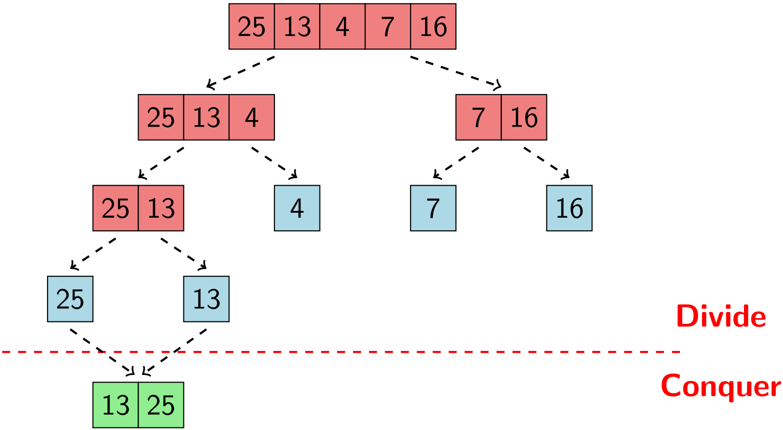
Divide

Conquer

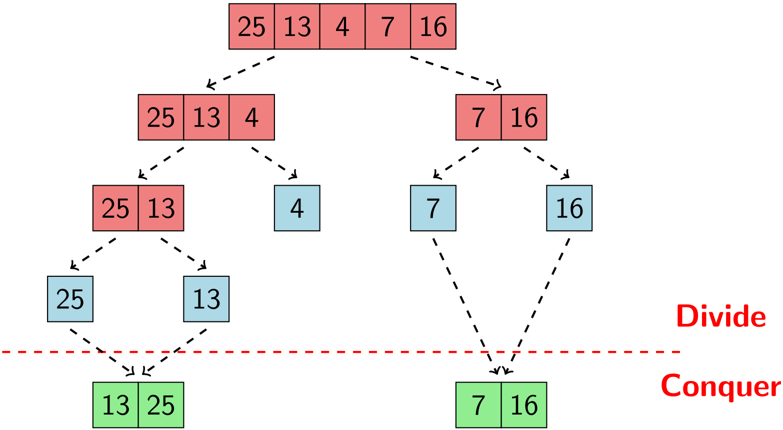
Illustration



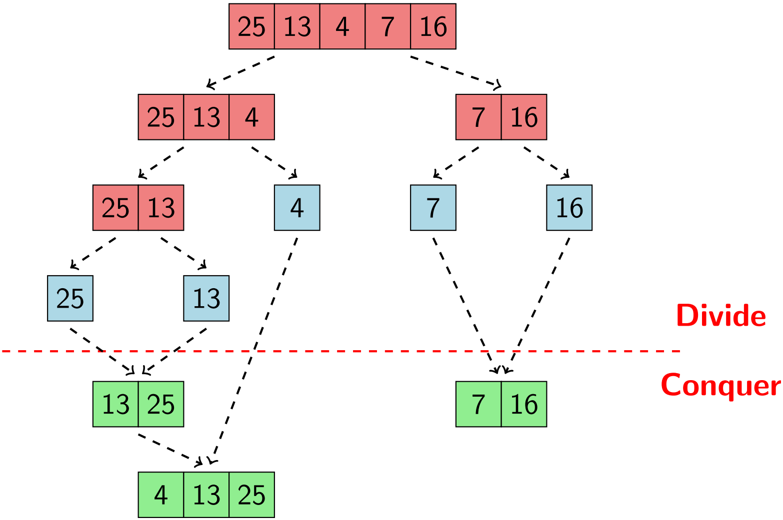
Illustration



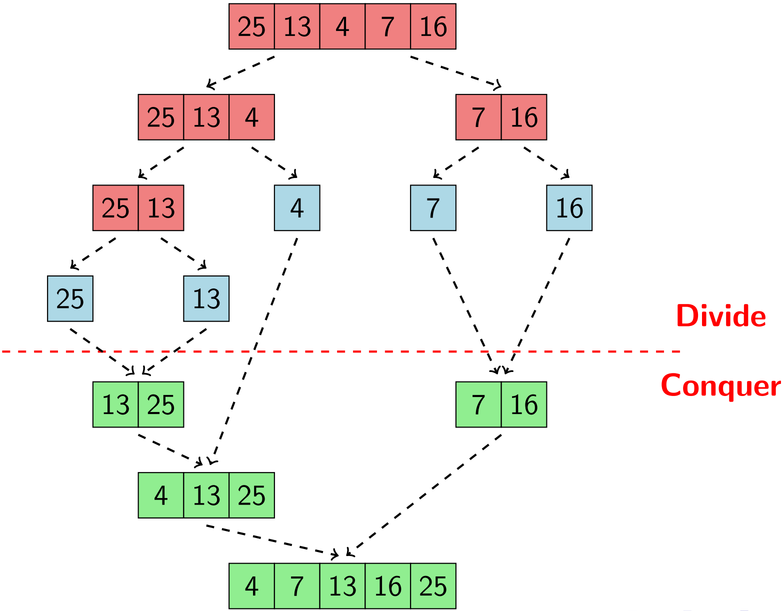
Illustration



Illustration



Illustration



MergeSort Algorithm

Data: Array A, left and right indices
Result: Sorted array A

MergeSort Algorithm

Data: Array A, left and right indices
Result: Sorted array A
\# to stop recursion
if left $<$ right **then**

MergeSort Algorithm

Data: Array A, left and right indices
Result: Sorted array A
\# to stop recursion
if left $<$ right **then**
  \# middle index
  $\text{mid} = \text{left} + \left\lfloor \frac{\text{right}-\text{left}}{2} \right\rfloor$;

MergeSort Algorithm

Data: Array A, left and right indices
Result: Sorted array A

```
# to stop recursion
if left < right then
    # middle index
    mid = left + ⌊(right−left)/2⌋;
    # recursively sort the two halves
    MergeSort(A, left, mid);
    MergeSort(A, mid+1, right);
```

Where $\text{mid} = \text{left} + \left\lfloor \frac{\text{right}-\text{left}}{2} \right\rfloor$;

MergeSort Algorithm

```
Data: Array A, left and right indices
Result: Sorted array A
# to stop recursion
if left < right then
    # middle index
    mid = left + ⌊(right−left)/2⌋;
    # recursively sort the two halves
    MergeSort(A, left, mid);
    MergeSort(A, mid+1, right);
    # merge the two halves
    merge(A, left, mid, right);
```

Merge step

4	13	25

7	16

Merge step

4	13	25

7	16

Merge step

4	13	25

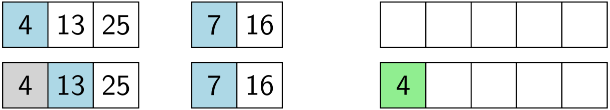
7	16

4	13	25

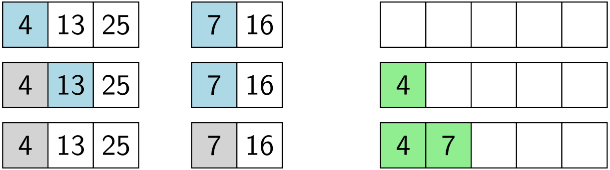
7	16

4				

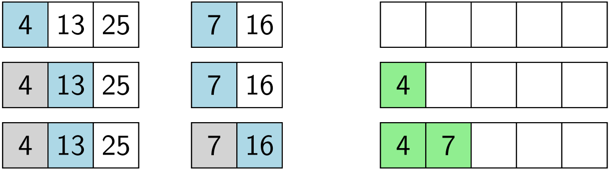
Merge step



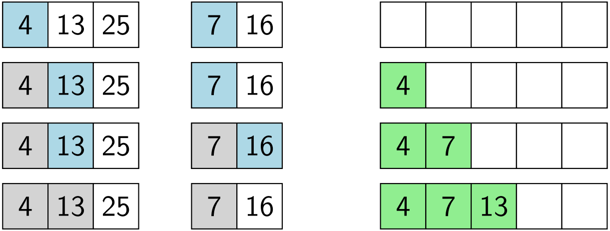
Merge step



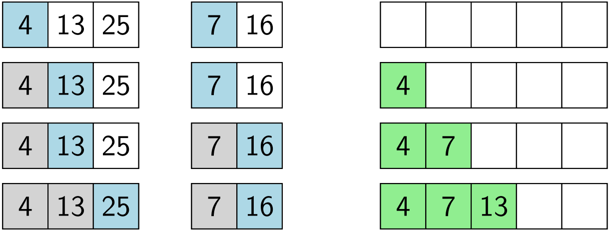
Merge step



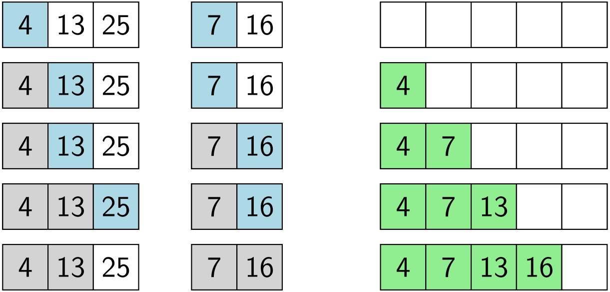
Merge step



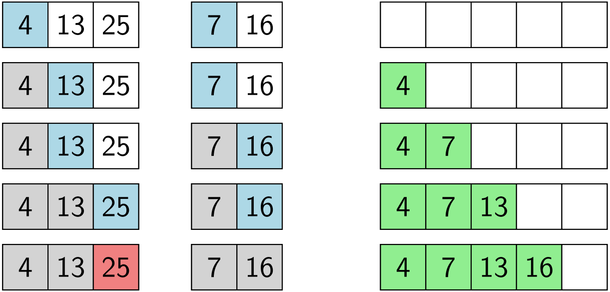
Merge step



Merge step

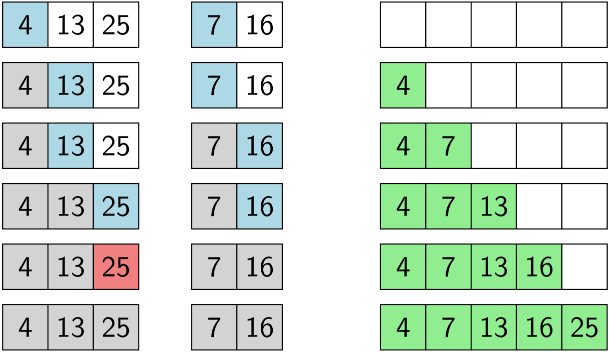


Merge step



4 13 25
7 16
4 13 25
7 16
4
4 13 25
7 16
4 7
4 13 25
7 16
4 7 13
4 13 25
7 16
4 7 13 16

Merge step


4 13 25
7 16
4 13 25
7 16
4
4 13 25
7 16
4 7
4 13 25
7 16
4 7 13
4 13 25
7 16
4 7 13 16
4 13 25
7 16
4 7 13 16 25

Quick sort

Idea

- Quick Sort is a divide-and-conquer algorithm
  1. Divide Step: choose a “pivot”
     - the pivot divides the array into elements $< p$ and those $\geq p$
  2. Conquer Step: Do nothing!

Idea

- Quick Sort is a divide-and-conquer algorithm
  1. Divide Step: choose a "pivot"
     - the pivot divides the array into elements $< p$ and those $\geq p$
  2. Conquer Step: Do nothing!
- *Merge Sort* spends more time in the *conquer step*, while *Quick Sort* spends more time in the *divide step*

Idea

- Quick Sort is a divide-and-conquer algorithm
  1. Divide Step: choose a "pivot"
     - the pivot divides the array into elements $< p$ and those $\geq p$
  2. Conquer Step: Do nothing!
- *Merge Sort* spends more time in the *conquer step*, while *Quick Sort* spends more time in the *divide step*
- Analysis (homework 😉):
  - Time Complexity: $\mathcal{O}(n \log n)$
  - Space Complexity: $\mathcal{O}(\log n)$

QuickSort Algorithm

Data: Array A, left and right indices
Result: Sorted array A

QuickSort Algorithm

Data: Array A, left and right indices

Result: Sorted array A

\# to stop recursion

if left $<$ right **then**

QuickSort Algorithm

Data: Array A, left and right indices
Result: Sorted array A

```
# to stop recursion
if left < right then
    # partition the array
    piv_index = partition(A, left, right);
```

QuickSort Algorithm

Data: Array A, left and right indices
Result: Sorted array A

```
# to stop recursion
if left < right then
    # partition the array
    piv_index = partition(A, left, right);
    # recursively sort the two halves
    QuickSort(A, left, piv_index - 1);
    QuickSort(A, piv_index + 1, right);
```

Illustration

10	61	46	56	1	35	51	55	22

Illustration

10	61	46	56	1	35	51	55	22

Illustration

10	61	46	56	1	35	51	55	22

Illustration

10	61	46	56	1	35	51	55	22

10								

Illustration

10	61	46	56	1	35	51	55	22

10								

Illustration

10	61	46	56	1	35	51	55	22

10								

Illustration

10	61	46	56	1	35	51	55	22

10								

Illustration

10	61	46	56	1	35	51	55	22

10	1							

Illustration

10	61	46	56	1	35	51	55	22

10	1			61				

Illustration

10	61	46	56	1	35	51	55	22

10	1			61				

Illustration

10	61	46	56	1	35	51	55	22

10	1			61				

Illustration

10	61	46	56	1	35	51	55	22

10	1			61				

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22		61				

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22		61				46

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

pivot pivot

1		22		46				

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

pivot **pivot**

1	10	22	35	46	56	51	55	61

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

pivot **pivot**

1	10	22	35	46	56	51	55	61

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

pivot **pivot**

1	10	22	35	46	56	51	55	61

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

pivot **pivot**

1	10	22	35	46	56	51	55	61

pivot

1	10	22	35	46	56	51	55	61

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

pivot **pivot**

1	10	22	35	46	56	51	55	61

pivot

1	10	22	35	46	56	51	55	61

Illustration

10	61	46	56	1	35	51	55	22

pivot

10	1	22	56	61	35	51	55	46

pivot **pivot**

1	10	22	35	46	56	51	55	61

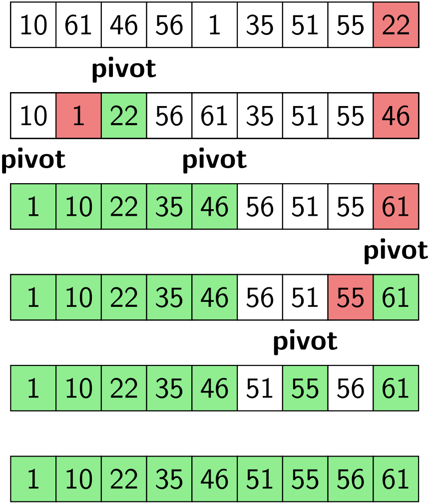
pivot

1	10	22	35	46	56	51	55	61

pivot

1	10	22	35	46	51	55	56	61

Illustration



Radix sort

Idea

- Radix (root): base in which we express an integer
  - Radix 10, Radix 2, ...
  - from right(LSD $\rightarrow$ MSD), from left (MSD $\rightarrow$ LSD)

Idea

- Radix (root): base in which we express an integer
  - Radix 10, Radix 2, ...
  - from right(LSD $\rightarrow$ MSD), from left (MSD $\rightarrow$ LSD)
- *Counting Sort*:
  - Non-comparative sorting (no **direct** use of $<, >$)
  - treats data as a "character" string (digit, bit, ...)

Idea

- Radix (root): base in which we express an integer
  - Radix 10, Radix 2, ...
  - from right(LSD $\rightarrow$ MSD), from left (MSD $\rightarrow$ LSD)
- *Counting Sort*:
  - Non-comparative sorting (no **direct** use of $<, >$)
  - treats data as a “character” string (digit, bit, ...)
- Analysis
  - Time Complexity: $\mathcal{O}(nk)$, (k: # of bits of largest number)
  - Space Complexity: $\mathcal{O}(n+k)$, $\mathcal{O}(nk)$, ...

Illustration

3	6	7	8	2

Illustration

3	6	7	8	2

0011	0110	0111	1000	0010

Illustration

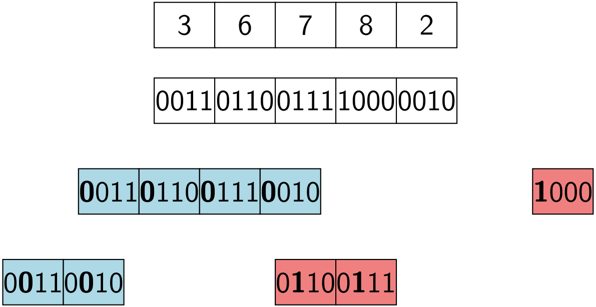
3	6	7	8	2

0011	0110	0111	1000	0010

0011	**0**110	**0**111	**0**010

1000

Illustration



Illustration

3	6	7	8	2

0011	0110	0111	1000	0010

0011	**0**110	**0**111	**0**010

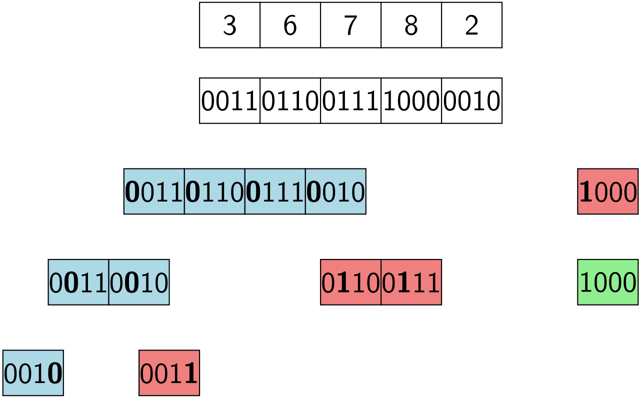
1000

0**0**11	0**0**10

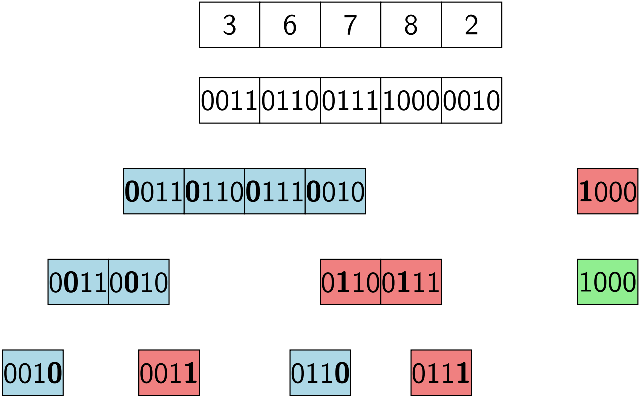
0**1**10	0**1**11

1000

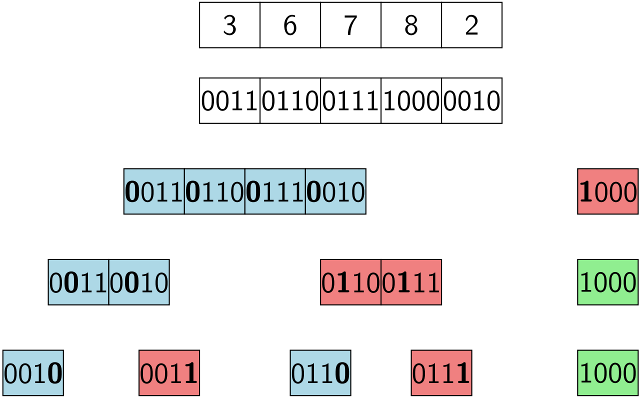
Illustration



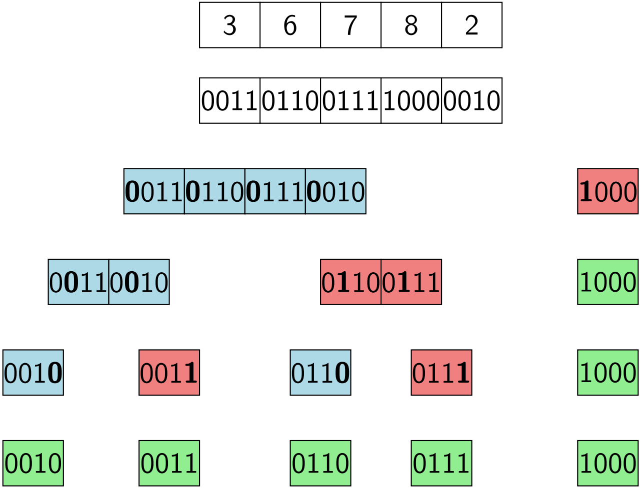
Illustration



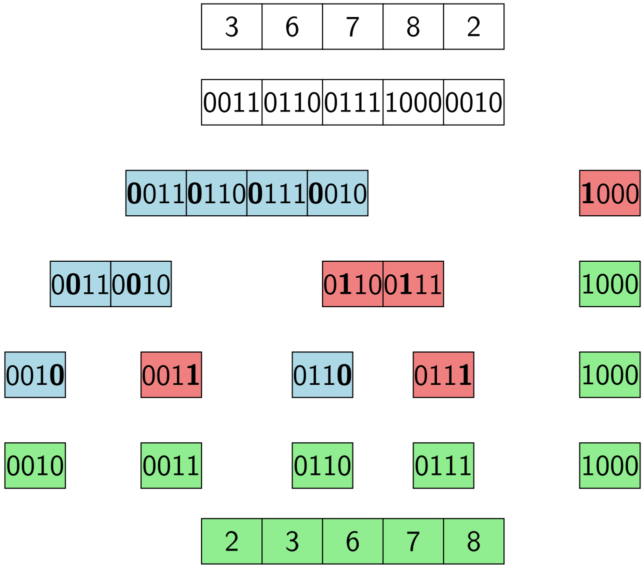
Illustration



Illustration



Illustration



Ref

- *Introduction to Algorithms*. T. Cormen, C. Leiserson, R. Rivest, C. Stein
- *The Art of Computer Programming*, Vol. 3. D. Knuth.
- for fun
  - `https://www.toptal.com/developers/sorting-algorithms`
  - `https://www.youtube.com/watch?v=kPRA0W1kECg`