Design and Analysis of Algorithms

Presented by Dr. Li Ning

Shenzhen Institutes of Advanced Technology, Chinese Academy of Science
Shenzhen, China

Introduction

1 Complexity: The Big-O Notation

2 Sort The Numbers

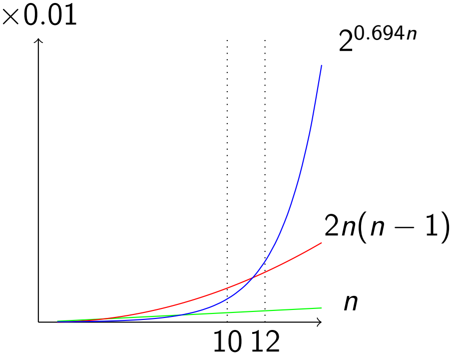
3 Divide and Conquer

Complexity: The Big-O Notation

Number of Operations

Which one is larger?

- Addition: n
- Multiplication: $2n(n-1)$
- Fibonacci: $2^{0.694n}$

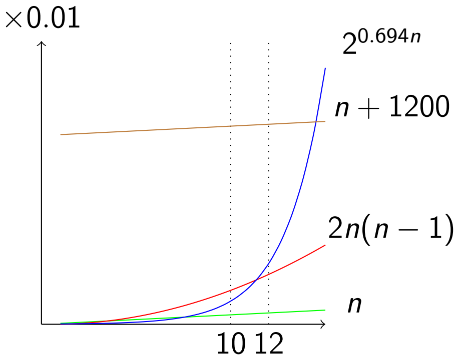


Number of Operations

Which one is larger?

- Addition: n
- Multiplication: $2n(n-1)$
- Fibonacci: $2^{0.694n}$

The ignored operations: if-test, return, ...



O-Notation

Consider a function $T(n) \geq 0$. We said it has upper bound $O(f(n))$, denoted by

$$T(n) = O(f(n)),$$

if and only if $\exists C > 0, N > 0$, s.t.

$$T(n) \leq Cf(n), \forall n \geq N.$$

O-Notation

- $T(n) = n$. Let $f(n) = n, C = 1, N = 1$. Then

$$T(n) \leq 1 \cdot n, \forall n \geq 1 \Rightarrow T(n) = O(n).$$

O-Notation

- $T(n) = 2n(n-1)$. Let $f(n) = n^2, C = 2, N = 1$. Then

$$T(n) \leq 2 \cdot n^2, \forall n \geq 1 \Rightarrow T(n) = O(n^2).$$

O-Notation

- $T(n) = 2^{0.694n}$. Let $f(n) = 2^n, C = 1, N = 1$. Then

$$T(n) \leq 1 \cdot 2^n, \forall n \geq 1 \Rightarrow T(n) = O(2^n).$$

O-Notation

- $T(n) = n$. Let $f(n) = n, C = 1, N = 1$. Then
$$T(n) \le 1 \cdot n, \forall n \ge 1 \Rightarrow T(n) = O(n).$$
- $T(n) = 2n(n-1)$. Let $f(n) = n^2, C = 2, N = 1$. Then
$$T(n) \le 2 \cdot n^2, \forall n \ge 1 \Rightarrow T(n) = O(n^2).$$
- $T(n) = 2^{0.694n}$. Let $f(n) = 2^n, C = 1, N = 1$. Then
$$T(n) \le 1 \cdot 2^n, \forall n \ge 1 \Rightarrow T(n) = O(2^n).$$

Let $T(n) = 0.694n$. Then
$$T(n) = O(n), \text{ and } 2^{0.694n} = 2^{O(n)}.$$

Ω-Notation

Consider a function $T(n) \geq 0$. We said it has lower bound $\Omega(g(n))$, denoted by

$$T(n) = \Omega(g(n)),$$

if and only if $\exists C > 0, N > 0$, s.t.

$$T(n) \geq Cg(n), \forall n \geq N.$$

Ω-Notation

- $T(n) = n$. Let $g(n) = n, C = 1, N = 1$. Then

$$T(n) \geq 1 \cdot n, \forall n \geq 1 \Rightarrow T(n) = \Omega(n).$$

Ω-Notation

- $T(n) = n$. Let $g(n) = n, C = 1, N = 1$. Then

$$T(n) \geq 1 \cdot n, \forall n \geq 1 \Rightarrow T(n) = \Omega(n).$$

- $T(n) = 2n(n-1)$. Let $g(n) = n^2, C = 1, N = 2$. Then

$$T(n) = n^2 + n(n-2) \geq 1 \cdot n^2, \forall n \geq 2 \Rightarrow T(n) = \Omega(n^2).$$

Ω-Notation

- $T(n) = n$. Let $g(n) = n, C = 1, N = 1$. Then

$$T(n) \geq 1 \cdot n, \forall n \geq 1 \Rightarrow T(n) = \Omega(n).$$

- $T(n) = 2n(n-1)$. Let $g(n) = n^2, C = 1, N = 2$. Then

$$T(n) = n^2 + n(n-2) \geq 1 \cdot n^2, \forall n \geq 2 \Rightarrow T(n) = \Omega(n^2).$$

- $T(n) = 0.694n$. Let $g(n) = n, C = 0.5, N = 1$. Then

$$T(n) \geq 0.5 \cdot n, \forall n \geq 1 \Rightarrow T(n) = \Omega(n),$$

  and

$$2^{0.694n} = 2^{\Omega(n)}.$$

Θ-Notation

Consider a function $T(n) \geq 0$. We said it has the same order with $\Theta(f(n))$, denoted by

$$T(n) = \Theta(f(n)),$$

if and only if $\exists C_2 > C_1 > 0, N > 0$, s.t.

$$C_1 f(n) \leq T(n) \leq C_2 f(n), \forall n \geq N.$$

Θ-Notation

Equivalently,

$$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)).$$

Θ-Notation

Equivalently,

$$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)).$$

- $T(n) = n = O(n) = \Omega(n)$, thus

$$n = \Theta(n).$$

Θ-Notation

Equivalently,

$$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)).$$

- $T(n) = n = O(n) = \Omega(n)$, thus
$$n = \Theta(n).$$
- $T(n) = 2n(n-1) = O(n^2) = \Omega(n^2)$, thus
$$2n(n-1) = \Theta(n^2).$$

Θ-Notation

Equivalently,

$$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)).$$

- $T(n) = n = O(n) = \Omega(n)$, thus
$$n = \Theta(n).$$
- $T(n) = 2n(n-1) = O(n^2) = \Omega(n^2)$, thus
$$2n(n-1) = \Theta(n^2).$$
- $T(n) = 2^{0.694n} = 2^{O(n)} = 2^{\Omega(n)}$, thus
$$2^{0.694n} = 2^{\Theta(n)}.$$

o-Notation and ω-Notation

Consider a function $T(n) \geq 0$. $T(n) = o(f(n))$ if and only if $\forall C > 0, \exists N > 0$, s.t.

$$T(n) < Cf(n), \forall n \geq N.$$

Equivalently,

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0.$$

o-Notation and ω-Notation

Consider a function $T(n) \geq 0$. $T(n) = \omega(g(n))$ if and only if $\forall C > 0, \exists N > 0$, s.t.

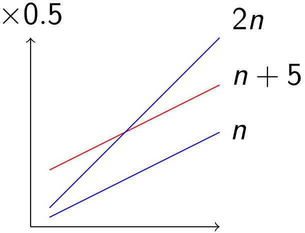
$$T(n) > Cg(n), \forall n \geq N.$$

Equivalently,

$$\lim_{n \to \infty} \frac{T(n)}{g(n)} = \infty.$$

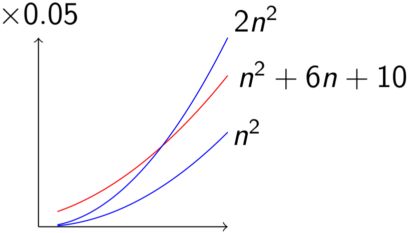
Examples

$$\begin{aligned} T(n) &= n + 5 \\ &= \Theta(n) \end{aligned}$$



Examples

$$
\begin{aligned}
T(n) &= {\color{red} n^2 + 6n + 10} \\
&= {\color{blue} \Theta(n^2)}
\end{aligned}
$$



Examples

$f(n) = an^2 + bn + c = \Theta(n^2)$, for $a > 0$.

Examples

$f(n) = an^2 + bn + c = \Theta(n^2)$, for $a > 0$.

- Let $N = 2 \cdot \max\{|b|/a, \sqrt{|c|/a}\}$.

Examples

$f(n) = an^2 + bn + c = \Theta(n^2)$, for $a > 0$.

- Let $N = 2 \cdot \max\{|b|/a, \sqrt{|c|/a}\}$.
- $f(n) = O(n^2)$: With $C = 7a/4$, for $n > N$, $an^2 > 4|c| \text{ and } an^2 > 2|b|n$, and then

$$an^2 + bn + c < (1 + \frac{1}{2} + \frac{1}{4})an^2 = \frac{7a}{4}n^2$$

Examples

$f(n) = an^2 + bn + c = \Theta(n^2)$, for $a > 0$.

- Let $N = 2 \cdot \max\{|b|/a, \sqrt{|c|/a}\}$.
- $f(n) = O(n^2)$: With $C = 7a/4$, for $n > N$, $an^2 > 4|c|$ and $an^2 > 2|b|n$, and then

$$an^2 + bn + c < (1 + \frac{1}{2} + \frac{1}{4})an^2 = \frac{7a}{4}n^2$$

- $f(n) = \Omega(n^2)$: With $C = a/4$, for $n > N$, $4c > -an^2$ and $2bn > -an^2$, and then

$$an^2 + bn + c > (1 - \frac{1}{2} - \frac{1}{4})an^2 = \frac{a}{4}n^2$$

Examples

$f(n) = \sum_{i=0}^{k} a_i n^i = \Theta(n^k)$, where $a_i > 0$.

$f(n) = an^k = O(n^{k+1})$, for $a > 0$.

Examples

$f(n) = \sum_{i=0}^{k} a_i n^i = \Theta(n^k)$, where $a_i > 0$.

$f(n) = an^k = O(n^{k+1})$, for $a > 0$.

C?, N?

Examples

$f(n) = \sum_{i=0}^{k} a_i n^i = \Theta(n^k)$, where $a_i > 0$.

$f(n) = an^k = O(n^{k+1})$, for $a > 0$.

$C = 1$, $N = \lceil a \rceil$, then for $n > N$,

$$a \cdot n^k < n \cdot n^k = n^{k+1}$$

Examples

$$f(n) = 6n^3 \neq \Theta(n^2).$$

Examples

$f(n) = 6n^3 \neq \Theta(n^2)$.

- Assume there exists $C > 0$ and $N > 0$, such that for $n \geq N$

$$6n^3 \leq Cn^2$$

Examples

$f(n) = 6n^3 \neq \Theta(n^2)$.

- Assume there exists $C > 0$ and $N > 0$, such that for $n \geq N$

$$6n^3 \leq Cn^2$$

- Let $n_* = \max\{C/6, N\} + 1 > N$,

$$6n_*^3 > Cn_*^2$$

Examples

$f(n) = 6n^3 \neq \Theta(n^2)$.

- Assume there exists $C > 0$ and $N > 0$, such that for $n \geq N$
$$6n^3 \leq Cn^2$$
- Let $n_* = \max\{C/6, N\} + 1 > N$,
$$6n_*^3 > Cn_*^2$$
- **Contradiction**!

Sort The Numbers

A Sequence of Numbers

Sort

5	3	1	2	4

to

1	2	3	4	5

Insertion Sort

5	3	1	2	4

1. Sort the first 2 numbers: compare the first number and the second number.

Insertion Sort

$\boxed{5}\quad\boxed{3}\quad\boxed{1}\quad\boxed{2}\quad\boxed{4}$

$\boxed{3}\quad\boxed{5}\quad\boxed{1}\quad\boxed{2}\quad\boxed{4}$

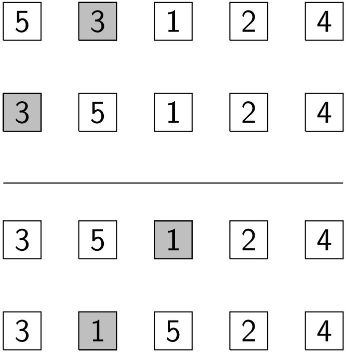
1. Sort the first 2 numbers: compare the first number and the second number.

Insertion Sort

5	3	1	2	4
3	5	1	2	4
3	5	1	2	4

2. Sort the first 3 numbers: insert the third number into the sorted numbers.

Insertion Sort



2. Sort the first 3 numbers: insert the third number into the sorted numbers.

Insertion Sort

2. Sort the first 3 numbers: insert the third number into the sorted numbers.

5	3	1	2	4
3	5	1	2	4

3	5	1	2	4
3	1	5	2	4
1	3	5	2	4

Insertion Sort

1	3	5	2	4

1. Sort the first 4 numbers.

Insertion Sort

1	3	5	2	4
1	3	2	5	4

1. Sort the first 4 numbers.

Insertion Sort

1	3	5	2	4
1	3	2	5	4
1	2	3	5	4

1. Sort the first 4 numbers.

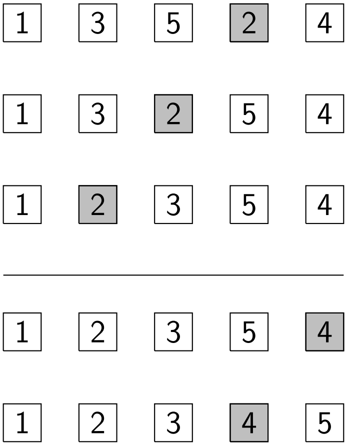
Insertion Sort

2. Sort the first 5 numbers.

1	3	5	2	4
1	3	2	5	4
1	2	3	5	4

1	2	3	5	4

Insertion Sort



2. Sort the first 5 numbers.

Insertion Sort : $O(n^2)$

5	3	1	2	4
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$

- **for**-loop run for $n-1$ times.
- **while**-loop run for at most i times.
- inside 1 loop, there is 1 comparison and at most 1 swap.

Algorithm: InsertSort(A)

$n = |A|$;
for $i = 1$ *to* $n-1$ **do**
  $j = i - 1$;
  while $j \geq 0$ *and* $A[j] > A[j+1]$ **do**
    $A[j] \rightleftharpoons A[j+1]$;
    $j = j - 1$;
  end
end
Return A;

Insertion Sort

Listing 1: insertsort.c

```c
int main() {
  int A[5] = {5, 3, 1, 2, 4};
  for (int i = 1; i < 5; i++) {
    int j = i - 1;
    while (j >= 0 && A[j] > A[j + 1]) {
      A[j + 1] += A[j];
      A[j] = A[j + 1] - A[j];
      A[j + 1] -= A[j];
      j--;
    }
  }
  return 0;
}
```

Insertion Sort

Listing 2: *-print.c

```
#include<stdio.h>

int main() {
  int A[5] = {5, 3, 1, 2, 4};
  for (int i = 0; i < 5; i++) {
    printf("%d ", A[i]);
  }
  printf("\n");
  return 0;
}
```

Insertion Sort

```
>> gcc insertsort.c -o insertsort
>> ./insertsort
5 3 1 2 4
3 5 1 2 4
1 3 5 2 4
1 2 3 5 4
1 2 3 4 5
```

Insertion Sort: Ideas

Notice that in sorting the first i numbers

- the first $i - 1$ numbers are sorted
- it is enough to find the proper position for the i-th number

Use the binary search to improve the insertion.

Issue: using array again?

Bubble Sort

5	3	1	2	4

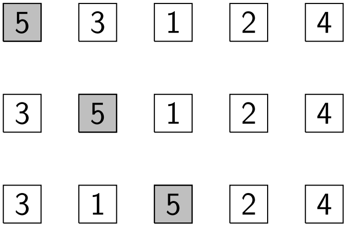
Find the biggest number among the first 5 numbers, and put it at the end.

Bubble Sort

5	3	1	2	4
3	5	1	2	4

Find the biggest number among the first 5 numbers, and put it at the end.

Bubble Sort



Find the biggest number among the first 5 numbers, and put it at the end.

Bubble Sort

Find the biggest number among the first 5 numbers, and put it at the end.

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
3	1	2	5	4

Bubble Sort

Find the biggest number among the first 5 numbers, and put it at the end.

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
3	1	2	5	4
3	1	2	4	5

Bubble Sort

$\boxed{3}$ $\boxed{1}$ $\boxed{2}$ $\boxed{4}$ $\boxed{5}$

Find the biggest number among the first 4 numbers, and put it at the end.

Bubble Sort

$\boxed{3}$ $\boxed{1}$ $\boxed{2}$ $\boxed{4}$ $\boxed{5}$

$\boxed{1}$ $\boxed{3}$ $\boxed{2}$ $\boxed{4}$ $\boxed{5}$

Find the biggest number among the first 4 numbers, and put it at the end.

Bubble Sort

Find the biggest number among the first 4 numbers, and put it at the end.

3	1	2	4	5
1	3	2	4	5
1	2	3	4	5

Bubble Sort

Find the biggest number among the first 4 numbers, and put it at the end.

3	1	2	4	5
1	3	2	4	5
1	2	3	4	5
1	2	3	4	5

Bubble Sort

- Find the biggest number among the first 3 numbers, and put it at the end.

$\boxed{1}$ $\boxed{2}$ $\boxed{3}$ $\boxed{4}$ $\boxed{5}$

Bubble Sort

- Find the biggest number among the first 2 numbers, and put it at the end.

1	2	3	4	5
1	2	3	4	5

Bubble Sort

Algorithm: BubbleSort(A)

$n = |A|$;
for $i = n - 1$ *to* 1 **do**
  $j = 0$;
  while $j < i$ **do**
    if $A[j] > A[j+1]$ **then**
      $A[j] \rightleftharpoons A[j+1]$;
    end
    $j = j + 1$;
  end
end
Return A;

Bubble Sort: $O(n^2)$

- **for**-loop runs in $n - 1$ times.
- inside each loop, there are i comparisons and at most i swaps. ($i = n - 1$ to 1)

Bubble Sort

Listing 3: bubblesort.c

```c
int main() {
  int A[5] = {5, 3, 1, 2, 4};
  for (int i = 4; i > 0; i--) {
    int j = 0;
    while (j < i) {
      if (A[j] > A[j + 1]) {
        A[j + 1] += A[j];
        A[j] = A[j + 1] - A[j];
        A[j + 1] -= A[j];
      }
      j++;
    }
  }
  return 0;
}
```

Bubble Sort

```
>> gcc bubblesort.c -o bubblesort
>> ./bubblesort
5 3 1 2 4
3 1 2 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Bubble Sort: Ideas

Instead of finding the maximum number among the first i numbers,

Bubble Sort: Ideas

Instead of finding the maximum number among the first i numbers,

- finding the maximum and the second maximum at the same time?

Bubble Sort: Ideas

Instead of finding the maximum number among the first i numbers,

- finding the maximum and the second maximum at the same time?
- finding the top-k numbers?

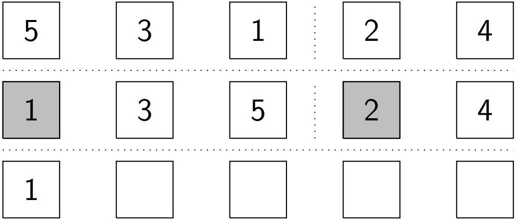
Merge Sort

5	3	1	2	4

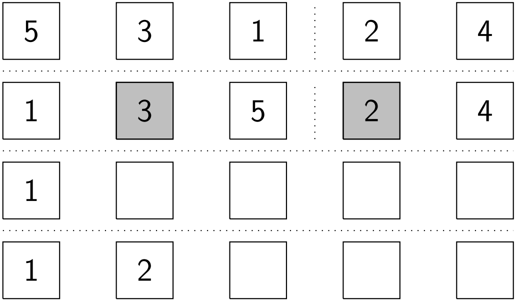
Merge Sort

5	3	1	2	4
1	3	5	2	4

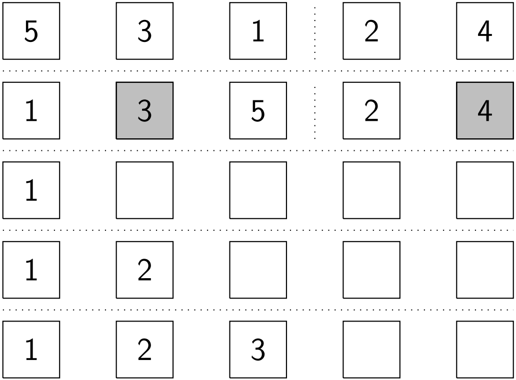
Merge Sort



Merge Sort



Merge Sort



Merge Sort

5	3	1	2	4
1	3	5	2	4
1				
1	2			
1	2	3		
1	2	3	4	

Merge Sort

5	3	1	2	4
1	3	5	2	4
1				
1	2			
1	2	3		
1	2	3	4	
1	2	3	4	5

Merge Sort

Algorithm: Merge(B, C)

$A = [\ ]$;
$i, j = 0$;
while $i < |B|$ *and* $j < |C|$ **do**
    if $B[i] < C[j]$ **then** append $B[i++]$ to A;
    else append $C[j++]$ to A;
end
while $i < |B|$ **do** append $B[i++]$ to A;
while $j < |C|$ **do** append $C[j++]$ to A;
return A;

Merge Sort

Algorithm: MergeSort(A)

if $|A| > 1$ **then**
    $m = \lfloor (|A| - 1)/2 \rfloor$;
    $B = A[0, \ldots, m]$;
    $C = A[m+1, \ldots, |A| - 1]$;
    $B =$ MergeSort(B);
    $C =$ MergeSort(C);
    $A =$ Merge(B, C);
end
return A;

Listing 4: mergesort.c

```c
void merge(int A[], int a1, int b1, int a2, int b2) {
  int B[5];
  int i = a1, j = a2, k = 0;
  while (i <= b1 && j <= b2) {
    if (A[i] < A[j]) B[k++] = A[i++];
    else B[k++] = A[j++];
  }
  while (i <= b1) B[k++] = A[i++];
  while (j <= b2) B[k++] = A[j++];
  for (i = a1, j = 0; i <= b2; i++, j++)
    A[i] = B[j];
}
```

Merge Sort

Listing 5: mergesort.c

```
void mergesort(int A[], int a, int b) {
  int mid = floor((a + b) / 2);
  if (a < b) {
    mergesort(A, a, mid);
    mergesort(A, mid + 1, b);
    merge(A, a, mid, mid + 1, b);
  }
}

int main() {
  int A[5] = {5, 3, 1, 2, 4};
  mergesort(A, 0, 4);
  return 0;
}
```

Merge Sort: $O(n \log n)$

How many operations are performed in MergeSort(A)?

- Let $n = |A|$, and $T(n)$ be the upper bounds of number of operations in MergeSort.
- Assume $n = 2^k$ and $k \in \mathbb{Z}^+$.

Merge Sort: $O(n \log n)$

How many operations are performed in MergeSort(A)?

- Let $n = |A|$, and $T(n)$ be the upper bounds of number of operations in MergeSort.
- Assume $n = 2^k$ and $k \in \mathbb{Z}^+$.
- $T(n) =$ MergeSort(1st half) $+$ MergeSort(2nd half) $+$ Merge(n numbers).

Merge Sort: $O(n \log n)$

How many operations are performed in MergeSort(A)?

- Let $n = |A|$, and $T(n)$ be the upper bounds of number of operations in MergeSort.
- Assume $n = 2^k$ and $k \in \mathbb{Z}^+$.
- $T(n) =$ MergeSort(1st half) $+$ MergeSort(2nd half) $+$ Merge(n numbers).
- $T(n) = 2T(n/2) + n$.

Merge Sort: $O(n \log n)$

$$1: \quad T(n) = 2T(n/2) + n \qquad \longrightarrow \qquad n$$

Merge Sort: $O(n \log n)$

$$1: \quad T(n) = 2T(n/2) + n \qquad \longrightarrow \qquad n$$

$$2: \quad \qquad = 4T(n/4) + 2(n/2) + n \qquad \longrightarrow \qquad 2n$$

Merge Sort: $O(n \log n)$

$$1: \quad T(n) = 2T(n/2) + n \qquad \longrightarrow \quad n$$

$$2: \qquad = 4T(n/4) + 2(n/2) + n \qquad \longrightarrow \quad 2n$$

$$3: \qquad = 8T(n/8) + 4(n/4) + 2(n/2) + n \qquad \longrightarrow \quad 3n$$

Merge Sort: $O(n \log n)$

$$
\begin{array}{lllll}
1: & T(n) = 2T(n/2) + n & & \longrightarrow & n \\
2: & & = 4T(n/4) + 2(n/2) + n & \longrightarrow & 2n \\
3: & & = 8T(n/8) + 4(n/4) + 2(n/2) + n & \longrightarrow & 3n \\
i: & & = 2^i T(n/2^i) + 2^{i-1}(n/2^{i-1}) + \cdots + n & \longrightarrow & i \cdot n
\end{array}
$$

Merge Sort: $O(n \log n)$

$$
\begin{array}{lll}
1: & T(n) = 2T(n/2) + n & \longrightarrow \quad n \\
2: & \quad = 4T(n/4) + 2(n/2) + n & \longrightarrow \quad 2n \\
3: & \quad = 8T(n/8) + 4(n/4) + 2(n/2) + n & \longrightarrow \quad 3n \\
i: & \quad = 2^i T(n/2^i) + 2^{i-1}(n/2^{i-1}) + \cdots + n & \longrightarrow i \cdot n \\
k: & \quad = 2^k T(n/2^k) + 2^{k-1}(n/2^{k-1}) + \cdots + n & \longrightarrow k \cdot n
\end{array}
$$

Merge Sort: $O(n \log n)$

$$
\begin{array}{lll}
1: & T(n) = 2T(n/2) + n & \longrightarrow \quad n \\
2: & \quad = 4T(n/4) + 2(n/2) + n & \longrightarrow \quad 2n \\
3: & \quad = 8T(n/8) + 4(n/4) + 2(n/2) + n & \longrightarrow \quad 3n \\
i: & \quad = 2^i T(n/2^i) + 2^{i-1}(n/2^{i-1}) + \cdots + n & \longrightarrow \quad i \cdot n \\
k: & \quad = 2^k T(n/2^k) + 2^{k-1}(n/2^{k-1}) + \cdots + n & \longrightarrow \quad k \cdot n
\end{array}
$$

Recall that $n = 2^k$.

What about $n \neq 2^k$?

Consider the case when $2^{k-1} < n < 2^k$.

- Let $n' = 2^k$.

What about $n \neq 2^k$?

Consider the case when $2^{k-1} < n < 2^k$.

- Let $n' = 2^k$.
- Add $n' - n = 2^k - n$ dummy numbers: $-\infty$.

0	0	0	5	3	1	2	4

What about $n \neq 2^k$?

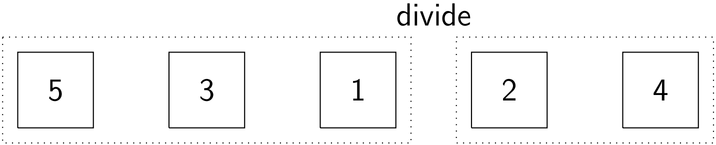
Consider the case when $2^{k-1} < n < 2^k$.

- Let $n' = 2^k$.
- Add $n' - n = 2^k - n$ dummy numbers: $-\infty$.
- Notice that $n' < 2n$. Thus

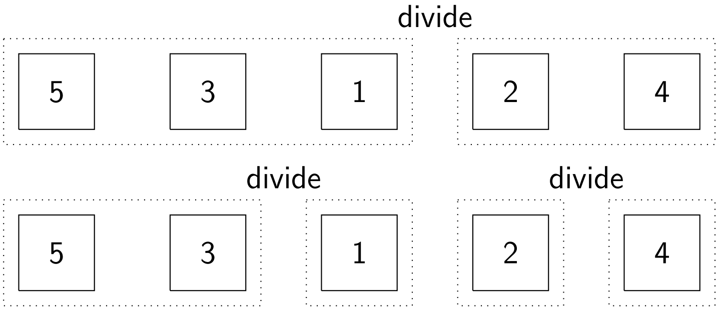
$$T(n) \leq T(n') = O(n' \log n') = O(n \log n).$$

0	0	0	1	2	3	4	5

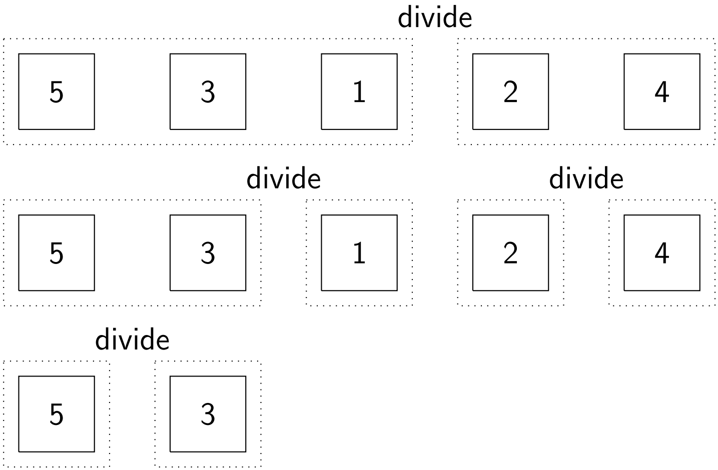
Merge Sort: Details of The Recursions



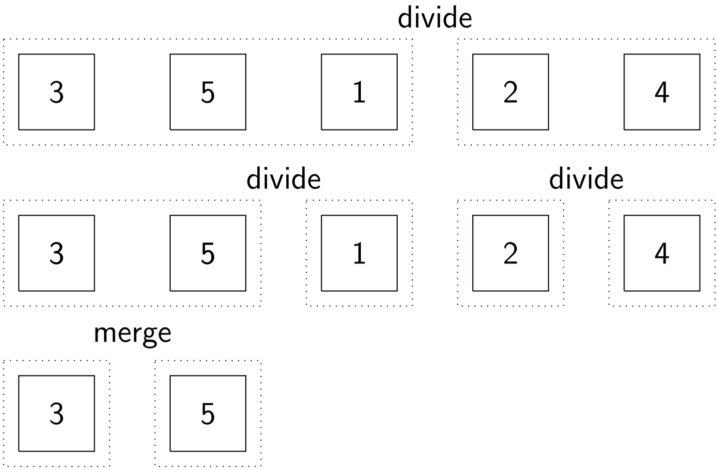
Merge Sort: Details of The Recursions



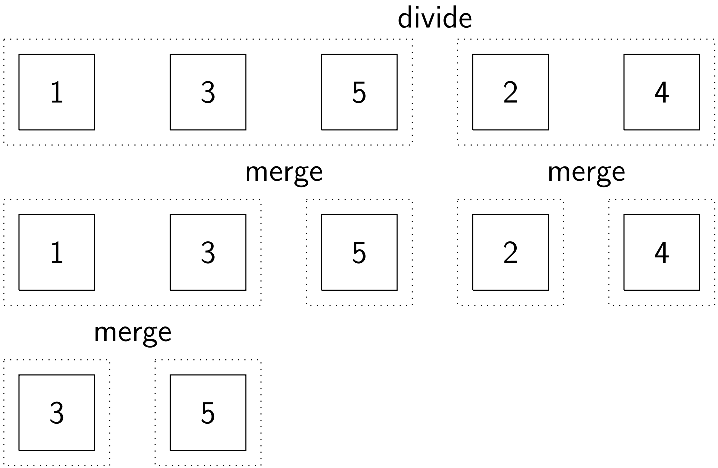
Merge Sort: Details of The Recursions



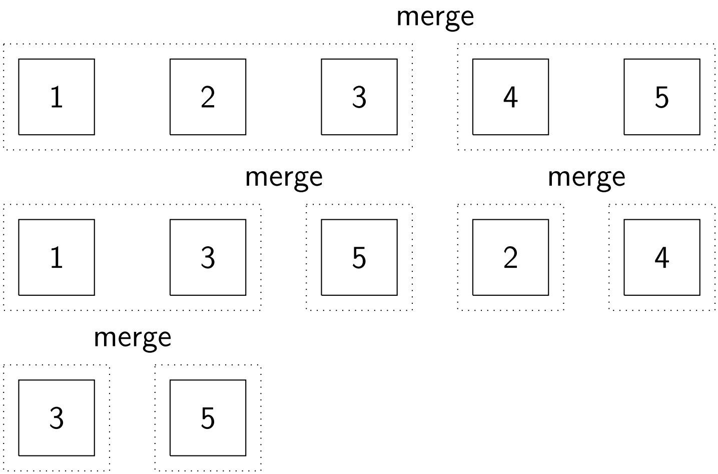
Merge Sort: Details of The Recursions



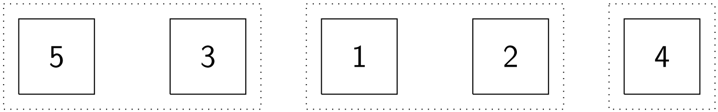
Merge Sort: Details of The Recursions



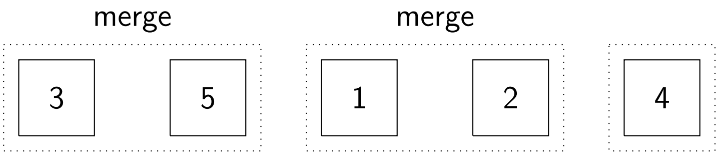
Merge Sort: Details of The Recursions



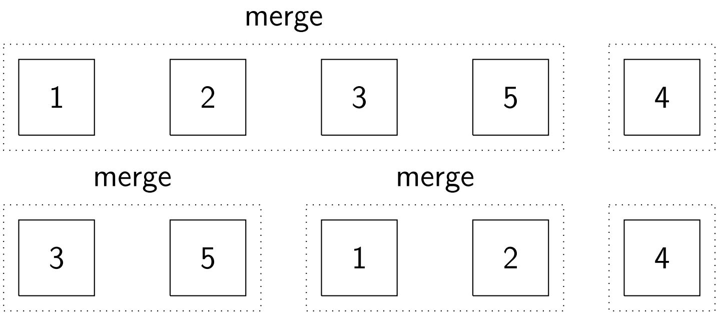
Merge Sort: Avoid Recursions



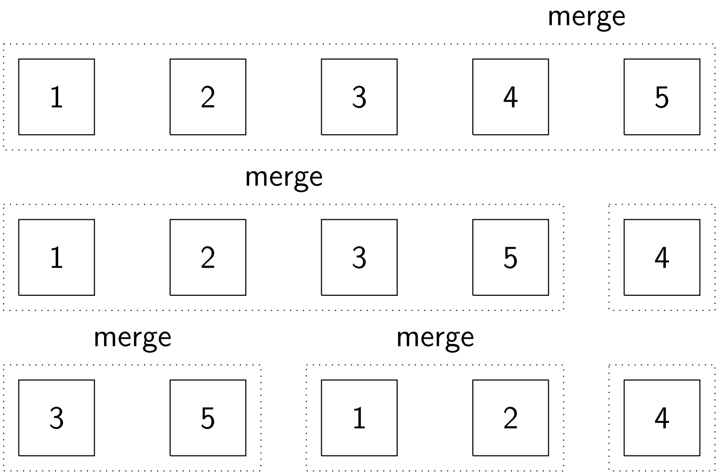
Merge Sort: Avoid Recursions



Merge Sort: Avoid Recursions



Merge Sort: Avoid Recursions



Merge Sort: Avoid Recursions

```
void mergesort(int A[], int a, int b) {
  int n = b - a + 1;
  int m = (int) floor(log2(n));
  for (int p = 0; p <= m; p++) {
    int k = (int) pow(2, p), r = 2 * k;
    for (int a1 = a; a1 <= (n / r) * r; a1 += r) {
      int b1 = a1 + k - 1;
      if (b1 < b) {
        int a2 = b1 + 1, b2 = a2 + k - 1;
        if (b2 > b) b2 = b;
        merge(A, a1, b1, a2, b2);
      }
    }
  }
}
```

Quick Sort

Pivot: a (arbitrarily selected) number in the given sequence

- the first part: numbers $\leq$ *pivot*
- the second part: numbers $>$ *pivot*

Sort the two parts in recursions.

Quick Sort

Algorithm: QuickSort(A)

if $|A| > 1$ **then**

  pivot $=$ a number in A;

  // Divide the others into B and C, where

  $B =$ numbers $\leq$ pivot ;

  $C =$ numbers $>$ pivot;

  $B =$ QuickSort(B);

  $C =$ QuickSort(C);

  $A = B +$ pivot $+ C$;

end

return A;

Quick Sort: Worst $O(n^2)$

5	4	3	2	1

Quick Sort: Worst $O(n^2)$

5	4	3	2	1
1	5	4	3	2

Quick Sort: Worst $O(n^2)$

5	4	3	2	1
1	5	4	3	2
1	2	5	4	3

Quick Sort: Worst $O(n^2)$

5	4	3	2	1
1	5	4	3	2
1	2	5	4	3
1	2	3	5	4

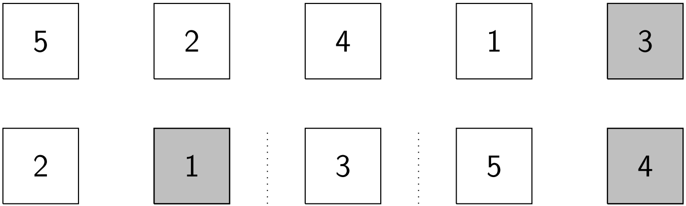
Quick Sort: Worst $O(n^2)$

5	4	3	2	1
1	5	4	3	2
1	2	5	4	3
1	2	3	5	4
1	2	3	4	5

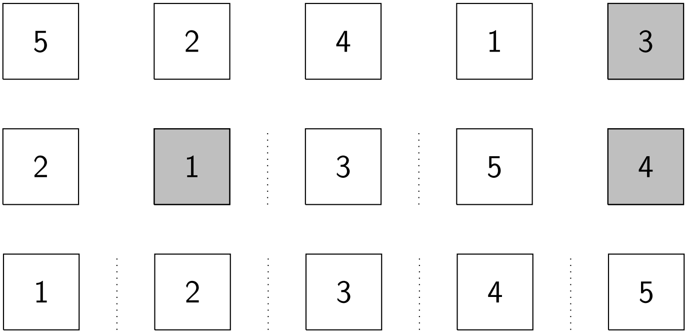
Quick Sort: Best $O(n \log n)$

5	2	4	1	3

Quick Sort: Best $O(n \log n)$



Quick Sort: Best $O(n \log n)$



Quick Sort: More Pivots

Why only one pivot?

Quick sort with two-pivots

- select two numbers from A: p_1 and p_2
- divide A into three parts
  - numbers $\leq p_1$
  - numbers $> p_1$ and $\leq p_2$
  - numbers $> p_2$
- recursively sort each part

Lower Bound for Comparison Sort

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Lower Bound for Comparison Sort

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Can we sort without comparisons?

Counting Sort

Sort n integers which range from 0 to $k-1$, where k is a constant.

Input size: n, as each integer can be represented by constant number of bits.

Counting Sort: $O(n)$

1	2	2	0	1

Counting Sort: $O(n)$

1	2	2	0	1

numbers of $0's$: 1

numbers of $1's$: 2

numbers of $2's$: 2

Counting Sort: $O(n)$

1	2	2	0	1

numbers of $0's$: 1

numbers of $1's$: 2

numbers of $2's$: 2

0	1	1	2	2

Divide and Conquer

Merge Sort

$m = \lfloor (|A| - 1)/2 \rfloor;$

$B = A[0, ..., m];$

$C = A[m + 1, ..., |A| - 1];$

$B =$MergeSort$(B);$

$C =$MergeSort$(C);$

$A =$Merge$(B,\ C);$

Merge Sort

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ & \uparrow & & \\ & m & & \end{array} \qquad \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ & & \uparrow & & \\ & & m & & \end{array}$$

$m = \lfloor (|A| - 1)/2 \rfloor;$

$B = A[0, ..., m];$

$C = A[m + 1, ..., |A| - 1];$

$B =$MergeSort$(B);$

$C =$MergeSort$(C);$

$A =$Merge$(B,\ C);$

Merge Sort

$$
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
 & \uparrow & & \\
 & m & &
\end{array}
\qquad
\begin{array}{ccccc}
0 & 1 & 2 & 3 & 4 \\
 & & \uparrow & & \\
 & & m & &
\end{array}
$$

$m = \lfloor (|A| - 1)/2 \rfloor;$

$B = A[0, ..., m];$

$$
\begin{array}{ccccc}
0 & 1 & 2 & \quad 3 & 4 \\
 & & \uparrow & & \\
 & & m & &
\end{array}
$$

$C = A[m + 1, ..., |A| - 1];$

$B =$MergeSort$(B);$

$C =$MergeSort$(C);$

$A =$Merge$(B,\ C);$

Merge Sort

$m = \lfloor (|A| - 1)/2 \rfloor;$

$B = A[0, ..., m];$

$C = A[m+1, ..., |A| - 1];$

$B =$MergeSort$(B);$

$C =$MergeSort$(C);$

$A =$Merge$(B,\ C);$

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ & \uparrow & & \\ & m & & \end{array} \qquad \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ & & \uparrow & & \\ & & m & & \end{array}$$

$$\begin{array}{ccc} 0 & 1 & 2 \\ & & \uparrow \\ & & m \end{array} \qquad \begin{array}{cc} 3 & 4 \end{array}$$

$$\boxed{T(n) = 2T(n/2) + n}$$

Divide and Conquer

1. **Divide** the problem instance into two/several smaller instances of the same problem.
2. **Conquer** the smaller problems.
3. **Combine** the results of the smaller problems to get the result of the original (large) instance.

Divide and Conquer

1. **Divide** the problem instance into two/several smaller instances of the same problem.
2. **Conquer** the smaller problems.
3. **Combine** the results of the smaller problems to get the result of the original (large) instance.

Recall that in MergeSort,

1. **Divide** the numbers into two subsets.
2. **Conquer** the sorting problem on each of the subsets.
3. **Combine** the results, by Merge.

THANK YOU

中国科学院深圳先进技术研究院
SHENZHEN INSTITUTES OF ADVANCED TECHNOLOGY
CHINESE ACADEMY OF SCIENCES