

## Week 02: Sorting

FanFly

March 15, 2020

# Time Complexity of a Problem

The time complexity  $T(n)$  of a problem  $P$  is the time complexity of the “best” algorithm that solves  $P$ .

# Time Complexity of a Problem

The time complexity  $T(n)$  of a problem  $P$  is the time complexity of the “best” algorithm that solves  $P$ .

- We have  $T(n) = O(f(n))$  if there is an algorithm that solves  $P$  in  $O(f(n))$  time.

# Time Complexity of a Problem

The time complexity  $T(n)$  of a problem  $P$  is the time complexity of the “best” algorithm that solves  $P$ .

- We have  $T(n) = O(f(n))$  if there is an algorithm that solves  $P$  in  $O(f(n))$  time.
- We have  $T(n) = \Omega(f(n))$  if any algorithm that solves  $P$  needs  $\Omega(f(n))$  time.

# The Primality Test Problem

## The Primality Test Problem

- Input: An  $n$ -bit positive integer  $m \geq 2$ .
- Output: True if  $m$  is prime, false if  $m$  is composite.

# The Primality Test Problem

## The Primality Test Problem

- Input: An  $n$ -bit positive integer  $m \geq 2$ .
  - Output: True if  $m$  is prime, false if  $m$  is composite.
- 
- Last week, we found an  $O(2^{n/2})$ -time algorithm that solves the primality test problem.

# The Primality Test Problem

## The Primality Test Problem

- Input: An  $n$ -bit positive integer  $m \geq 2$ .
  - Output: True if  $m$  is prime, false if  $m$  is composite.
- 
- Last week, we found an  $O(2^{n/2})$ -time algorithm that solves the primality test problem.
  - An  $O(n^{12}(\log_2 n)^\epsilon)$ -time algorithm, called the AKS primality test, was proposed in 2002. ( $\epsilon$  is a positive number.)

# The Primality Test Problem

## The Primality Test Problem

- Input: An  $n$ -bit positive integer  $m \geq 2$ .
  - Output: True if  $m$  is prime, false if  $m$  is composite.
- 
- Last week, we found an  $O(2^{n/2})$ -time algorithm that solves the primality test problem.
  - An  $O(n^{12}(\log_2 n)^\epsilon)$ -time algorithm, called the AKS primality test, was proposed in 2002. ( $\epsilon$  is a positive number.)
  - In 2005, it is improved to run in  $O(n^6(\log_2 n)^\epsilon)$  time.



# The Primality Test Problem

## The Primality Test Problem

- Input: An  $n$ -bit positive integer  $m \geq 2$ .
  - Output: True if  $m$  is prime, false if  $m$  is composite.
- 
- Last week, we found an  $O(2^{n/2})$ -time algorithm that solves the primality test problem.
  - An  $O(n^{12}(\log_2 n)^\epsilon)$ -time algorithm, called the AKS primality test, was proposed in 2002. ( $\epsilon$  is a positive number.)
  - In 2005, it is improved to run in  $O(n^6(\log_2 n)^\epsilon)$  time.
  - Thus, now we know that the primality test problem can be solved in  $O(n^6(\log_2 n)^\epsilon)$  time, but no one knows if there is a faster algorithm than the ones above.

# The Champion Problem

## The Champion Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: An index  $i$  such that  $A[i] \geq A[j]$  for any index  $j$ .

# The Champion Problem

## The Champion Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: An index  $i$  such that  $A[i] \geq A[j]$  for any index  $j$ .

There is a  $O(n)$ -time algorithm that solves the problem as follows.

# The Champion Problem

## The Champion Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: An index  $i$  such that  $A[i] \geq A[j]$  for any index  $j$ .

There is a  $O(n)$ -time algorithm that solves the problem as follows.

```
def index_max(A):  
    i = 0  
    for j in range(1, len(A)):  
        if A[j] > A[i]:  
            i = j  
    return i
```

# The Champion Problem

## The Champion Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: An index  $i$  such that  $A[i] \geq A[j]$  for any index  $j$ .

There is a  $O(n)$ -time algorithm that solves the problem as follows.

```
def index_max(A):  
    i = 0  
    for j in range(1, len(A)):  
        if A[j] > A[i]:  
            i = j  
    return i
```

Also, note that we need at least  $n - 1$  comparisons to solve the problem, implying that the time complexity of the problem is  $\Omega(n)$ .

# The Champion Problem

## The Champion Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: An index  $i$  such that  $A[i] \geq A[j]$  for any index  $j$ .

There is a  $O(n)$ -time algorithm that solves the problem as follows.

```
def index_max(A):  
    i = 0  
    for j in range(1, len(A)):  
        if A[j] > A[i]:  
            i = j  
    return i
```

Also, note that we need at least  $n - 1$  comparisons to solve the problem, implying that the time complexity of the problem is  $\Omega(n)$ .

Thus, we have found an **optimal** algorithm for the champion problem.

# The Sorting Problem

## The Sorting Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: A non-decreasing array that is reordered from  $A$ .

# The Sorting Problem

## The Sorting Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: A non-decreasing array that is reordered from  $A$ .

We have known that there is an algorithm, called merge sort, that can solve the sorting problem.



# Merging Sorted Arrays

First, we propose an algorithm merging two sorted arrays  $P$  and  $Q$  into a sorted array  $A$ .

# Merging Sorted Arrays

First, we propose an algorithm merging two sorted arrays  $P$  and  $Q$  into a sorted array  $A$ .

```
def merge(P, Q, i, j):  
    if j == len(Q):  
        return P  
    elif i == len(P):  
        return Q  
    else:  
        if P[i] <= Q[j]:  
            return [P[i]] + merge(P, Q, i + 1, j)  
        else:  
            return [Q[j]] + merge(P, Q, i, j + 1)
```



# Merging Sorted Arrays

First, we propose an algorithm merging two sorted arrays  $P$  and  $Q$  into a sorted array  $A$ .

```
def merge(P, Q, i, j):  
    if j == len(Q):  
        return P  
    elif i == len(P):  
        return Q  
    else:  
        if P[i] <= Q[j]:  
            return [P[i]] + merge(P, Q, i + 1, j)  
        else:  
            return [Q[j]] + merge(P, Q, i, j + 1)
```



It can be shown that the algorithm runs in  $\Theta(n)$  time.

# Merge Sort

Then we can sort the array by repeating merging its subarrays.

```
def merge_sort(A, l, r):  
    if r - l <= 1:  
        return a[l:r]  
    else:  
        m = (l + r) // 2  
        P = merge_sort(A, l, m)  
        Q = merge_sort(A, m, r)  
        return merge(P, Q, 0, 0)
```

# Merge Sort

Then we can sort the array by repeating merging its subarrays.

```
def merge_sort(A, l, r):  
    if r - l <= 1:  
        return a[l:r]  
    else:  
        m = (l + r) // 2  
        P = merge_sort(A, l, m)  
        Q = merge_sort(A, m, r)  
        return merge(P, Q, 0, 0)
```

The time complexity of merge sort is

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n), & \text{otherwise.} \end{cases}$$

# Merge Sort

Then we can sort the array by repeating merging its subarrays.

```
def merge_sort(A, l, r):  
    if r - l <= 1:  
        return a[l:r]  
    else:  
        m = (l + r) // 2  
        P = merge_sort(A, l, m)  
        Q = merge_sort(A, m, r)  
        return merge(P, Q, 0, 0)
```

The time complexity of merge sort is

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n), & \text{otherwise.} \end{cases}$$

What is the exact time complexity of merge sort?

# Master Theorem

## Theorem (Master Theorem)

# Master Theorem

## Theorem (Master Theorem)

Let  $T(n)$  be a positive function satisfying the following recurrence relation.

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ aT(n/b) + M(n), & \text{otherwise.} \end{cases}$$

Let  $c = \log_b a$ .

- If  $M(n) = O(n^k)$  with  $k < c$ , then  $T(n) = \Theta(n^c)$ .



# Master Theorem

## Theorem (Master Theorem)

Let  $T(n)$  be a positive function satisfying the following recurrence relation.

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ aT(n/b) + M(n), & \text{otherwise.} \end{cases}$$

Let  $c = \log_b a$ .

- If  $M(n) = O(n^k)$  with  $k < c$ , then  $T(n) = \Theta(n^c)$ .
- If  $M(n) = \Theta(n^k)$  with  $k = c$ , then  $T(n) = \Theta(n^c \log_2 n)$ .

# Master Theorem

## Theorem (Master Theorem)

Let  $T(n)$  be a positive function satisfying the following recurrence relation.

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ aT(n/b) + M(n), & \text{otherwise.} \end{cases}$$

Let  $c = \log_b a$ .

- If  $M(n) = O(n^k)$  with  $k < c$ , then  $T(n) = \Theta(n^c)$ .
- If  $M(n) = \Theta(n^k)$  with  $k = c$ , then  $T(n) = \Theta(n^c \log_2 n)$ .
- If  $M(n) = \Omega(n^k)$  with  $k > c$ , then  $T(n) = \Theta(M(n))$ .

# Master Theorem

## Theorem (Master Theorem)

Let  $T(n)$  be a positive function satisfying the following recurrence relation.

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ aT(n/b) + M(n), & \text{otherwise.} \end{cases}$$

Let  $c = \log_b a$ .

- If  $M(n) = O(n^k)$  with  $k < c$ , then  $T(n) = \Theta(n^c)$ .
- If  $M(n) = \Theta(n^k)$  with  $k = c$ , then  $T(n) = \Theta(n^c \log_2 n)$ .
- If  $M(n) = \Omega(n^k)$  with  $k > c$ , then  $T(n) = \Theta(M(n))$ .

Thus, the time complexity of merge sort is  $T(n) = \Theta(n \log_2 n)$  since

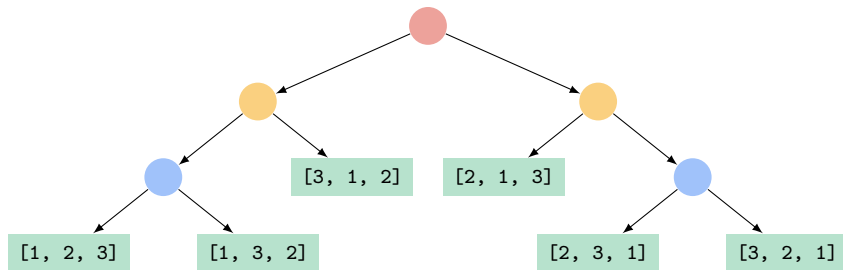
$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n), & \text{otherwise.} \end{cases}$$

# Lower Bound of Comparison-Based Sorting

Any comparison-based algorithm can be seen as a binary tree.

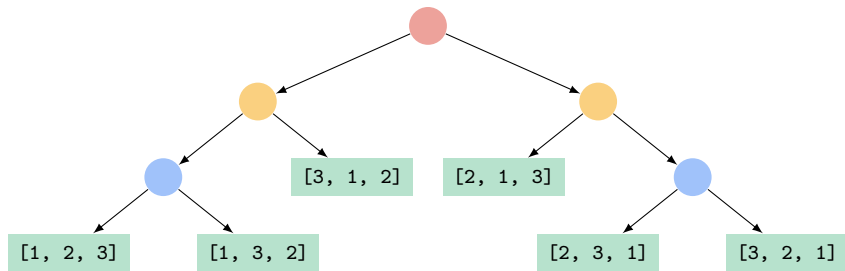
# Lower Bound of Comparison-Based Sorting

Any comparison-based algorithm can be seen as a binary tree.



# Lower Bound of Comparison-Based Sorting

Any comparison-based algorithm can be seen as a binary tree.



Since there are  $n!$  leaves, we know that the height of the binary tree is at least

$$h = \log_2(n!),$$

implying that any comparison-based algorithm runs in  $\Omega(\log_2(n!))$  time.

# Lower Bound of Comparison-Based Sorting (cont.)

Note that

$$\begin{aligned}\log_2(n!) &= \log_2(n \times (n-1) \times \cdots \times 2 \times 1) \\ &\geq \log_2\left(n \times (n-1) \times \cdots \times \left\lceil \frac{n+1}{2} \right\rceil\right) \\ &\geq \frac{n}{2} \log_2\left(\frac{n}{2}\right) \\ &= \frac{n}{2}(\log_2 n - 1) \\ &= \Omega(n \log_2 n).\end{aligned}$$

# Lower Bound of Comparison-Based Sorting (cont.)

Note that

$$\begin{aligned}\log_2(n!) &= \log_2(n \times (n-1) \times \cdots \times 2 \times 1) \\ &\geq \log_2\left(n \times (n-1) \times \cdots \times \left\lceil \frac{n+1}{2} \right\rceil\right) \\ &\geq \frac{n}{2} \log_2\left(\frac{n}{2}\right) \\ &= \frac{n}{2}(\log_2 n - 1) \\ &= \Omega(n \log_2 n).\end{aligned}$$

Thus, any comparison-based algorithm runs in  $\Omega(n \log_2 n)$  time.



# Conclusion

## The Sorting Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: A non-decreasing array that is reordered from  $A$ .

For any computational problem, the final goal is to find an optimal algorithm that solves the problem.

# Conclusion

## The Sorting Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: A non-decreasing array that is reordered from  $A$ .

For any computational problem, the final goal is to find an optimal algorithm that solves the problem.

- The merge sort algorithm runs in  $O(n \log_2 n)$  time.

# Conclusion

## The Sorting Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: A non-decreasing array that is reordered from  $A$ .

For any computational problem, the final goal is to find an optimal algorithm that solves the problem.

- The merge sort algorithm runs in  $O(n \log_2 n)$  time.
- Any comparison-based algorithm that solves the sorting problem must run in  $\Omega(n \log_2 n)$  time.

# Conclusion

## The Sorting Problem

- Input: An array  $A$  of  $n$  numbers (indexed from 0 to  $n - 1$ ).
- Output: A non-decreasing array that is reordered from  $A$ .

For any computational problem, the final goal is to find an optimal algorithm that solves the problem.

- The merge sort algorithm runs in  $O(n \log_2 n)$  time.
- Any comparison-based algorithm that solves the sorting problem must run in  $\Omega(n \log_2 n)$  time.
- Thus, the merge sort algorithm is an optimal comparison-based algorithm that solves the sorting problem.

# Exercise #1

We have learned the binary search algorithm, which can search a value in a sorted array.

# Exercise #1

We have learned the binary search algorithm, which can search a value in a sorted array.

```
def binary_search(A, val, l, r):  
    if val <= A[l]:  
        return l  
    elif val > A[r - 1]:  
        return r  
    else:  
        m = (l + r) // 2  
        if val <= A[m]:  
            return binary_search(A, val, l, m)  
        else:  
            return binary_search(A, val, m + 1, r)
```

## Exercise #1 (cont.)

It can be shown that the time complexity of binary search is

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ T(n/2) + O(1), & \text{otherwise.} \end{cases}$$

## Exercise #1 (cont.)

It can be shown that the time complexity of binary search is

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ T(n/2) + O(1), & \text{otherwise.} \end{cases}$$

Please find the exact time complexity of binary search using the master theorem.

### Theorem (Master Theorem)

Let  $T(n)$  be a positive function satisfying the following recurrence relation.

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ aT(n/b) + M(n), & \text{otherwise.} \end{cases}$$

Let  $c = \log_b a$ .

- If  $M(n) = O(n^k)$  with  $k < c$ , then  $T(n) = \Theta(n^c)$ .
- If  $M(n) = \Theta(n^k)$  with  $k = c$ , then  $T(n) = \Theta(n^c \log_2 n)$ .
- If  $M(n) = \Omega(n^k)$  with  $k > c$ , then  $T(n) = \Theta(M(n))$ .



## Exercise #1 (cont.)

### Solution

Let  $c = \log_2 1 = 0$  and  $M(n) = O(1)$ . Since  $M(n) = \Theta(n^c)$ , we have

$$T(n) = \Theta(n^c \log_2 n) = \Theta(\log_2 n),$$

which means that binary search runs in logarithmic time.

## Exercise #2

Suppose that  $f$  and  $g$  are functions such that

$$f(n) = \Theta(\log_2 n) \quad \text{and} \quad g(n) = \Theta(\log_e n),$$

and we know the fact that  $2.718 < e < 2.719$ .

## Exercise #2

Suppose that  $f$  and  $g$  are functions such that

$$f(n) = \Theta(\log_2 n) \quad \text{and} \quad g(n) = \Theta(\log_e n),$$

and we know the fact that  $2.718 < e < 2.719$ .

Which of the following is true?

## Exercise #2

Suppose that  $f$  and  $g$  are functions such that

$$f(n) = \Theta(\log_2 n) \quad \text{and} \quad g(n) = \Theta(\log_e n),$$

and we know the fact that  $2.718 < e < 2.719$ .

Which of the following is true?

- $f(n) = O(g(n))$  and  $f(n) \neq \Omega(g(n))$ .

## Exercise #2

Suppose that  $f$  and  $g$  are functions such that

$$f(n) = \Theta(\log_2 n) \quad \text{and} \quad g(n) = \Theta(\log_e n),$$

and we know the fact that  $2.718 < e < 2.719$ .

Which of the following is true?

- $f(n) = O(g(n))$  and  $f(n) \neq \Omega(g(n))$ .
- $f(n) = \Theta(g(n))$ .

## Exercise #2

Suppose that  $f$  and  $g$  are functions such that

$$f(n) = \Theta(\log_2 n) \quad \text{and} \quad g(n) = \Theta(\log_e n),$$

and we know the fact that  $2.718 < e < 2.719$ .

Which of the following is true?

- $f(n) = O(g(n))$  and  $f(n) \neq \Omega(g(n))$ .
- $f(n) = \Theta(g(n))$ .
- $f(n) = \Omega(g(n))$  and  $f(n) \neq O(g(n))$ .

## Exercise #2

Suppose that  $f$  and  $g$  are functions such that

$$f(n) = \Theta(\log_2 n) \quad \text{and} \quad g(n) = \Theta(\log_e n),$$

and we know the fact that  $2.718 < e < 2.719$ .

Which of the following is true?

- $f(n) = O(g(n))$  and  $f(n) \neq \Omega(g(n))$ .
- $f(n) = \Theta(g(n))$ .
- $f(n) = \Omega(g(n))$  and  $f(n) \neq O(g(n))$ .

### Solution

We have  $f(n) = \Theta(g(n))$  since

$$\frac{\log_e n}{\log_2 n} = \log_e 2.$$

(From now on we'll use  $O(\log n)$  instead of  $O(\log_k n)$  to represent logarithm.)

## Exercise #3

Please find the time complexity of the following algorithm using big-O notation.



## Exercise #3

Please find the time complexity of the following algorithm using big-O notation.

```
def magic_sort(A):  
    P = list(A)  
    Q = []  
    while P:  
        Q.append(P.pop(index_min(P)))  
    return Q
```

## Exercise #3

Please find the time complexity of the following algorithm using big-O notation.

```
def magic_sort(A):  
    P = list(A)  
    Q = []  
    while P:  
        Q.append(P.pop(index_min(P)))  
    return Q
```

We have the following assumption, where  $n$  is the length of  $1s$ .

## Exercise #3

Please find the time complexity of the following algorithm using big-O notation.

```
def magic_sort(A):  
    P = list(A)  
    Q = []  
    while P:  
        Q.append(P.pop(index_min(P)))  
    return Q
```

We have the following assumption, where  $n$  is the length of  $ls$ .

- `index_min(ls)` returns the index of the smallest item in  $ls$  in  $O(n)$  time.

## Exercise #3

Please find the time complexity of the following algorithm using big-O notation.

```
def magic_sort(A):  
    P = list(A)  
    Q = []  
    while P:  
        Q.append(P.pop(index_min(P)))  
    return Q
```

We have the following assumption, where  $n$  is the length of `ls`.

- `index_min(ls)` returns the index of the smallest item in `ls` in  $O(n)$  time.
- `ls.append(val)` adds `val` to the end of `ls` in  $O(1)$  time.

## Exercise #3

Please find the time complexity of the following algorithm using big-O notation.

```
def magic_sort(A):  
    P = list(A)  
    Q = []  
    while P:  
        Q.append(P.pop(index_min(P)))  
    return Q
```

We have the following assumption, where  $n$  is the length of `ls`.

- `index_min(ls)` returns the index of the smallest item in `ls` in  $O(n)$  time.
- `ls.append(val)` adds `val` to the end of `ls` in  $O(1)$  time.
- `ls.pop(i)` removes the item with index `i` in `ls` in  $O(n)$  time.

## Exercise #3

Please find the time complexity of the following algorithm using big-O notation.

```
def magic_sort(A):  
    P = list(A)  
    Q = []  
    while P:  
        Q.append(P.pop(index_min(P)))  
    return Q
```

We have the following assumption, where  $n$  is the length of `ls`.

- `index_min(ls)` returns the index of the smallest item in `ls` in  $O(n)$  time.
- `ls.append(val)` adds `val` to the end of `ls` in  $O(1)$  time.
- `ls.pop(i)` removes the item with index `i` in `ls` in  $O(n)$  time.

### Solution

The magic sort runs in  $O(n^2)$  time.