MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2004
Algorithms and Data Structures

Ian Wern Han Lim
lim.wern.han@monash.edu

Referencing materials by
Nathan Companez, Aamir Cheema, Arun Konagurthu and Lloyd Allison

GROUP OF EIGHT AUSTRALIA

Ready?

Agenda

- Complexity Analysis
  - Big-O
  - Recurrence relation

Agenda

- Complexity Analysis
  - Big-O
  - Recurrence relation

Agenda

- Complexity Analysis
  - Big-O
  - Recurrence relation

Covered in Tutorial 02 using tutorial questions as case study

Let us begin…

Recap

- You have done time complexity last time

Complexity Analysis
Recap

- You have done time complexity last time
- You know what Big-O is

Complexity Analysis
Recap

- You have done time complexity last time
- You know what Big-O is

What is the complexity of an algorithm in Big-O notation that runs in 30N log (N²) + 10 log N + 8N?

A. **O(N log N)**
B. **O(N log (N²))**
C. **O(N log (N²) + N + log N)**
D. **Option D because**

What is your answer?

- You have done time complexity last time
- You know what Big-O is

What is the complexity of an algorithm in Big-O notation that runs in $30N \log (N^2) + 10 \log N + 8N$?

A. $O(N \log N)$
B. $O(N \log (N^2))$
C. $O(N \log (N^2) + N + \log N)$
D. Option D because

- You have done time complexity last time
- You know what Big-O is

What is the complexity of an algorithm in Big-O notation that runs in $30N \log (N^2) + 10 \log N + 8N$?

A. $O(N \log N)$
B. $O(N \log (N^2))$
C. $O(N \log (N^2) + N + \log N)$
D. Option D because

Complexity Analysis
Recap

- You have done time complexity last time
- You know what Big-O is

What is the complexity of an algorithm in Big-O notation that runs in $30N \log (N^2) + 10 \log N + 8N$?

A. $\boxed{O(N \log N)}$

B. $O(N \log (N^2))$

C. $O(N \log (N^2) + N + \log N)$

D. Option D because

Complexity Analysis

Recap

- You have done time complexity last time
- You know what Big-O is

What is the complexity of an algorithm in Big-O notation that runs in $30N \log (N^2) + 10 \log N + 8N$?

A. $O(N \log N)$

B. $O(N \log (N^2))$

C. $O(N \log (N^2) + N + \log N)$

D. Option D because

Log N^2 = 2 log N

- You have done time complexity last time
- You know what Big-O is

- If I have a list, and I want to sort it with bubble sort…

Complexity Analysis

Recap

- You have done time complexity last time
- You know what Big-O is

- If I have a list, and I want to sort it with bubble sort…
  - Best case?
  - Worst case?

Complexity Analysis
Recap

- You have done time complexity last time
- You know what Big-O is

- If I have a list, and I want to sort it with bubble sort…
  - Best case? O(1) when list is empty
  - Worst case? O(N^2) when list is sorted in reverse order

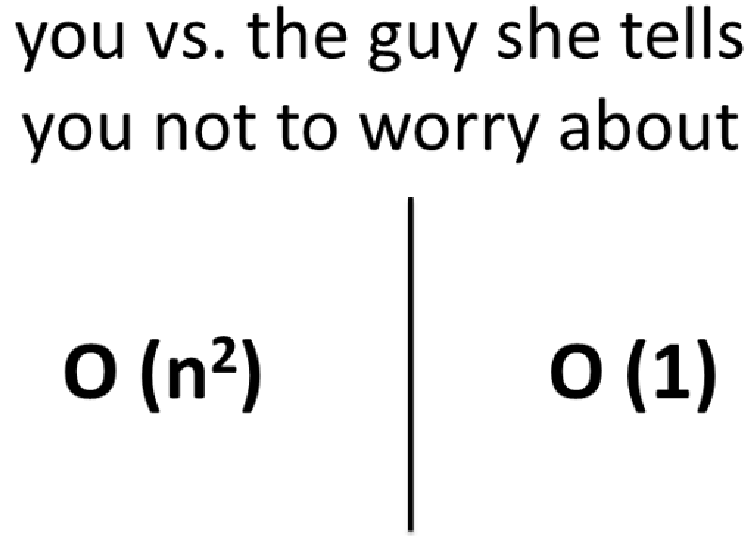
- You have done time complexity last time
- You know what Big-O is

- If I have a list, and I want to sort it with bubble sort…
  - Best case? O(1) when list is empty
  - Worst case? O(N^2) when list is sorted in reverse order

- You have done time complexity last time
- You know what Big-O is

- If I have a list, and I want to sort it with bubble sort…
  – Best case? O(N) when list is sorted and we can terminate earlier
  – Worst case? O(N^2) when list is sorted in reverse order

- You have done time complexity last time
- You know what Big-O is



- You have done time complexity last time
- You know what Big-O is

- So what’s new?

Complexity Analysis
Recap

- You have done time complexity last time
- You know what Big-O is

- So what’s new?
  – Space complexity
  – Time complexity for recursion

- You have done time complexity last time
- You know what Big-O is

- So what's new?
  - Space complexity
  - Time complexity for recursion
    - By solving recurrence relation

- You have done time complexity last time
- You know what Big-O is

- So what's new?
  - Space complexity
    - We'll leave this for Lecture 02 when we go through some sorting algorithms
  - Time complexity for recursion
    - By solving recurrence relation

Questions?

Complexity Analysis
Recurrence relation

- This is asked a lot
  - Final exam

- This is asked a lot
  - Final exam

- Helps you figure out the complexity of recursive functions

- Look at the algorithm written in Python

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Look at the algorithm written in Python

- Note: Can you explain what this function do? … and reason out the complexity?
  - Using FIT1008 knowledge

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis

Recurrence relation

- Look at the algorithm written in Python
- What is the recurrence relation?

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis
Recurrence relation

- Look at the algorithm written in Python
- What is the recurrence relation?
  - Base case
  - Recursive case

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis

Recurrence relation

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
  - Recursive case

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

MONASH University

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
    T(0) = a
  - Recursive case

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis
Recurrence relation

MONASH University

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
    T(0) = a
    T(1) = b
  - Recursive case

```
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
    T(0) = a
    T(1) = b
    Constant operating cost…
  - Recursive case

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis

Recurrence relation

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
    T(0) = a
    T(1) = b
    Constant operating cost…
  - Recursive case
    T(N) = T(N-1)

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis
Recurrence relation

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
    T(0) = a
    T(1) = b
    Constant operating cost…
  - Recursive case
    T(N) = T(N-1) * X?

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis
Recurrence relation

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
    T(0) = a
    T(1) = b
    Constant operating cost…
  - Recursive case
    T(N) = T(N-1) + c
    Cause of constant operating cost in multiplying…

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis

Recurrence relation

MONASH University

- Look at the algorithm written in Python
- What is the recurrence relation? T(N)
  - Base case
    T(0) = a
    T(1) = b
    Constant operating cost…
  - Recursive case (general case)
    T(N) = T(N-1) + c
    Cause of constant operating cost in multiplying…

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis
Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N-1) + c

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N-1) + c

```
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Now solve it for the complexity

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N-1) + c

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Now solve it for the complexity
  – T(N) = T(N-1) + c
  – so T(N-1) = T(N-2) + c

Complexity Analysis

Recurrence relation

- $T(0) = a$
- $T(1) = b$
- $T(N) = T(N-1) + c$

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Now solve it for the complexity
  - $T(N) = T(N-1) + c$
  - so $T(N-1) = T(N-2) + c$
  - $T(N) = T(N-2) + c + c$
  - $T(N) = T(N-3) + c + c + c$

MONASH University

- $T(0) = a$
- $T(1) = b$
- $T(N) = T(N-1) + c$

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Now solve it for the complexity
  – $T(N) = T(N-1) + c$
  – so $T(N-1) = T(N-2) + c$
  – $T(N) = T(N-2) + c + c$
  – $T(N) = T(N-3) + c + c + c$

Known as telescoping

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N-1) + c

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Now solve it for the complexity
  - T(N) = T(N-1) + c
  - so T(N-1) = T(N-2) + c
  - T(N) = T(N-2) + c + c
  - T(N) = T(N-3) + c + c + c
  - generalized into T(N) = T(N-k) + kc

Complexity Analysis

Recurrence relation

- $T(0) = a$
- $T(1) = b$
- $T(N) = T(N-1) + c$

```
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Now solve it for the complexity
  - $T(N) = T(N-1) + c$
  - so $T(N-1) = T(N-2) + c$
  - $T(N) = T(N-2) + c + c$
  - $T(N) = T(N-3) + c + c + c$
  - generalized into $T(N) = T(N-k) + kc$
  - Base case when $N=0$ or $N=1$

Complexity Analysis

Recurrence relation

MONASH University

- $T(0) = a$
- $T(1) = b$
- $T(N) = T(N-1) + c$

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

- Now solve it for the complexity
  - $T(N) = T(N-1) + c$
  - so $T(N-1) = T(N-2) + c$
  - $T(N) = T(N-2) + c + c$
  - $T(N) = T(N-3) + c + c + c$
  - generalized into $T(N) = T(N-k) + kc$
  - Base case when $N=0$ or $N=1$
  - $N-k = 0$, therefore base case when $k=N$. we replace into above…

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N-1) + c
- Now solve it for the complexity
  - T(N) = T(N-1) + c
  - so T(N-1) = T(N-2) + c
  - T(N) = T(N-2) + c + c
  - T(N) = T(N-3) + c + c + c
  - generalized into T(N) = T(N-k) + kc
  - Base case when N=0 or N=1
  - N-k = 0, therefore base case when k=N. we replace into above…
  - T(N) = T(N-N) + Nc = T(0) + Nc = a + Nc

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Complexity Analysis

Recurrence relation

- $T(0) = a$
- $T(1) = b$
- $T(N) = T(N-1) + c$
- Now solve it for the complexity
  - $T(N) = T(N-1) + c$
  - so $T(N-1) = T(N-2) + c$
  - $T(N) = T(N-2) + c + c$
  - $T(N) = T(N-3) + c + c + c$
  - generalized into $T(N) = T(N-k) + kc$
  - Base case when $N=0$ or $N=1$
  - $N-k = 0$, therefore base case when $k=N$. we replace into above…
  - $T(N) = T(N-N) + Nc = T(0) + Nc = a + Nc = O(N)$, eliminating the constant

```python
def power(x,n):
    """
    Returns x^n calculated recursively
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * power (x,n-1)
```

Questions?

Complexity Analysis
Recurrence relation

- Let us try another one
- Come up with the recurrence relation

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    elif n%2 == 0:
        return power (x*x, n//2)
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis

Recurrence relation

- Let us try another one
- Come up with the recurrence relation

- Fun fact, this was a programming question for the exam…

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    elif n%2 == 0:
        return power (x*x, n//2)
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis

Recurrence relation

- Let us try another one
- Come up with the recurrence relation

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

- Fun fact, this was a programming question for the exam…
- Do you understand the code?

Complexity Analysis
Recurrence relation

- Let us try another one
- Come up with the recurrence relation

- Fun fact, this was a programming question for the exam…
- Do you understand the code?

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Should be *x

Complexity Analysis

Recurrence relation

- T(0) = a

```
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis
Recurrence relation

- T(0) = a
- T(1) = b

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis

Recurrence relation

- $T(0) = a$
- $T(1) = b$
- $T(N) = T(N//2) + c$ when N even

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis
Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis

Recurrence relation

MONASH University

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

  – Not that it isn't T(N) = T(N//2) * N
  WHY?

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

- Now you solve it…

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis
Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

- Now you solve it…
  Let me take the even one to start

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

- Now you solve it…
  Let me take the even one to start
  - T(N) = T(N//2) + c
  - T(N) = T(N//4) + 2c
  - T(N) = T(N//8) + 3c

```
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

Complexity Analysis

Recurrence relation

MONASH University

- $T(0) = a$
- $T(1) = b$
- $T(N) = T(N//2) + c$ when N even
- $T(N) = T(N//2) + d$ when N odd

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

- Now you solve it…
  Let me take the even one to start
  - $T(N) = T(N//2) + c$
  - $T(N) = T(N//4) + 2c$
  - $T(N) = T(N//8) + 3c$ can you see the pattern?

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

- Now you solve it…
  Let me take the even one to start
  - T(N) = T(N//2) + c
  - T(N) = T(N//4) + 2c
  - T(N) = T(N//8) + 3c can you see the pattern?
  - T(N) = T(N//2^k) + kc

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

```
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

- Now you solve it…
  Let me take the even one to start
  - T(N) = T(N//2) + c
  - T(N) = T(N//4) + 2c
  - T(N) = T(N//8) + 3c can you see the pattern?
  - T(N) = T(N//2^k) + kc
  - Base when N//2^k = 1… thus N = 2^k

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

- Now you solve it…
  Let me take the even one to start
  - T(N) = T(N//2) + c
  - T(N) = T(N//4) + 2c
  - T(N) = T(N//8) + 3c can you see the pattern?
  - T(N) = T(N//2^k) + kc
  - Base when N//2^k = 1… thus N = 2^k which is k = log N

Complexity Analysis

Recurrence relation

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

- Now you solve it…
  Let me take the even one to start
  - T(N) = T(N//2) + c
  - T(N) = T(N//4) + 2c
  - T(N) = T(N//8) + 3c can you see the pattern?
  - T(N) = T(N//2^k) + kc = T(1) + log N c = b + log N c
  - Base when N//2^k = 1… thus N = 2^k which is k = log N

Complexity Analysis

Recurrence relation

```python
def power_squaring(x,n):
    """
    Returns x^n
    via exponential by squaring
    """
    if n == 0:
        return 1
    elif n == 1:
        return x
    # when power is even
    # N^4 = N^2 * N^2
    elif n%2 == 0:
        return power (x*x, n//2)
    # when the power is odd
    # N^9 = N^4 & N^4 * N
    elif n%2 == 1:
        return power (x*x, n//2) * n
```

- T(0) = a
- T(1) = b
- T(N) = T(N//2) + c when N even
- T(N) = T(N//2) + d when N odd

- Now you solve it…
  Let me take the even one to start
  - T(N) = T(N//2) + c
  - T(N) = T(N//4) + 2c
  - T(N) = T(N//8) + 3c can you see the pattern?
  - T(N) = T(N//2^k) + kc = T(1) + log N c = b + log N c = O(log N)
  - Base when N//2^k = 1… thus N = 2^k which is k = log N

Questions?

- Both functions we saw just now are similar
- They are power functions x^n

- Both functions we saw just now are similar
- They are power functions x^n

- But we also saw how their complexity differs
  - O(N)
  - O(log N)

- Both functions we saw just now are similar
- They are power functions x^n

- But we also saw how their complexity differs
  - O(N) = normal power
  - O(log N) = exponential by squaring

Questions?

- So now you know why functions have such complexity?

- So now you know why functions have such complexity?
- Some of the other common ones…

Recurrence and complexity

Recurrence relation:

$T(N) = T(N/2) + c$

$T(1) = b$

Example algorithm?

Binary search

Solution:

$O(\log N)$

Recurrence and complexity

Recurrence relation:

$T(N) = T(N-1) + c$

$T(1) = b$

Example algorithm?

Linear search

Solution:

$O(N)$

Recurrence and complexity

Recurrence relation:

$T(N) = 2*T(N/2) + c*N$

$T(1) = b$

Example algorithm?

Merge sort

Solution:

$O(N \log N)$

Recurrence and complexity

Recurrence relation:

$T(N) = T(N-1) + c*N$

$T(1) = b$

Example algorithm?

Selection sort

Solution:

$O(N^2)$

Recurrence and complexity

Recurrence relation:

$T(N) = 2*T(N-1) + c$

$T(0) = b$

Example algorithm?

Naïve recursive Fibonacci

Solution:

$O(2^N)$

- So now you know why functions have such complexity?
- Some of the other common ones…

- Exam would also ask you to proof by induction for complexity…

- So now you know why functions have such complexity?
- Some of the other common ones…

- Exam would also ask you to proof by induction for complexity…
  - We will discuss more in the tutorial
  - Now, let us try to make my life difficult in the zoom session later this week…

Questions?

Thank You