September 29, 2017

Analysis of Algorithms

CS 141, Fall 2017

UC RIVERSIDE
UNIVERSITY OF CALIFORNIA

1

Analysis of Algorithms: Issues

- Correctness/Optimality
- Running time ("*time complexity*")
- Memory requirements ("*space complexity*")
- Power
- I/O utilization
- Ease of implementation
- …

2

Worst Case Time-Complexity

- Definition: The worst case time-complexity of an algorithm *A* is the *asymptotic* running time of *A* as a *function of the size of the input*, when the input is the one that makes the algorithm *slower* in the limit
- How do we measure the running time of an algorithm?

5

Python (the language)

- We will use python code to describe algorithms (sometime mixed w English)
- Python is
  - High-level (easy to read/use/learn)
  - Object-oriented
  - Interpreted (but can be compiled)
  - Portable
  - Free/open-source

6

Python: an example

- Algorithm for finding the maximum element of an array

```
def iMax(A):
    currentMax = A[0]
    for i in range(1,len(A)):
        if currentMax < A[i]:
            currentMax = A[i]
    return currentMax
```

7

… more python-ish

- Algorithm for finding the maximum element of an array

```
def iMax(A):
    currentMax = A[0]
    for x in A[1:]:
        if currentMax < x:
            currentMax = x
    return currentMax
```

8

Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Searching for key in a list of n items	Number of items in the list, i.e., n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	size of n = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or #edges	Visiting a vertex or traversing an edge

10

Example (Max iterative)

```
def iMax(A):
    currentMax = A[0]
    for i in range(len(A)):
        if currentMax < A[i]:
            currentMax = A[i]
    return currentMax
```

The program executes n-1 comparisons (irrespective from the type of input) where n=*len*(A) therefore the worst case time-complexity is $O(n)$

11

Example (Max recursive)

```
def rMax(A):
    if len(A) == 1:
        return A[0]
    return max(rMax(A[1:]),A[0])
```

The program executes *n-1* comparisons (irrespective from the type of input) therefore the worst case time-complexity is $O(n)$

12

Asymptotic notation

Section 0.3 of the textbook

13

The "Big-Oh" Notation

- Definition: Given functions ***f(n)*** and ***g(n)***, we say that ***f(n)*** is *O(**g(n)**)*

  if and only if

  there are positive constants ***c*** and n_0 such that $f(n) \leq c\ g(n)$ for $n \geq n_0$

14

The "Big-Oh" Notation

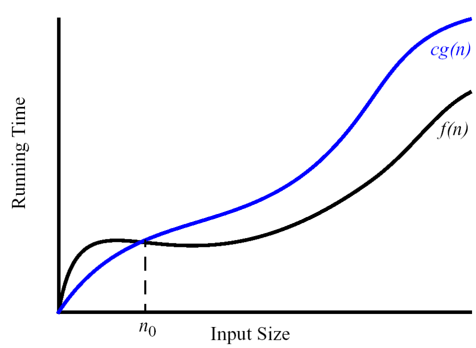


Figure 1.3: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

15

Proof

- $f(n)=2n+6$
- $g(n)=n$
- $2n+6 \leq 4n$ when $n \geq 3$
- So, if we choose $c=4,$ then $n_0=3$ satisfies $f(n)\leq c\ g(n)$ for $n \geq n_0$
- Conclusion: $2n+6$ is $O(n)$

17

Asymptotic Notation

- Note: Even though it is correct to say "$7n - 3$ is $O(n^3)$", a more precise statement is "$7n - 3$ is $O(n)$"

- Simple Rule: Drop lower order terms and constant factors

  $7n\text{-}3$ **is** $O(n)$

  $8n^2 \log n + 5n^2 + n$ **is** $O(n^2 \log n)$

18

Asymptotic Notation

- Special classes of algorithms
  - constant: $O(1)$
  - logarithmic: $O(log\ n)$
  - linear: $O(n)$
  - quadratic: $O(n^2)$
  - cubic: $O(n^3)$
  - polynomial: $O(n^k),\ k \geq 1$
  - exponential: $O(a^n),\ n > 1$

19

Asymptotic Notation

- "Relatives" of the Big-Oh
  - $\Omega(f(n))$: Big Omega
    - asymptotic *lower* bound
  - $\Theta(f(n))$: Big Theta
    - asymptotic *tight* bound

20

Big Omega

- Definition: Given two functions ***f(n)*** and ***g(n)*****,** we say that ***f(n)*** is $\Omega(\mathbf{g(n)})$ *if and only if* there are positive constants ***c*** and n_0 such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

- Property: ***f(n)*** is $\Omega(\mathbf{g(n)})$ *iff* ***g(n)*** is $O(\mathbf{f(n)})$

21

Big Theta

- Definition: Given two functions ***f(n)*** and ***g(n)*****,** we say that ***f(n)*** is $\Theta(\mathbf{g(n)})$ *if and only if* there are positive constants c_1, c_2 and n_0 such that $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for $n \geq n_0$
- Property: ***f(n)*** is $\Theta(\mathbf{g(n)})$ if and only if "***f(n)*** is $O(\mathbf{g(n)})$ AND ***f(n)*** is $\Omega(\mathbf{g(n)})$"

22

Establishing order of growth using limits

$$\lim_{n\to\infty} f(n)/g(n) = \begin{cases} \mathbf{0} & \text{order of growth of } \boldsymbol{f(n)} < \text{ order of growth of } \boldsymbol{g(n)} \\ \boldsymbol{c > 0} & \text{order of growth of } \boldsymbol{f(n)} = \text{ order of growth of } \boldsymbol{g(n)} \\ \infty & \text{order of growth of } \boldsymbol{f(n)} > \text{ order of growth of } \boldsymbol{g(n)} \end{cases}$$

Examples:

- **$10n$ vs. n^2**
- **$n(n+1)/2$ vs. n^2**

24

Orders of growth: some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is
- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ in $\Theta(n^k)$
- Exponential functions a^n have different orders of growth for different a's
- order $\log n$ < order n < order $n \log n$ < order n^k ($k \geq 2$ constant) < order a^n < order $n!$ < order n^n
- Caution: Be aware of very large constant factors

25

Suppose each operation takes 1 nanoseconds (10^{-9} seconds)

n	lg n	n	n lg n	n^2	2^n	n!
10	0.003μs	0.01μs	0.033μs	0.1μs	1μs	3.63ms
20	0.004μs	0.02μs	0.086μs	0.4μs	1ms	77.1years
30	0.005μs	0.02μs	0.147μs	0.9μs	1sec	>10^{15}years
100	0.007μs	0.1μs	0.644μs	10μs	>10^{13}years	
10,000	0.013μs	10μs	130μs	100ms		
1,000,000	0.020μs	1ms	19.92μs	16.7min		

- For n < 10, the difference is insignificant.
- Θ (n!) algorithms are useless well before n = 20.
- Θ (2^n) algorithms are practical for n < 40.
- Θ (n^2) and Θ (n lg n) are both useful, but Θ (n lg n) is significantly faster.

26

Time analysis for iterative algorithms

Steps

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst* case(s) for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

27

Example of Asymptotic Analysis

```
def prefixAverages1(X):
    A = []
    for i in range(len(X)):
        a = 0
        for j in range(i+1):
            a += X[j]   ← step
        A.append(a/float(i+1))
    return A
```

n iterations

$i+1$ iterations

…then the algorithm is $O(n^2)$

28

A faster algorithm

- Observe that

$$
\begin{aligned}
A[i-1] &= (X[0]+X[1]+\cdots+X[i-1])/i \\
A[i] &= (X[0]+X[1]+\cdots+X[i-1]+X[i])/(i+1).
\end{aligned}
$$

29

A linear-time algorithm

```
def prefixAverages2(X):
    A,a = [],0
    for i in range(len(X)):
        a = a + X[i]
        A.append(a/float(i+1))
    return A
```

30

A trickier example

- Analyze the worst-case time complexity of the following algorithm, and give a tight bound using the big-theta notation

```
def weirdLoop(n):
    i = n
    while i >= 1:
        for j in range(i):
            print 'Hello'
        i = i/2
    return
```

31

Math review

Appendix A of the textbook

32

Summations

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a} \text{ when } a > 0,\ a \neq 1$$

$$\text{e.g., } \sum_{i=0}^{n} 2^i = 1+2+4+\ldots+2^n = 2^{n+1}-1$$

33

Binomial expansion

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

In particular, if we choose $a = 1,\ b = 1$

we get $2^n = \sum_{k=0}^{n} \binom{n}{k}$

34

Bounding sums

- *Upper bound*: Any sum is at most the number of terms times the maximum term
  - *Example*: 1+4+9+...+n^2 is at most $n*n^2 = n^3$
- *Lower bound*: If the terms are non-negative, any sum is at least half the number of terms times the median term
  - *Example*: 1+4+9+...+n^2 is at least $(n/2)*(n/2)^2 = n^3/8$

35

Proving (or disproving) $p \rightarrow q$

- *Counterexample* (used to prove that $p \rightarrow q$ is false showing one particular choice of p that makes q false)

- *Direct* proof ($p \rightarrow p_1 \rightarrow \ldots \rightarrow p_n \rightarrow q$)
- *Contrapositive* (prove that $\sim q \rightarrow \sim p$)
- *Contradiction* (assume p and $\sim q$ true, find a contradiction)
- *Induction* (prove base case + induction)

36

Induction proof

Theorem: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Proof: by induction on n.

Base case: $n = 1$. Trivial since $1 = 1(1+1)/2$.

Induction step: $n \geq 2$. Assume the claim is true for any $n' < n$. Then $\sum_{i=1}^{n} i = n + \underbrace{\sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2}}_{\text{using induction}} = \frac{n(n+1)}{2}$

37

Recurrence Relation Analysis

Recurrence relation

- A *recurrence relation* is an equation that recursively define a sequence: each term of the sequence is defined as a function of the preceding term(s)
- For instance

$$f(n) = \begin{cases} 2 & n=1 \\ f(n-1)+n & n>1 \end{cases}$$

General form

Base of recursion

Running time for base

$$T(n) = \begin{cases} c \quad \text{if } n = n_0 \\ a.T(f(n)) + g(n) \quad \text{otherwise} \end{cases}$$

Number of times recursive call is made

All other processing not counting recursive calls

Size of problem solved by recursive call

42

Definition of the Factorial function

$$F(n) = \begin{cases} 1 & n = 0 \\ nF(n-1) & n \geq 1 \end{cases}$$

Recursive implementation

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Time complexity?

$$T(n) = \begin{cases} & n \leq \\ & n > \end{cases}$$

43

Definition of the Fibonacci function

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

Recursive implementation

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Time complexity?

$$T(n) = \begin{cases} & n \leq \\ & n > \end{cases}$$

44

Example

```
def bugs(n):
    if n <= 1:
        do_something()
    else:
        bugs(n-1)
        bugs(n-2)
        for i in range(n):
            do_something_else()
```

$$T(n) = \begin{cases} c_1 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + nc_2 & \text{otherwise} \end{cases}$$

45

Example

```
def daffy(n):
    if n == 1 or n == 2:
        do_something()
    else:
        daffy(n-1)
        for i in range(n):
            do_something_else()
        daffy(n-1)
```

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \text{ or } n = 2 \\ 2T(n-1) + nc_2 & \text{otherwise} \end{cases}$$

46

Example

```
def elmer(n):
    if n == 1:
        do_something()
    elif n == 2:
        do_something_else()
    else:
        for i in range(n):
            elmer(n-1)
            do_something_different()
```

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ c_2 & \text{if } n = 2 \\ n(T(n-1) + c_3) & \text{otherwise} \end{cases}$$

47

Example

```
def yosemite(n):
    if n == 1:
        do_something()
    else:
        for i in range(1,n):
            yosemite(i)
            do_something_different()
```

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ \sum_{i=1}^{n-1} \left(T(i) + c_2 \right) & \text{otherwise} \end{cases}$$

48

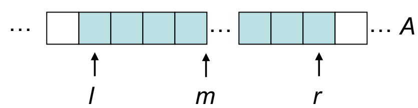
MergeSort

- MergeSort is a divide & conquer algorithm
  - *Divide*: divide an n-element sequence into two subsequences of approx $n/2$ elements
  - *Conquer*: sort the subsequences recursively
  - *Combine*: merge the two sorted subsequences to produce the final sorted sequence

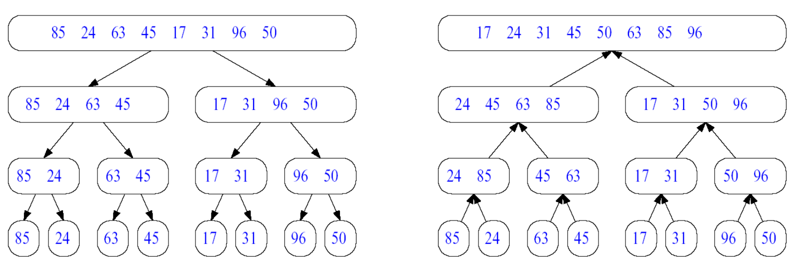
49

MergeSort

```
def mergesort(A):
    if len(A) < 2:
        return A
    else:
        m = len(A)/2
        l = mergesort(A[:m])
        r = mergesort(A[m:])
        return merge(l,r)
```



Example



(a) (b)

Figure 4.2: Merge-sort tree T for an execution of the merge-sort algorithm on a sequence with 8 elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T.

Merge of MergeSort

```
def merge(l,r):
    result, i, j = [], 0, 0
    while i < len(l) and j < len(r):
        if l[i] <= r[j]:
            result.append(l[i])
            i += 1
        else:
            result.append(r[j])
            j += 1
    result += l[i:]
    result += r[j:]
    return result
```

52

MergeSort Analysis

- Divide: Just computes the middle of the subsequence, thus takes constant time:

  $D(n) = \Theta(1)$
- Conquer: We solve 2 subproblems of size approximately n/2:

  $a = 2, \quad b = 2$
- Combine: Merge takes $\Theta(n)$:

  $C(n) = \Theta(n)$
- Noting that $\Theta(n) + \Theta(1)$ is still $\Theta(n)$, we get:

  $$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$
- Later we will see that:

  $T(n) = \Theta(n \lg n)$

53

“Visual” Analysis

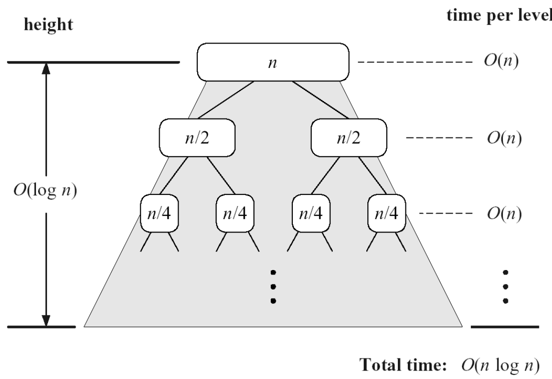


Figure 4.4: A visual analysis of the running time of merge-sort. Each node of the merge-sort tree is labeled with the size of its subproblem.

54

Solving Recurrence Relation

55

Methods

- Two methods for solving recurrences
  - Iterative substitution method
  - Master method

  - (not covered: Recursion Tree)
  - (not covered: Guess-and-Test method)

56

Iterative substitution

- Assume *n* large enough
- Substitute *T* on the right-hand side of the recurrence relation
- Iterate the substitution until we see a pattern which can be converted into a general closed-form formula

57

MergeSort recurrence relation

$$T(N) = 2T\left(\frac{N}{2}\right) + N \qquad \text{for } N \geq 2$$
$$T(1) = 1$$

$$\begin{aligned}
T(N) &= 2\left(2T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N \\
&= 4T\left(\frac{N}{4}\right) + 2N \\
&= 4\left(2T\left(\frac{N}{8}\right) + \frac{N}{4}\right) + 2N \\
&= 8T\left(\frac{N}{8}\right) + 3N \\
&\cdots \\
&= \boxed{2^i T}\left(\frac{N}{2^i}\right) + iN
\end{aligned}$$

$T(1) = 1$

…

The expansion stops for $i = \log_2 N$, so that

$T(N) = N + N \log_2 N$

Verify the correctness

- How to verify the solution is correct?
- Use proof by induction!
- Important: make sure the constant c works for both the base case and the induction step

60

Proof by induction

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

Fact: $T(n) \in O(n \log_2 n)$

Proof. Base case: $T(2) = 2T(1) + 2 = 4 \leq c(2\log_2 2) = 2c$.

Hence, $c \geq 2$.

Induction hypothesis: $T(n/2) \leq c\frac{n}{2}\log_2\frac{n}{2}$

Induction: $T(n) = 2T(n/2) + n$

$$\leq 2c\frac{n}{2}\log_2\frac{n}{2} + n$$

$$= cn\log_2\frac{n}{2} + n = cn\log_2 n - cn\log_2 2 + n$$

$$= cn\log_2 n + n(1-c) \leq cn\log_2 n \text{ when } c \geq 1$$

Choose $c = 2$.

The constant c used in the induction and the base case has to be the *same* !

61

Wrong proof by induction

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

Fact (wrong): $T(n) \in O(n)$

Proof. Base case: $T(1) = 1 \le c1$, hence $c \ge 1$

Induction hypothesis: $T(n/2) \le c(n/2)$

Induction: $T(n) = 2T(n/2) + n$

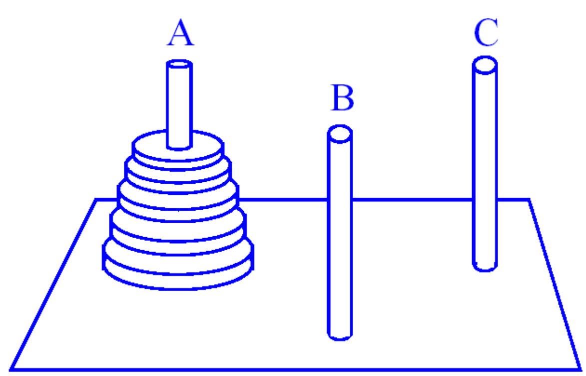
$$\le 2c(n/2) + n$$

$$= cn + n \in O(n)$$

proof is WRONG, but where is the mistake?

62

Towers of Hanoi



63

Towers of Hanoi

Goal: transfer all N disks from peg A to peg C

Rules:

- move one disk at a time
- never place larger disk above smaller one

Recursive solution:

- transfer $N - 1$ disks from A to B
- move largest disk from A to C
- transfer $N - 1$ disks from B to C

Total number of moves:

- $T(N) = 2\,T(N - 1) + 1$

64

Towers of Hanoi

```
def hanoi(n, a='A', b='B', c='C'):
    if n == 0:
        return
    hanoi(n-1, a, c, b)
    print a, '->', c
    hanoi(n-1, b, a, c)
```

65

Towers of Hanoi: Recurrence Relation

Solve

$$T(N) = \begin{cases} 2T(N-1)+1 & N>1 \\ 1 & N=1 \end{cases}$$

66

Towers of Hanoi: Unfolding the relation

$$\begin{aligned} T(N) &= 2\,(\,2\,T(N-2)+1\,)+1 = \\ &= 4\,T(N-2)+2+1= \\ &= 4\,(\,2\,T(N-3)+1\,)+2+1 = \\ &= 8\,T(N-3)+4+2+1 = \\ &\dots \\ &= 2^i\,T(N-i)+2^{i-1}+2^{i-2}+\dots+2^1+2^0 \end{aligned}$$

the expansion stops when $i = N - 1$

$$T(N) = 2^{N-1}+2^{N-2}+2^{N-3}+\dots+2^1+2^0$$

This is a ***geometric sum***, so that we have:

$$T(N) = 2^N - 1 \in \Theta\left(2^N\right)$$

67

Problem

Problem: Solve exactly (by iterative substitution)

$$T(n) = \begin{cases} 4 & n = 1 \\ 4T(n-1) + 3 & n > 1 \end{cases}$$

68

Problem

Problem: Solve exactly (by iterative substitution)

$$T(n) = \begin{cases} 4 & n = 1 \\ 4T(n-1) + 3 & n > 1 \end{cases}$$

Solution: $T(n) = 4^n + 4^{n-1} - 1$

Proof?

69

Another example

$$T(N) = 2T(\sqrt{N}) + 1 \qquad T(2) = 0$$

$$2T(N^{1/2}) + 1$$
$$2(2T(N^{1/4}) + 1) + 1$$
$$4T(N^{1/4}) + 1 + 2$$
$$8T(N^{1/8}) + 1 + 2 + 4$$
$$\ldots$$

70

Another example

$$2^i T\left(N^{\frac{1}{2^i}}\right) + 2^0 + 2^1 + \ldots + 2^{i-1}$$

The expansion stops for $N^{\frac{1}{2^i}} = 2$
i.e., $i = \log\log N$

$$T(N) = 2^0 + 2^1 + \ldots + 2^{\log\log N - 1} = \log N - 1$$

71

Master Theorem method

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d, \end{cases}$$

Theorem 5.6 [The Master Theorem]: *Let $f(n)$ and $T(n)$ be defined as above.*

1. *If there is a small constant $\varepsilon > 0$ such that $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.*
2. *If there is a constant $k \geq 0$ such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If there are small constants $\varepsilon > 0$ and $\delta < 1$ such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.*

n/b stands for $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

72

Master Theorem

Condition on *f(n)*	Condition	Conclusion on *T(n)*
$O\left(n^{\log_b a-\varepsilon}\right)$	$\varepsilon > 0$	$\Theta\left(n^{\log_b a}\right)$
$\Theta\left(n^{\log_b a} \log^k n\right)$	$k \geq 0$	$\Theta\left(n^{\log_b a} \log^{k+1} n\right)$
$\Omega\left(n^{\log_b a+\varepsilon}\right)$	$\varepsilon > 0,\ \delta < 1$ $af\left(n/b\right) \leq \delta f(n)$	$\Theta\left(f(n)\right)$

73

Master method (first case)

Example 5.7: *Consider the recurrence*

$$T(n) = 4T(n/2) + n.$$

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. *Thus, we are in Case 1, for* $f(n)$ *is* $O(n^{2-\varepsilon})$ *for* $\varepsilon = 1$. *This means that* $T(n)$ *is* $\Theta(n^2)$ *by the master method.*

74

Master method (second case)

Example 5.8: *Consider the recurrence*

$$T(n) = 2T(n/2) + n\log n,$$

which is one of the recurrences given above. In this case, $n^{\log_b a} = n^{\log_2 2} = n$. *Thus, we are in Case 2, with* $k = 1$, *for* $f(n)$ *is* $\Theta(n \log n)$. *This means that* $T(n)$ *is* $\Theta(n \log^2 n)$ *by the master method.*

75

Master method: binary search (second case)

- The Master Theorem allows us to ignore the floor or ceiling function around n/b in T(n/b) in general.
- Binary Search has for any $n > 0$ a running time of

  $T(n) = T(n/2) + \Theta(1)$.

  Hence $a = 1$, $b = 2$, $f(n) = \Theta(1)$. Since $1 = n^{\log_2 1}$ the second case applies and we get:

  $T(n) = \Theta(\lg n)$

76

Master method: merge-sort (second case)

- For arbitrary $n > 0$, the running time of Merge-Sort is

  $T(n) = \Theta(1)$ if $n = 1$
  $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$ if $n > 1$

  We can approximate this from below and above by

  $T(n) = 2\ T(\lfloor n/2 \rfloor) + \Theta(n)$ if $n > 1$
  $T(n) = 2\ T(\lceil n/2 \rceil) + \Theta(n)$ if $n > 1$

  respectively. According to the Master Theorem, both have the same solution which we get by taking

  $a = 2$, $b = 2$, $f(n) = \Theta(n)$.

  Since $n = n^{\log_2 2}$, the second case applies and we get:

  $T(n) = \Theta(n \lg n)$

Master method (third case)

Example 5.9: *Consider the recurrence*

$$T(n) = T(n/3) + n,$$

which is the recurrence for a geometrically decreasing summation that starts with n. *In this case,* $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. *Thus, we are in Case 3, for* $f(n)$ *is* $\Omega(n^{0+\varepsilon})$, *for* $\varepsilon = 1$, *and* $af(n/b) = n/3 = (1/3)f(n)$. *This means that* $T(n)$ *is* $\Theta(n)$ *by the master method.*

Example 5.10: *Consider the recurrence*

$$T(n) = 9T(n/3) + n^{2.5}.$$

In this case, $n^{\log_b a} = n^{\log_3 9} = n^2$. *Thus, we are in Case 3, for* $f(n)$ *is* $\Omega(n^{2+\varepsilon})$, *for* $\varepsilon = 1/2$, *and* $af(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2} f(n)$. *This means that* $T(n)$ *is* $\Theta(n^{2.5})$ *by the master method.*

78

Summary (1/3)

- <u>Goal:</u> analyze the worst-case time-complexity of iterative and recursive algorithms
- Tools:
  - Pseudo-code/Python
  - Big-O, Big-Omega, Big-Theta notations
  - Recurrence relations
  - Discrete Math (summations, induction proofs, methods to solve recurrence relations)

79

Summary (2/3)

- Pure iterative algorithm:
  - Analyze the loops
  - Determine how many times the inner core is repeated as a function of the input size
  - Determine the worst-case for the input
  - Write the number of repetitions as a function of the input size
  - Simplify the function using big-O or big-Theta notation (optional)

80

Summary (3/3)

- Recursive + iterative algorithm:
  - Analyze the recursive calls and the loops
  - Determine how many recursive calls are made and the size of the arguments of the recursive calls
  - Determine how much extra processing (loops) is done
  - Determine the worst-case for the input
  - Derive a recurrence relation
  - Solve the recurrence relation
  - Simplify the solution using big-O, or big-Theta

81