CHAPTER 1

BASIC CONCEPT

How to create programs

- Requirements
- Analysis: bottom-up vs. top-down
- Design: data objects and operations
- Refinement and Coding
- Verification
  - Program Proving
  - Testing
  - Debugging

Algorithm

- Definition
  An *algorithm* is a finite set of instructions that accomplishes a particular task.
- Criteria
  - input
  - output
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

Data Type

- Data Type
  A *data type* is a collection of *objects* and a set of *operations* that act on those objects.
- Abstract Data Type
  An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

Specification vs. Implementation

- Operation specification
  - function name
  - the types of arguments
  - the type of the results
- Implementation independent

***Structure 1.1:**Abstract data type *Natural_Number* (p.17)

structure Natural_Number is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all x, y ∈ *Nat_Number*; *TRUE*, *FALSE* ∈ *Boolean*

and where +, -, <, and == are the usual integer operations.

Nat_No Zero ( ) ::= 0

Boolean Is_Zero(x) ::= **if** (x) **return** *FALSE*
else return *TRUE*

Nat_No Add(x, y) ::= **if** ((x+y) <= *INT_MAX*) **return** x+y
else return *INT_MAX*

Boolean Equal(x,y) ::= **if** (x== y) **return** *TRUE*
else return *FALSE*

Nat_No Successor(x) ::= **if** (x == *INT_MAX*) **return** x
else return x+1

Nat_No Subtract(x,y) ::= **if** (x<y) **return** 0
else return x-y

end *Natural_Number*

::= is defined as

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
  - …
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

Space Complexity $S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

***Program 1.9:** Simple arithmetic function (p.19)

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}
```

$S_{abc}(I) = 0$

***Program 1.10:** Iterative function for summing a list of numbers (p.20)

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i<n; i++)
     tempsum += list [i];
  return tempsum;
}
```

$S_{sum}(I) = 0$

Recall: pass the address of the first element of the array & pass by value

9

***Program 1.11:** Recursive function for summing a list of numbers (p.20)

```
float rsum(float list[ ], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
}
```

$S_{sum}(I)=S_{sum}(n)=6n$

Assumptions:

***Figure 1.1:** Space needed for one recursive call of Program 1.11 (p.21)

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2(unless a far address)
TOTAL per recursive call		6

10

Time Complexity

$T(P)=C+T_P(I)$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition $\boxed{T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)}$
  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example
  – abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  – abc = a + b + c

Regard as the same unit
machine independent

Methods to compute the step count

- Introduce variable count into programs
- Tabular method
  – Determine the total number of steps contributed by each statement
  step per execution × frequency
  – add up the contribution of all statements

Iterative summing of a list of numbers

***Program 1.12:** Program 1.10 with count statements (p.23)*

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;          /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;        /* last execution of for */
   return tempsum;
   count++;        /* for return */
}
```

2n + 3 steps

13

***Program 1.13:** Simplified version of Program 1.12 (p.23)*

```
float sum(float list[ ], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
       count += 2;
   count += 3;
   return 0;
}
```

2n + 3 steps

14

Recursive summing of a list of numbers

***Program 1.14:** Program 1.11 with count statements added (p.24)

```
float rsum(float list[ ], int n)
{
        count++;     /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}
```

$\boxed{2n+2}$

15

Matrix addition

***Program 1.15:** Matrix addition (p.25)

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                  int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j= 0; j < cols; j++)
        c[i][j] = a[i][j] +b[i][j];
 }
```

16

***Program 1.16:** Matrix addition with count statements (p.25)

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                  int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
      count++; /* for i for loop */
      for (j = 0; j < cols; j++) {
        count++; /* for j for loop */
        c[i][j] = a[i][j] + b[i][j];
        count++; /* for assignment statement */
      }
      count++;    /* last time of j for loop */
 }
 count++;        /* last time of i for loop */
}
```

2rows * cols + 2 rows + 1

17

***Program 1.17:** Simplification of Program 1.16 (p.26)

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],
                  int c[ ][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for( i = 0; i < rows; i++) {
     for (j = 0; j < cols; j++)
        count += 2;
        count += 2;
   }
   count++;
}
```

$2rows \times cols + 2rows + 1$

Suggestion: Interchange the loops when rows >> cols

18

Tabular Method

***Figure 1.2:** Step count table for Program 1.10 (p.26)

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

19

Recursive Function to sum of a list of numbers

***Figure 1.3:** Step count table for recursive summing function (p.27)

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

20

Matrix Addition

***Figure 1.4:** Step count table for matrix addition (p.27)

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]. . .)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows. (cols+1)	rows. cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows. cols	rows. cols
}	0	0	0
Total			2rows. cols+2rows+1

21

Exercise 1

***Program 1.18:** Printing out a matrix (p.28)

```
void print_matrix(int matrix[ ][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i = 0; i < row; i++) {
    for (j = 0; j < cols; j++)
        printf(“%d”, matrix[i][j]);
    printf( “\n”);
  }
}
```

22

Exercise 2

***Program 1.19:Matrix multiplication function**(p.28)*

```
void mult(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE])
{
  int i, j, k;
  for (i = 0; i < MAX_SIZE; i++)
     for (j = 0; j< MAX_SIZE;  j++) {
        c[i][j] = 0;
     for (k = 0; k < MAX_SIZE; k++)
         c[i][j]  +=  a[i][k] * b[k][j];
     }
}
```

23

Exercise 3

***Program 1.20:Matrix product function**(p.29)*

```
void prod(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE],
                                            int rowsa, int colsb, int colsa)
{
  int i, j, k;
  for (i = 0; i < rowsa; i++)
     for (j = 0; j< colsb;  j++) {
        c[i][j] = 0;
     for (k = 0; k< colsa; k++)
         c[i][j]  +=  a[i][k] * b[k][j];
     }
}
```

24

Exercise 4

***Program 1.21:Matrix transposition function** (p.29)

```
void transpose(int a[ ][MAX_SIZE])
{
   int i, j, temp;
   for (i = 0; i < MAX_SIZE-1; i++)
      for (j = i+1; j < MAX_SIZE;  j++)
         SWAP (a[i][j], a[j][i], temp);
}
```

25

Asymptotic Notation (O)

- Definition
  f(n) = O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.
- Examples
  - 3n+2=O(n) /* $3n+2\leq 4n$ for $n\geq 2$ */
  - 3n+3=O(n) /* $3n+3\leq 4n$ for $n\geq 3$ */
  - 100n+6=O(n) /* $100n+6\leq 101n$ for $n\geq 10$ */
  - $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2\leq 11n^2$ for $n\geq 5$ */
  - $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \leq 7*2^n$ for $n\geq 4$ */

26

Example

- Complexity of $c_1n^2+c_2n$ and c_3n
  - for sufficiently large of value, c_3n is faster than $c_1n^2+c_2n$
  - for small values of n, either could be faster
    - $c_1=1$, $c_2=2$, $c_3=100$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 98$
    - $c_1=1$, $c_2=2$, $c_3=1000$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 998$
  - break even point
    - no matter what the values of c1, c2, and c3, the n beyond which c_3n is always faster than $c_1n^2+c_2n$

27

- O(1): constant
- O(n): linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- O(logn)
- O(nlogn)

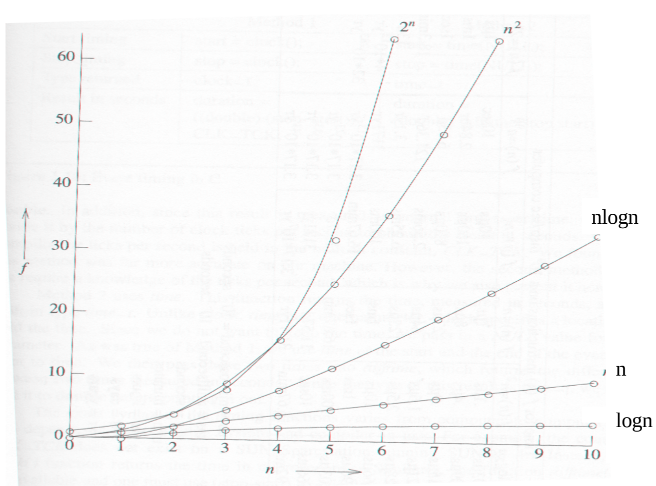
28

*Figure 1.7:Function values (p.38)

Time	Name	Instance characteristic n					
		1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

29

*Figure 1.8:Plot of function values(p.39)



30

***Figure 1.9:**Times on a 1 billion instruction per second computer(p.40)

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01µs	.03µs	.1µs	1µs	10µs	10sec	1µs
20	.02µs	.09µs	.4µs	8µs	160µs	2.84hr	1ms
30	.03µs	.15µs	.9µs	27µs	810µs	6.83d	1sec
40	.04µs	.21µs	1.6µs	64µs	2.56ms	121.36d	18.3min
50	.05µs	.28µs	2.5µs	125µs	6.25ms	3.1yr	13d
100	.10µs	.66µs	10µs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00µs	9.96µs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10.00µs	130.03µs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100.00µs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^7$yr	$3.17*10^{43}$yr	

µs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years

31

Solving recurrences

- Recurrences are a major tool for analysis of algorithms

Substitution method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

Example: $T(n) = 4T(n/2) + 100n$

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \le ck^3$ for $k < n$.
- Prove $T(n) \le cn^3$ by induction.

Example of substitution

$$\begin{aligned} T(n) &= 4T(n/2) + 100n \\ &\le 4c(n/2)^3 + 100n \\ &= (c/2)n^3 + 100n \\ &= cn^3 - ((c/2)n^3 - 100n) \quad \longleftarrow \textit{desired – residual} \\ &\le cn^3 \quad \longleftarrow \textit{desired} \end{aligned}$$

whenever $(c/2)n^3 - 100n \ge 0$, for example, if $c \ge 200$ and $n \ge 1$.

residual

Example (continued)

- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base:*** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.

This bound is not tight!

A tighter upper bound?

We shall prove that $T(n) = O(n^2)$.

Assume that $T(k) \le ck^2$ for $k < n$:

$$\begin{aligned} T(n) &= 4T(n/2) + 100n \\ &\le cn^2 + 100n \\ &\le cn^2 \end{aligned}$$

for ***no*** choice of $c > 0$. Lose!

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.

Inductive hypothesis: $T(k) \le c_1k^2 - c_2k$ for $k < n$.

$$\begin{aligned} T(n) &= 4T(n/2) + 100n \\ &\le 4(c_1(n/2)^2 - c_2(n/2)) + 100n \\ &= c_1n^2 - 2c_2n + 100n \\ &= c_1n^2 - c_2n - (c_2n - 100n) \\ &\le c_1n^2 - c_2n \quad \text{if } c_2 > 100. \end{aligned}$$

Pick c_1 big enough to handle the initial conditions.

Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$T(n)$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$n^2$$

$$T(n/4) \qquad\qquad T(n/2)$$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

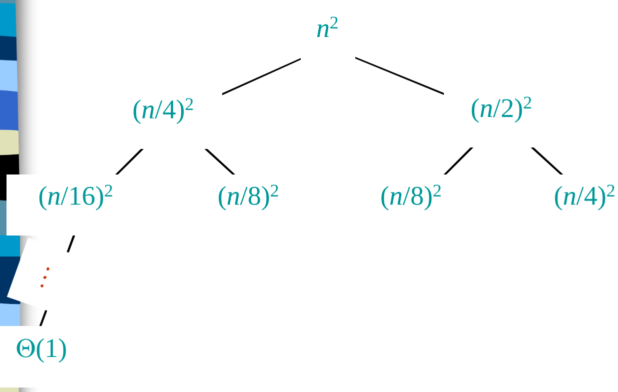
$$n^2$$

$$(n/4)^2 \qquad\qquad (n/2)^2$$

$$T(n/16) \qquad T(n/8) \qquad T(n/8) \qquad T(n/4)$$

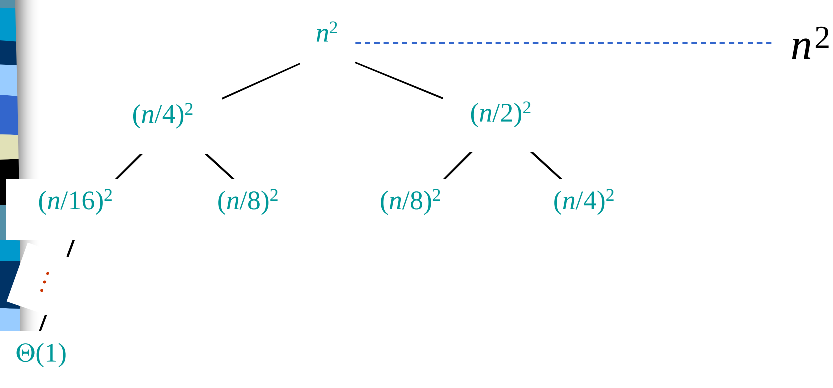
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

n^2 .... n^2

$(n/4)^2$ $(n/2)^2$ .... $\frac{5}{16}n^2$

$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$

$\vdots$

$\Theta(1)$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

n^2 .... n^2

$(n/4)^2$ $(n/2)^2$ .... $\frac{5}{16}n^2$

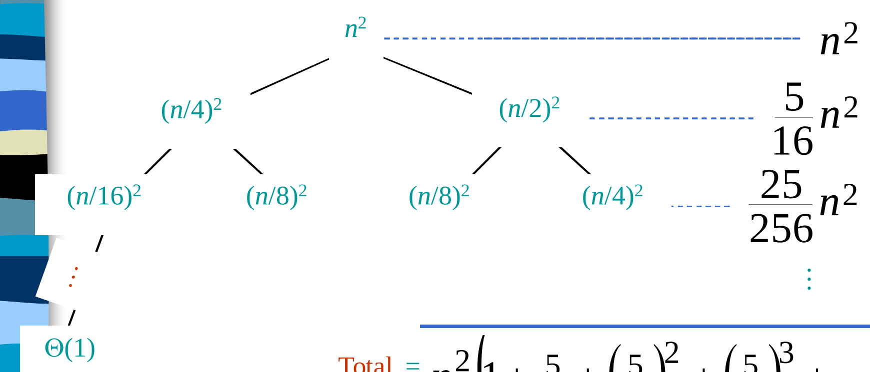
$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$ .... $\frac{25}{256}n^2$

$\vdots$ $\vdots$

$\Theta(1)$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Total $= n^2\left(1+\frac{5}{16}+\left(\frac{5}{16}\right)^2+\left(\frac{5}{16}\right)^3+\cdots\right)$

$= \Theta(n^2)$ *geometric series*

Appendix: geometric series

$$1+x+x^2+\cdots+x^n = \frac{1-x^{n+1}}{1-x} \quad \text{for } x \neq 1$$

$$1+x+x^2+\cdots = \frac{1}{1-x} \quad \text{for } |x| < 1$$

Return to last slide viewed.

The master method

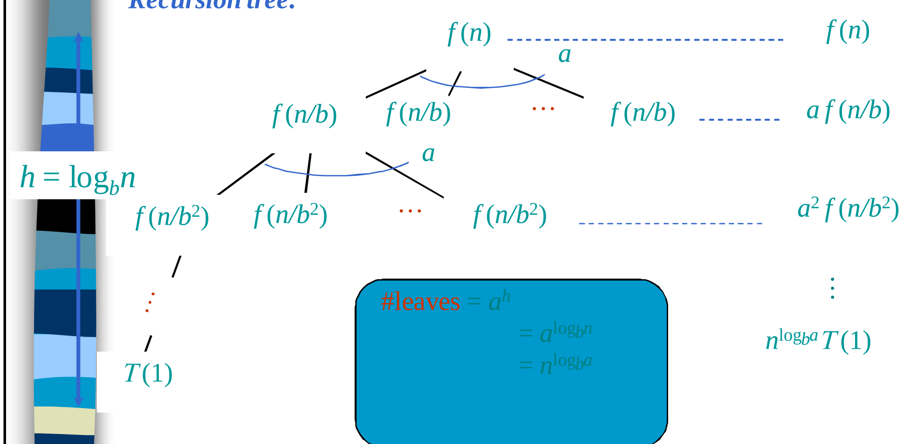
The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Idea of master theorem

Recursion tree:



Three common cases

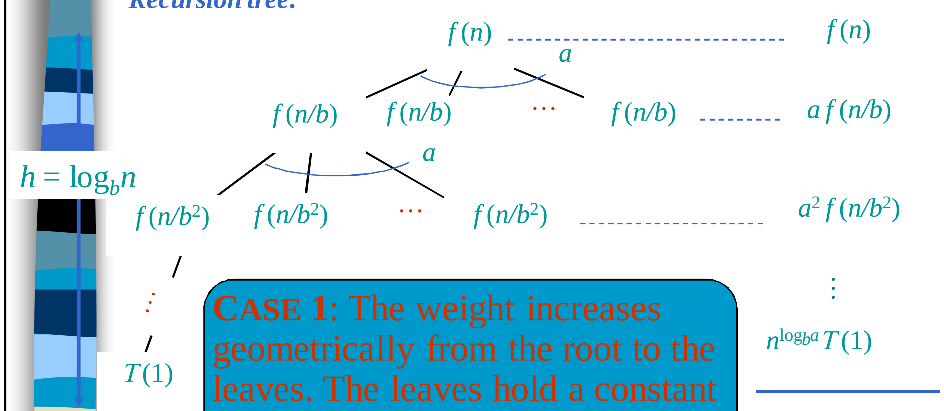
Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

Idea of master theorem

Recursion tree:



CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

Three common cases

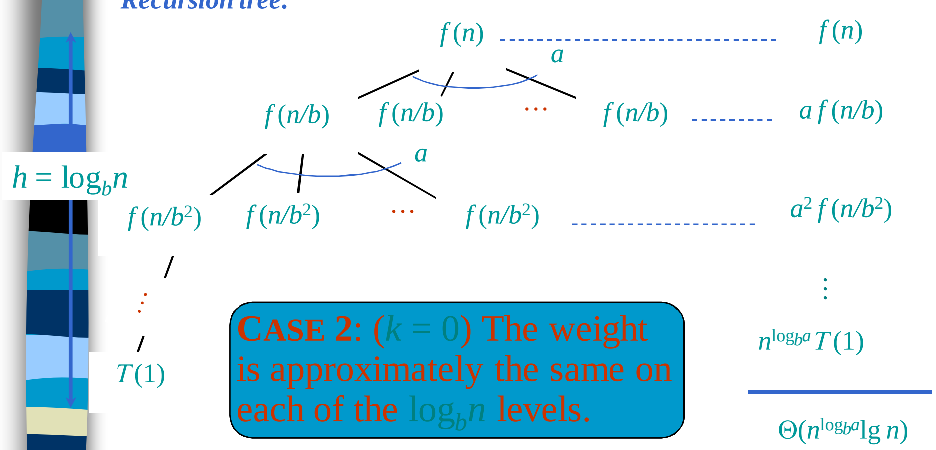
Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Idea of master theorem

Recursion tree:



Three common cases (cont.)

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.

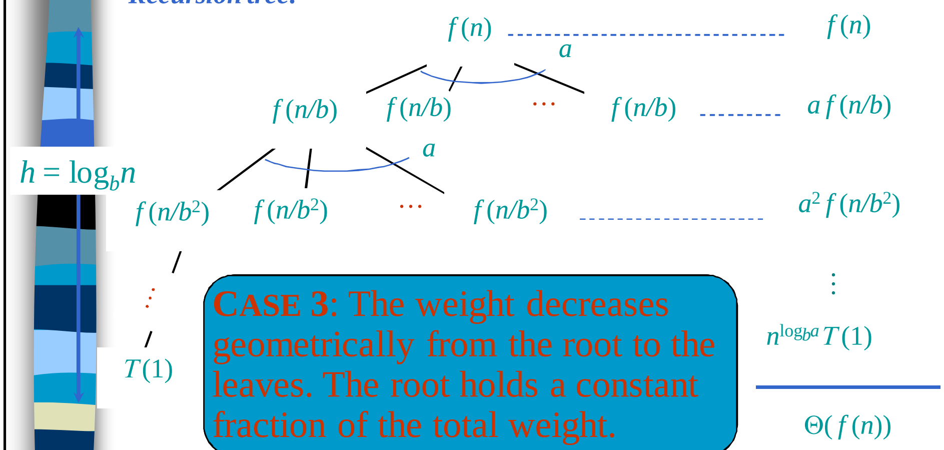
- $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

and $f(n)$ satisfies the ***regularity condition*** that $a\, f(n/b) \le c\, f(n)$ for some constant $c < 1$.

Solution: $T(n) = \Theta(f(n))$.

Idea of master theorem

Recursion tree:



Examples

Ex. $T(n) = 4T(n/2) + n$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$.
CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.
$\therefore\ T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$.
CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.
$\therefore\ T(n) = \Theta(n^2 \lg n)$.

Examples

Ex. $T(n) = 4T(n/2) + n^3$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$.
CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$
and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.
$\therefore\ T(n) = \Theta(n^3)$.

Ex. $T(n) = 4T(n/2) + n^2/\lg n$
$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2/\lg n$.
Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.