

CSC236 Week 7

Larry Zhang

learning continued...

Recap: last week

- We designed a couple algorithms using the divide-and-conquer technique
 - max_segment_sum()
 - MergeSort()
- Both of them had runtime like

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn + e & n > 1 \end{cases}$$

- After applying the substitution method (5-6 steps), we figured out that $T(n)$ is in $O(n \log n)$
- We said there is a quicker way to get $O(n \log n)$ in just **1 step**.

Typical form of the runtime of divide-and-conquer algorithms

$$T(n) = aT(n/b) + \Theta(n^k)$$

- **a** is the number of recursive calls
- **b** is the rate at which subproblem size decreases
- **k** represents the runtime of the non-recursive part of the algorithm
 - like max_crossing in max_seg_sum, k=1
 - merge in MergeSort, k=1

There is a **theorem** that, just given the values of **a**, **b** and **k**, can **directly** give us the asymptotic bound on T(n).

- no need to do repeated substitution



Master Theorem

Master Theorem

Let $T(n)$ be defined by the recurrence $T(n) = aT(n/b) + \Theta(n^k)$, for some constants $a \geq 1$, $b > 1$, and $k \geq 0$. Then we can conclude the following about the asymptotic complexity of $T(n)$:

- (1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.
- (2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.
- (3) If $k > \log_b a$, then $T(n) = O(n^k)$.

How to use master theorem

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = aT(n/b) + n^k$$

- First make sure you can actually use the master theorem
 - for some recurrences you cannot use it, like $T(n) = T(n-1) + 1$
 - the recurrence must be of the above form
 - if cannot use master theorem
 - there are more powerful theorem's available, but they are not allowed in this course
 - use the good-old repeated substitution
- If master theorem apply, start by calculating **$\log_b a$**
- Then compare **$\log_b a$** to **k** , and decide which case it belongs to

Exercises

Consider the following functions. For each, decide which case of the master theorem applies (if any), and give the asymptotic worst-case runtime

EX 1

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 2T(n/2) + dn + e$$

$$a = 2 \quad b = 2 \quad k = 1$$

$$\log_b a = 1 = k$$

$$\text{Case 1} \quad T(n) = O(n \log n)$$

EX 2

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 9T(n/3) + n$$

$$a = 9 \quad b = 3 \quad k = 1$$

$$\log_b a = 2 > k$$

$$\text{Case 2} \quad T(n) = O(n^2)$$

EX 3

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 10T(n/3) + n$$

$$\log_b a = \log_3 10 \approx 2.1 > k = 1$$

$$\text{Case 2 : } T(n) = O(n^{\log_3 10}) \approx O(n^{2.1})$$

EX 4

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 10T(n/3) + n^4$$

$$\log_b a = \log_3 10 \approx 2.1 < k = 4$$

Case 3: $T(n) = O(n^4)$

EX 5

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = T(2n/3) + 1$$

$$a = 1 \quad b = 3/2 \quad k = 0$$

$$\log_b a = \log_{3/2} 1 = 0 = k$$

$$\text{Case 1: } T(n) = O(\log n)$$

EX 6

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = T(n/2) + T(n/3) + n$$

Cannot use the master theorem directly, but can still do some bounding

$$T(n/3) \leq T(n/2)$$

$$T(n) = T(n/2) + T(n/3) + n \leq 2T(n/2) + n = O(n \log n)$$

$$T(n) = O(n \log n)$$

Divide-and-Conquer + Master Theorem



The combination of the two gives you the ability to very quickly iterate between algorithm **design** and its runtime **analysis**.

Very **pro** way of algorithm development!

Now we know how to use the master theorem,
but ...

ONE DOES NOT SIMPLY USE A THEOREM



imgflip.com

WITHOUT PROVING IT FIRST



Prove the Master Theorem

Proof:

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ aT(n/b) + n^k, & \text{if } n > 1 \end{cases}$$

Basically, we want to find the closed form the above recurrence.
How?

Use repeated substitution!

Step 1: substitute a few times

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ aT(n/b) + n^k, & \text{if } n > 1 \end{cases}$$

$$j = 1 \quad T(n) = aT(n/b) + n^k$$

$$\begin{aligned} j = 2 \quad T(n) &= a(aT(n/b^2) + (n/b)^k) + n^k \\ &= a(aT(n/b^2) + (n^k/b^k)) + n^k \\ &= a^2 T(n/b^2) + a(n^k/b^k) + n^k \\ &= a^2 T(n/b^2) + n^k(1 + a/b^k) \end{aligned}$$

one more sub...

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ aT(n/b) + n^k, & \text{if } n > 1 \end{cases}$$

$$a^2 T(n/b^2) + n^k(1 + a/b^k)$$

$$\begin{aligned} j = 3 \quad T(n) &= a^2(aT(n/b^3) + (n/b^2)^k) + n^k(1 + a/b^k) \\ &= a^2(aT(n/b^3) + (n^k/b^{2k})) + n^k(1 + a/b^k) \\ &= a^3 T(n/b^3) + a^2(n^k/b^{2k}) + n^k(1 + a/b^k) \\ &= a^3 T(n/b^3) + n^k(1 + a/b^k + (a/b^k)^2) \end{aligned}$$

$$\text{Guess: } T(n) = a^j T(n/b^j) + n^k \sum_{i=0}^{j-1} (a/b^k)^i$$

$$T(n) = a^j T(n/b^j) + n^k \sum_{i=0}^{j-1} (a/b^k)^i$$

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ aT(n/b) + n^k, & \text{if } n > 1 \end{cases}$$

solve j for base case

plug in j

$$n/b^j = 1$$

$$n = b^j$$

$$j = \log_b n$$

$$T(n) = ca^j + n^k \sum_{i=0}^{j-1} (a/b^k)^i$$

$$= ca^{\log_b n} + n^k \sum_{i=0}^{j-1} (a/b^k)^i$$

$$= cn^{\log_b a} + n^k \sum_{i=0}^{j-1} (a/b^k)^i$$

$$a^{\log_b n} = a^{\frac{\log_a n}{\log_a b}} = a^{\log_a n \cdot \log_b a} = (a^{\log_a n})^{\log_b a} = n^{\log_b a}$$

$$T(n) = cn^{\log_b a} + n^k \sum_{i=0}^{j-1} (a/b^k)^i$$

$$j = \log_b n$$

geometric series with
common ratio a/b^k

If $a/b^k = 1$, then

- ▶ $a = b^k$, and $\log_b a = k$
- ▶ each term of the sum is equal to 1, so the entire sum is j

$$\begin{aligned} T(n) &= cn^{\log_b(b^k)} + n^k \sum_{i=0}^{j-1} 1^i \\ &= cn^k + n^k j \\ &= cn^k + n^k * \log_b(n) \\ &= O(n^k \log n) \end{aligned}$$

- (1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.
- (2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.
- (3) If $k > \log_b a$, then $T(n) = O(n^k)$.

This is exactly
Case 1 of the
master theorem!

$$T(n) = cn^{\log_b a} + n^k \sum_{i=0}^{j-1} (a/b^k)^i$$

$$\sum_{k=0}^{n-1} ar^k = a \frac{1 - r^n}{1 - r},$$

if $a/b^k \neq 1$ just do the sum of geometric series

$$\begin{aligned} T(n) &= cn^{\log_b a} + n^k \sum_{i=0}^{j-1} (a/b^k)^i \\ &= cn^{\log_b a} + n^k (1 - (a/b^k)^j) / (1 - a/b^k) \\ &= cn^{\log_b a} + n^k (1 - (a^j / b^{jk})) / (1 - a/b^k) \\ &= cn^{\log_b a} + n^k (1 - (n^{\log_b a} / n^k)) / (1 - a/b^k) \end{aligned}$$

$$T(n) = cn^{\log_b a} + n^k(1 - (n^{\log_b a}/n^k))/(1 - a/b^k)$$

This is nasty. Let:

- ▶ $n_1 = n^{\log_b a}$
- ▶ $n_2 = n^k$
- ▶ $z = 1 - a/b^k$

We get:

$$cn_1 + n_2(1 - (n_1/n_2))/z$$

- (1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.
- (2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.
- (3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$cn_1 + n_2(1 - (n_1/n_2))/z$$

Write this so that it is of the form $n_1(\dots) + n_2(\dots)$

$$\begin{aligned} & cn_1 + n_2(1 - (n_1/n_2))/z \\ &= cn_1 + n_2((n_2 - n_1)/n_2)/z \\ &= cn_1 + n_2((n_2 - n_1)/n_2z) \\ &= cn_1 + (n_2 - n_1)/z \\ &= cn_1 + (n_2/z) - (n_1/z) \\ &= n_1(c - 1/z) + n_2(1/z) \end{aligned}$$

$$\blacktriangleright n_1 = n^{\log_b a}$$

$$\blacktriangleright n_2 = n^k$$

$$\blacktriangleright z = 1 - a/b^k$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

- ▶ $n_1 = n^{\log_b a}$
- ▶ $n_2 = n^k$
- ▶ $z = 1 - a/b^k$

$$T(n) = n_1(c - 1/z) + n_2(1/z)$$

$$= n^{\log_b a}(c - 1/z) + n^k(1/z)$$

if $k < \log_b a$, the first term is higher order $T(n) = O(n^{\log_b a})$

if $k > \log_b a$, the second term is higher order $T(n) = O(n^k)$

Q.E.D.



- (1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.
- (2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.
- (3) If $k > \log_b a$, then $T(n) = O(n^k)$.

Summary

- Master theorem lets you go from the recurrence to the asymptotic bound very quickly, so you're more like a pro.
- It typically works well for divide-and-conquer algorithms
- But master theorem does not apply to all recurrences. When it does not apply, you can:
 - do some upper/lower bounding and get a potentially looser bound
 - use the substitution method
- The course notes have several interesting examples of using divide-and-conquer and the master theorem. Read them!

NEW TOPIC

Program Correctness

Program Correctness

- So far we have been studying the runtime of algorithms
- But what's more important than runtime is that the algorithms actually work **correctly!**
- How did you guarantee algorithm correctness in CSC148?
 - You used test cases. The more test cases you passed, the more confident you were about your code working correctly.
 - But you were never 100% sure that it is correct ...
- In CSC236, we will learn to **formally prove** the correctness of your program, without using test cases. **PRO-LEVEL++**



Preconditions and Postconditions

- A **precondition** of a function is what the function **requires** of its parameters so that it can guarantee correct execution.
- A **postcondition** of a function is what the function **promises**, assuming that it was called in a way that satisfies the precondition.

```
def fact_rec(a):  
    '''  
    Pre: a is an integer >= 1  
    Post: returns the factorial of a  
    '''  
    if a == 1:  
        return a # path 1 ends  
    else:  
        return a * fact_rec(a - 1) # path 2 ends
```

Definition of program correctness

Let f be a function that has a given precondition and postcondition.

Then f is correct with respect to the precondition and postcondition if:

- for every call $f(I)$ where I satisfies the precondition, $f(I)$ **terminates** in a way that **satisfies** the postcondition

Prove correctness for recursive programs

- For each **program path** from the first line to a **return** statement, we need to show that it terminates and satisfies the postcondition.
 - if there is no recursive call, analyze the code directly
 - if there are recursive calls
 - show that the precondition holds at the time of each recursive call
 - Show that the recursive call occurs on “smaller” values than the original call. (so it will terminate eventually)
 - You can then assume that the recursive call satisfies the postcondition
 - show that the postcondition is satisfied based on the assumption

E.g.,
assume
 $f(n-1)$ is
correct, then
use it to
show $f(n)$ is
correct.
This is
induction!

Examples

Prove the correctness of `fact_rec`

```
def fact_rec(a):  
    '''  
    Pre: a is an integer >= 1  
    Post: returns the factorial of a  
    '''  
    if a == 1:  
        return a # path 1 ends  
    else:  
        return a * fact_rec(a - 1) # path 2 ends
```

There are **two program paths** from the first line to a return statement.

Analyze them one by one.

We need to show:

For all integer **a** ≥ 1 , `fact_rec(a)` terminates and returns the factorial of **a**.

Analyze Path 1

Path 1 has no recursive call.

- To get to this path, **a=1**
- Path 1 returns 1, which is exactly the factorial of **a** when **a=1**
- **Correct.**

```
def fact_rec(a):  
    '''  
    Pre: a is an integer >= 1  
    Post: returns the factorial of a  
    '''  
    if a == 1:  
        return a # path 1 ends  
    else:  
        return a * fact_rec(a - 1) # path 2 ends
```

Analyze Path 2

Path 2 has recursive calls

- Check that recursive calls have preconditions satisfied
 - to get to Path 2, $a \geq 2$, so $a-1 \geq 1$, precondition satisfied
- Check that the recursive calls are on “smaller” values.
 - $a-1$ is smaller than a
 - check.

```
def fact_rec(a):  
    '''  
    Pre: a is an integer  $\geq 1$   
    Post: returns the factorial of a  
    '''  
    if a == 1:  
        return a # path 1 ends  
    else:  
        return a * fact_rec(a - 1) # path 2 ends
```

Analyze Path 2, continued

- Assume that the recursive call satisfies the postcondition
 - assume `fact_rec(a-1)` satisfies the postcondition
 - i.e., it returns **(a-1)!**, the factorial of **a-1**
- Show that postcondition is satisfied based on the assumption
 - show that `fact_rec(a)` satisfies the post condition.

```
fact_rec(a) = a * (a-1)! # by assumption
            = a! # postcondition satisfied
```

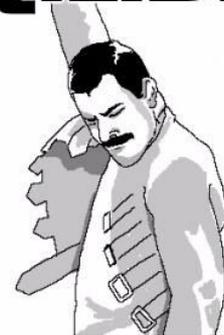
```
def fact_rec(a):
    '''
    Pre: a is an integer >= 1
    Post: returns the factorial of a
    '''
    if a == 1:
        return a # path 1 ends
    else:
        return a * fact_rec(a - 1) # path 2 ends
```

```
def fact_rec(a):  
    '''  
    Pre: a is an integer >= 1  
    Post: returns the factorial of a  
    '''  
    if a == 1:  
        return a # path 1 ends  
    else:  
        return a * fact_rec(a - 1) # path 2 ends
```

All program paths have been shown to be terminating and satisfying the postcondition.

Therefore, the above fact_rec program is **correct**.

Q.E.D.



Example 2

Prove the correctness of `gcd_rec`

```
def gcd_rec(a, b):  
    '''  
    Pre: a and b are integers >= 1, and a >= b  
    Post: returns the greatest common divisor of a and b  
    '''  
    if a == 1 or b == 1:  
        return 1                # path 1 ends  
    elif a % b == 0:  
        return b                # path 2 ends  
    else:  
        return gcd_rec(b, a % b) # path 3 ends
```

Need to show

For all integers $a, b \geq 1$ such that $a \geq b$, `gcd_rec(a, b)` terminates and returns the greatest common divisor (GCD) of a and b .

- There are three program paths
- Analyze them separately.

```
def gcd_rec(a, b):  
    '''  
    Pre: a and b are integers >= 1, and a >= b  
    Post: returns the greatest common divisor of a and b  
    '''  
    if a == 1 or b == 1:  
        return 1                # path 1 ends  
    elif a % b == 0:  
        return b                # path 2 ends  
    else:  
        return gcd_rec(b, a % b) # path 3 ends
```

Analyze Path 1 & 2

Path 1:

- to get into this path, $a == 1$ or $b == 1$
- the GCD of 1 with other number must be 1, so “return 1” is correct.

Path 2:

- to get into this path, $a \% b = 0$, i.e., b is a divisor of a .
- b is the largest divisor of itself, no larger divisor is possible.
- so “return b ” is correct

```
def gcd_rec(a, b):  
    '''  
    Pre: a and b are integers >= 1, and a >= b  
    Post: returns the greatest common divisor of a and b  
    '''  
    if a == 1 or b == 1:  
        return 1                # path 1 ends  
    elif a % b == 0:  
        return b                # path 2 ends  
    else:  
        return gcd_rec(b, a % b) # path 3 ends
```

Analyze Path 3

- Check that the recursive call satisfies the precondition
 - $b \geq 1$, from the function precondition
 - $a \% b \geq 1$, because ...
 - if $a \% b == 0$, Path 2 would have applied
 - $b \geq a \% b$, because $a \% b \leq b - 1$ by def
- Check that the recursive call is on smaller value
 - ARG1: $b \leq a$, from the function precondition
 - ARG2: $a \% b < b$, because $a \% b \leq b - 1$ by def
 - So overall the input is smaller value

```
def gcd_rec(a, b):  
    '''  
    Pre: a and b are integers  $\geq 1$ , and  $a \geq b$   
    Post: returns the greatest common divisor of a and b  
    '''  
  
    if a == 1 or b == 1:  
        return 1 # path 1 ends  
    elif a % b == 0:  
        return b # path 2 ends  
    else:  
        return gcd_rec(b, a % b) # path 3 ends
```

Analyze Path 3 continued

- Assume the recursive call returns the correct value **GCD(b, a % b)**
- Use the math identity
 - $\text{GCD}(a, b) = \text{GCD}(b, a \% b)$
 - “Euclidean method”
- $\text{GCD}(a, b)$ is exactly the correct return value of the `gcd_rec(a, b)`
- So postcondition satisfied
- Path 3 proven correct.

gcd_rec is correct

```
def gcd_rec(a, b):  
    '''  
    Pre: a and b are integers >= 1, and a >= b  
    Post: returns the greatest common divisor of a and b  
    '''  
  
    if a == 1 or b == 1:  
        return 1                # path 1 ends  
    elif a % b == 0:  
        return b                # path 2 ends  
    else:  
        return gcd_rec(b, a % b) # path 3 ends
```

Q.E.D.



Next week

- Correctness of programs with **loops**