CSC236 Week 5

Larry Zhang

Announcements

- PS1: Make sure you adopt the feedback and make an improvement.
- PS3 due Friday
- No lecture or tutorial next week (reading week)
  - But there IS a problem set (PS4) due on Friday
  - Office hours: Tue 3-5, Wed 5-7, Fri 3-5
- Week after reading week: We will have **Test 1** in class
- Go to the test of the section you are in on ACORN/ROSI

Test 1

- First hour of the lecture (50 minutes), don’t be late.
- No aid allowed
- There will be **three** questions
  - Proof using induction (simple/complete/structural)
  - Proof for big-Oh/Omega/Theta
  - Developing recurrence, finding closed-form
  - Maybe a bonus question

Prepare for the test

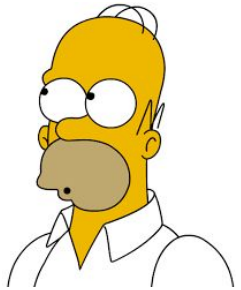
- There are a bunch of past Test 1’s posted on the course web page.
- Review lecture examples, tutorials and problem sets.
- The course notes also have a number of highly relevant exercises after each chapter.
- Come to office hours.

Today: Lectorial
More exercises of developing recurrence and find closed form

Recap questions

- We learned two types of functions according to how they are defined
  - One is closed-form function, the other is ...
  - **Recursively defined function**
- We want to find the closed form of a recursively defined function, why?
- We learned a method to find the closed form, it’s called…
  - **Repeated substitution**

Recap: The steps of repeated substitution



Step 1: **Substitute** a few time to find a pattern

Step 2: **Guess** the recurrence formula after k substitutions (in terms of k and n)

For each base case:

Step 3: **solve** for k

Step 4: **Plug** k back into the formula (from Step 2) to find a **potential** closed form. (“Potential” because it might be wrong)

Step 5: **Prove** the potential closed form is equivalent to the recursive definition using induction.

We did these two examples

$$T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1) + 1, & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2T(n-2), & \text{if } n > 2 \end{cases}$$

Next example

For simplicity, assume all **n**'s that we care about are powers of 2, so dividing by 2 always gives integers.

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

Something interesting: the recursive relation is between n and n/2.

Try it yourself first!

Step 1: Substitute for a few times

$k = 1$ $\quad T(n) = 2T(n/2) + n/2$

$k = 2$ $\quad T(n) = 2(2T(n/4) + n/4) + n/2$

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

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

$k = 3$ $\quad T(n) = 2^2(2T(n/8) + n/8) + n$

$= 2^3 T(n/8) + n/2 + n$

$= 2^3 T(n/8) + 3n/2$

Step 2: Guess the formula after k substitutions

$$
\begin{aligned}
k = 1 \qquad & T(n) = 2T(n/2) + n/2 \\
k = 2 \qquad & T(n) = 2(2T(n/4) + n/4) + n/2 \\
& \quad = 2^2 T(n/4) + n/2 + n/2 \\
& \quad = 2^2 T(n/4) + n \\
k = 3 \qquad & T(n) = 2^2(2T(n/8) + n/8) + n \\
& \quad = 2^3 T(n/8) + n/2 + n \\
& \quad = 2^3 T(n/8) + 3n/2
\end{aligned}
$$

$$T(n) = 2^k T(n/2^k) + kn/2$$

Step 3: Set ***k*** so that we get the base case

$$T(n) = 2^k T(n/2^k) + kn/2$$

Let $n/2^k = 1$

$$2^k = n$$

$$k = \log_2 n$$

Step 4: Plug solved k back into the formula

$$T(n) = 2^k T(n/2^k) + kn/2$$

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

$$k = \log_2 n$$

$$T(n) = 2^{\log_2 n} T(1) + (\log_2 n)n/2$$

$$= n + (\log_2 n)n/2$$

Potential closed form

Step 5: Prove potential closed form

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

Is equivalent to

$$T(n) = n + (\log_2 n)n/2$$

Try it yourself!

In the test, we may ask you to skip this step, just to save time. Read the question carefully!

What type of induction to use here?

Complete induction!

Next Example

Sometimes you’re not given the recursive definition directly, so you need to **develop the recursive definition** first

Example

Give a recursive definition of

the number of 2-element subsets of n elements.

(Pretend that you never knew “n choose 2”, and let’s develop it from first principles).

For example, when **n=4**, we have **4** elements, $\mathbf{e_1, e_2, e_3, e_4}$,
the 2-element subsets are $(\mathbf{e_1,e_2})$, $(\mathbf{e_1,e_3})$, $(\mathbf{e_1,e_4})$, $(\mathbf{e_2,e_3})$, $(\mathbf{e_2,e_4})$, $(\mathbf{e_3,e_4})$
There are **6** of them.

Strategy

1. Find the **base case(s)** that can be evaluated directly
2. Find a way to **break the larger problem** to smaller problems, so that you can define **f(n)** in terms of **f(m) for some m < n**.

Give a recursive definition of

f(n): The number of 2-element subsets of n elements.

1. **Base case:**

$$n = 0$$

$$f(0) = 0$$

Give a recursive definition of

f(n): The number of 2-element subsets of n elements.

2. Break the larger problem to smaller ones:

For the set of n elements, how many 2-element subsets does it have?

Our set looks like: $S = \{e_1, e_2, \ldots, e_n\}$

Break all subsets of ***S*** into two parts

- the subsets that contain e_n
- the subsets that do NOT contain e_n

$$S = \{e_1, e_2, \ldots, e_n\}$$

The 2-element subsets that contain e_n

How many?

Two elements …

- One has to be e_n
- The other can be $e_1, e_2, \ldots, e_{n-1}$

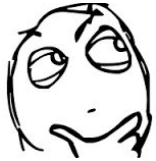
So there are n-1 of such subsets!

The number of 2-element subsets of n elements.

$$S = \{e_1, e_2, \ldots, e_n\}$$

The 2-element subsets that do NOT contain e_n

How many?



They are subsets of $S' = \{e_1, e_2, \ldots, e_{n-1}\}$

The number of 2-element subsets of **n-1** elements

It's f(n-1) !

$$S = \{e_1, e_2, \ldots, e_n\}$$

Sum up after breaking the larger problem into smaller problems

The number of 2-element subsets of n elements

- number of subsets with e_n: **n-1**
- number of subsets without e_n: **f(n-1)**

$$f(n) = n - 1 + f(n-1)$$

Combining with the base case, the final developed recurrence:

f(n): the number of 2-element subsets of n elements.

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

Home exercise: find its closed form using the substitution method.
You know what the result should be.

OK, that’s a good amount of math, now let’s bring in the computer stuff

Analyse the runtime of the following recursive program

```
def factorial(n):
1   if n == 1:
2      return 1
3   else:
4      return n * factorial(n-1)
```

What is the worst-case runtime **T(n)** of “`factorial(n)`”?

Strategy

1. Look at the base case, get its runtime
2. Look at the recursive calls, get the runtime in terms of the runtime of the recursive calls.

```
def factorial(n):
1   if n == 1:
2      return 1
3   else:
4      return n * factorial(n-1)
```

Base case

```
def factorial(n):
1   if n == 1:
2      return 1
3   else:
4      return n * factorial(n-1)
```

T(1) ?

The amount of work is constant

T(1) = c

where c is some constant

The recursive calls

```
def factorial(n):
1   if n == 1:
2      return 1
3   else:
4      return n * factorial(n-1)
```

T(n) ?

The amount of work include

- time spent on the **factorial(n-1)**: T(n-1)
- some constant amount of work

T(n) = T(n-1) + d, where **d** is some constant

So altogether ...

```
def factorial(n):
1   if n == 1:
2       return 1
3   else:
4       return n * factorial(n-1)
```

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ T(n-1) + d, & \text{if } n > 1 \end{cases}$$

This is a recursively defined function, we want to find the closed form of this function, and we know how to do it.

Apply substitution method

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ T(n-1) + d, & \text{if } n > 1 \end{cases}$$

$k = 1 \qquad T(n) = T(n-1) + d$

$k = 2 \qquad T(n) = (T(n-2) + d) + d \ = T(n-2) + 2d$

$k = 3 \qquad T(n) = (T(n-3) + d) + 2d = T(n-3) + 3d$

Guess: $T(n) = T(n-k) + kd$

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ T(n-1) + d, & \text{if } n > 1 \end{cases}$$

$$T(n) = T(n-k) + kd$$

Solve **n - k = 1**, get **k = n - 1**
Plug in **k = n - 1**

$$T(n) = T(1) + (n-1)d$$

$$= c + (n-1)d$$

$$= dn + c - d$$

Potential closed form which can be proven to be correct. Do the proof yourself!

So far, we can say

```
def factorial(n):
1   if n == 1:
2       return 1
3   else:
4       return n * factorial(n-1)
```

This recursive algorithm has worst-case runtime

$$T(n) = dn + c - d$$

Wanna be more like a pro, how?

Use **asymptotic notations**!

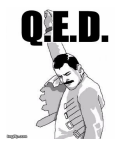
Prove $T(n) = dn + c - d \in \mathcal{O}(n)$

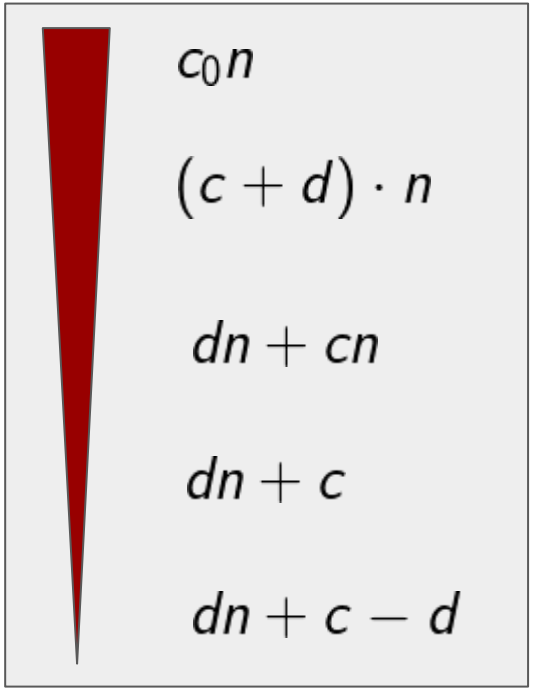
Proof:

Pick $n_0 = 1, c_0 = c + d$, then $\forall n \geq n_0$

$$
\begin{aligned}
& dn + c - d \\
\leq\ & dn + c \\
\leq\ & dn + cn \\
=\ & (c + d)n \\
=\ & c_0 n
\end{aligned}
$$

By definition of $\mathcal{O}$, $T(n) \in \mathcal{O}(n)$





Now, finally, you can say ...

```
def factorial(n):
1   if n == 1:
2      return 1
3   else:
4      return n * factorial(n-1)
```

This is an O(n) algorithm.

```
def factorial(n):
1   if n == 1:
2      return 1
3   else:
4      return n * factorial(n-1)
```

Exercise for home:
Show that this is a **Θ(n)** algorithm.

Next Example

First, clarify a notation

The “dot dot notation”:

`[a..b]` means all numbers from a to b, **inclusive**.

e.g., `[0..5]` include 0, 1, 2, 3, 4, 5

`[0..0]` is just 0

`[0..-1]` is an empty list

This is different from the Python colon (:) operator.

```
def xxx_xxxxx(A, x):
    '''
    Pre: A is a non-empty sorted list in non-decreasing order
    Post: Returns True if and only if x is in A
    '''
1   if len(A) == 1:
2      return A[0] == x
3   else:
4      m = len(A) // 2    # integer division, rounds down
5      if x <= A[m-1]:
6         return xxx_xxxxx(A[0..m-1], x)
7      else:
8         return xxx_xxxxx(A[m..len(A)-1], x)
```

```
def bin_search(A, x):
    '''
    Pre: A is a non-empty sorted list in non-decreasing order
    Post: Returns True if and only if x is in A
    '''
1   if len(A) == 1:
2      return A[0] == x
3   else:
4      m = len(A) // 2    # integer division, rounds down
5      if x <= A[m-1]:
6         return bin_search(A[0..m-1], x)
7      else:
8         return bin_search(A[m..len(A)-1], x)
```

Binary Search

Analyse the runtime of Binary Search

```
def bin_search(A, x):
    '''
    Pre: A is a non-empty sorted list in non-decreasing order
    Post: Returns True if and only if x is in A
    '''
1   if len(A) == 1:
2      return A[0] == x
3   else:
4      m = len(A) // 2    # integer division, rounds down
5      if x <= A[m-1]:
6         return bin_search(A[0..m-1], x)
7      else:
8         return bin_search(A[m..len(A)-1], x)
```

Base case: constant time

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

Recursive call: input size becomes n/2

Plus some constant work (assuming list slicing takes constant time)

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

Find its closed form

$$k = 1 \qquad T(n) = T(n/2) + d$$

$$k = 2 \qquad T(n) = (T(n/4) + d) + d \quad = T(n/4) + 2d$$

$$k = 3 \qquad T(n) = (T(n/8) + d) + 2d \quad = T(n/8) + 3d$$

Guess: $T(n) = T(n/2^k) + kd$

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

$$T(n) = T(n/2^k) + kd$$

Solve $n/2^k = 1$, get $k = \log_2 n$

Plug in k $\quad T(n) = T(1) + d \cdot \log_2 n = c + d \cdot \log_2 n$

You can prove this potential closed form is correct.

Then you can prove
T(n) = c + d log n is in **O(log n)**
Therefore you can say ...

A caveat

```
def bin_search(A, x):
    '''
    Pre: A is a non-empty sorted list in non-decreasing order
    Post: Returns True if and only if x is in A
    '''
1   if len(A) == 1:
2      return A[0] == x
3   else:
4      m = len(A) // 2    # integer division, rounds down
5      if x <= A[m-1]:
6         return bin_search(A[0..m-1], x)
7      else:
8         return bin_search(A[m..len(A)-1], x)
```

- We assumed that list slicing (like A[0..m-1]) takes constant time,
- but in reality it takes longer (O(n) because it copies half of the list).
- This would make the overall runtime different. (Try it!)
- But we assumed it anyway because we knew the slicing part can be in constant time if done cleverly.

Plus some constant work (assuming list slicing takes constant time)

Better implemented binary search

```
def bin_search(A, x, first, last):
    '''
    Pre: A is a non-empty sorted list in non-decreasing order
    Post: Returns True if and only if x is in A
    '''
1   if first == last:
2      return A[first] == x
3   else:
4      m = (first + last + 1) // 2    # integer division, rounds down
5      if x <= A[m-1]:
6         return bin_search(A, x, first, m-1)
7      else:
8         return bin_search(A, x, m, last)
```

Idea: Manipulating indices on one list rather than making copies of the list.

After this optimization, the runtime of the algorithm is really O(log n).

Takeaway

This mathematical analysis of the runtime is a very powerful tool for algorithm design.

- You know exactly what **impact** each modification has on the overall runtime of the algorithm.
    - Like how a master chef knows the impact of each ingredient.
- You can **predict**, before you code it, what is worth optimizing, and what is not.
- You can be super confident about it because everything is **proven**!
- Rather than just reusing existing algorithms created by other people, you can invent your own delicious dishes (efficient algorithms).

Tutorial this week

- Exercises for the substitution method