CS240
Fall 2014

Mike Lam, Professor

Recurrences

Announcement

- A solution to PA2 has been posted on Canvas
- Most of the functions can be re-used between PA2 and PA3
  - As long as they are written in terms independent of the underlying representation

BAD:

```
def is_subset(self, other):
    for i in range(self._len):
        found = False
        for j in range(other._len):
            if self._a[i] == other._a[j]:
                found = True
        if not found:
            return False
    return True
```

GOOD:

```
def is_subset(self, other):
    for elem in self:
        if elem not in other:
            return False
    return True
```

- You may use the posted implementations in PA3, but you should give credit in your documentation if you choose to do so

Announcement

- Special session on Friday (10/17)
- Combined meeting of CS 240/280
  - CS 280: Topics: Competitive programming
  - Compete in worldwide ACM competitions
  - Meets weekly for practices
- Meet in ISAT 243 at normal time
  - Short lecture on binary search algorithms
  - Join CS 280 students in ISAT 143
  - Work on problems in teams

Recurrences

- Recurrence: an equation that expresses the value of a function in terms of its value at another point
- Similarity to recursion: a problem solution expressed in terms of solutions to subproblems

$$T(1)=1$$
$$T(n)=1+2T(n-1)$$

$$T(1)=1$$
$$T(n)=n+T\left(\frac{n}{2}\right)$$

Finding Recurrences

- Function in terms of running time: T(n)
- Find the base case
  - Probably T(0) or T(1)
  - How many operations? (usually a constant; often 0 or 1)
- Find the recursive case
  - How many operations?
  - How many recursive calls?
    - Usually once (single recursion) or twice (binary recursion)
    - Could be more
  - How does n change in new calls to T()?
    - Most common: T(n-1) or T(n/2)

Solving Recurrences

- Can be difficult; not always possible!
- One method: Backward substitution
  - Substitute for *n*
    - Substitute into itself
    - Repeat as necessary
  - Identify pattern
    - Express using new term *i*
  - Substitute for *i*
    - In terms of *n*
    - Eliminate recursion
  - Clean up
    - Find closed form (evaluable in finite # of operations)

Example

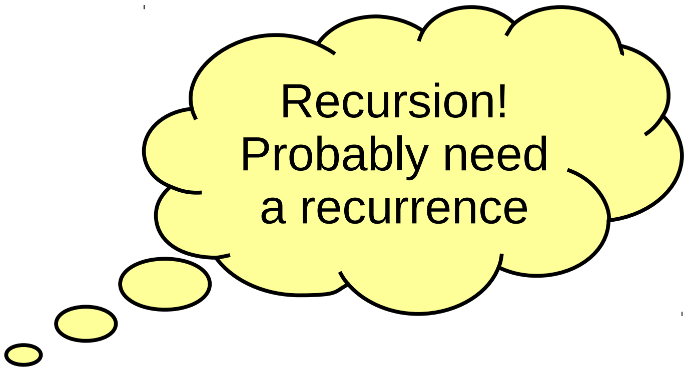
- What is the running time of "foo"?

```
def foo(x):
    if x <= 1:
        return 1
    else:
        return x * foo(x-1)
```

Example

- What is the running time of "foo"?

```
def foo(x):
    if x <= 1:
        return 1
    else:
        return x * foo(x-1)
```



Example

- What is the running time of "foo"?

```
def foo(x):
    if x <= 1:
        return 1
    else:
        return x * foo(x-1)
```

Find base case (initial condition)
and recursive case (inductive condition)

n = x

Example

- What is the running time of "foo"?

```
def foo(x):
    if x <= 1:
        return 1
    else:
        return x * foo(x-1)
```

Find base case (initial condition)
and recursive case (inductive condition)

n = x

$$T(0) = 0$$
$$T(n) = 1 + T(n-1)$$

Example

- What is the running time of "foo"?

```
def bar(x):
    if len(x) == 0:
        return 0
    else:
        return x[0] + bar(x[1:])
```

Example

- What is the running time of "foo"?

```
def bar(x):
    if len(x) == 0:
        return 0
    else:
        return x[0] + bar(x[1:])
```

Find base case (initial condition)
and recursive case (inductive condition)

n = len(x)

Example

- What is the running time of "foo"?

```
def bar(x):
    if len(x) == 0:
        return 0
    else:
        return x[0] + bar(x[1:])
```

Find base case (initial condition)
and recursive case (inductive condition)

n = len(x)

$$T(0) = 0$$
$$T(n) = 1 + T(n-1)$$

Example

- Different functions; same recurrence!

Example

- Let's try some values:

$$T(0) = 0$$
$$T(n) = 1 + T(n-1)$$

Example

- Let's try some values:

$$T(0) = 0$$
$$T(n) = 1 + T(n-1)$$

What's the pattern?

$$T(0) = 0$$
$$T(1) = 1 + T(0) = 1 + 0 = 1$$
$$T(2) = 1 + T(1) = 1 + 1 = 2$$
$$T(3) = 1 + T(2) = 1 + 2 = 3$$

Example

- Let's try some values:

$$T(0)=0$$
$$T(n)=1+T(n-1)$$

$$T(0)=0$$
$$T(1)=1+T(0)=1+0=1$$
$$T(2)=1+T(1)=1+1=2$$
$$T(3)=1+T(2)=1+2=3$$

What's the pattern?

$T(n) = n$

Example

- We think we've "solved" the recurrence
  - It *looks* right, anyway
- That's not a very formal argument
- Let's make this more rigorous

Solving Recurrences

- Method: Backward substitution
  - Substitute for *n*
    - Substitute into itself
    - Repeat as necessary
  - Identify pattern
    - Express using new term *i*
  - Substitute for *i*
    - In terms of *n*
    - Eliminate recursion
  - Clean up
    - Find closed form

Can be useful to rewrite as $T(x) = f(x)$

Example

- Substitute for n:

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

Aha! A pattern!

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

- Then identify the pattern:

$$T(n) = i + T(n-i)$$

We need to get rid of the recursive term
So we want T(0) here; what should "i" be?
Solve "$n - i = 0$" for i

Example

- Substitute for i: i = n

$$T(n)=i+T(n-i)$$

- Then clean up:

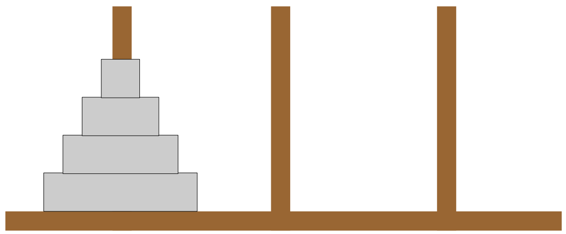
$$T(n)=(n)+T(n-(n))$$
$$T(n)=n+T(0)=n+0$$
$$T(n)=n$$

This matches our previous guess!

Example

- What is the running time?

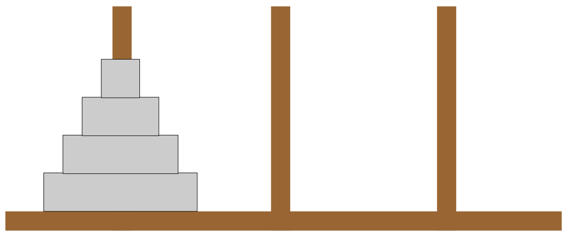
```
def hanoi(n, src, dst, tmp):
    if n == 1:
        print("move from " + str(src) +
              " to " + str(dst))
    else:
        hanoi(n-1, src, tmp, dst)
        hanoi(  1, src, dst, tmp)
        hanoi(n-1, tmp, dst, src)
```



Example

- What is the running time?

```
def hanoi(n, src, dst, tmp):
    if n == 1:
        print("move from " + str(src) +
              " to " + str(dst))
    else:
        hanoi(n-1, src, tmp, dst)
        hanoi(  1, src, dst, tmp)
        hanoi(n-1, tmp, dst, src)
```



$$T(1) = 1$$
$$T(n) = T(n-1)+T(1)+T(n-1) = 1+2T(n-1)$$

Example

- Try some values

$$
\begin{aligned}
T(1) &= 1 \\
T(2) &= 1+2T(1) = 1+2(1) = 3 \\
T(3) &= 1+2T(2) = 1+2(3) = 7 \\
T(4) &= 1+2T(3) = 1+2(7) = 15 \\
T(5) &= 1+2T(4) = 1+2(15) = 31
\end{aligned}
$$

Example

- Substitute for n and identify pattern

$$
\begin{aligned}
T(n) &= 1+2T(n-1) \\
T(n) &= 1+2(1+2T((n-1)-1)) = 1+2+4T(n-2) \\
T(n) &= 1+2+4(1+2T((n-1)-2)) = 1+2+4+8T(n-3) \\
T(n) &= 1+2+4+\ldots+2^{i}T(n-i)
\end{aligned}
$$

$$
T(n) = \sum_{j=0}^{i-1} 2^{j} + 2^{i}T(n-i)
$$

Solve "$n - i = 1$" to find value for i

Example

- Substitute for i and clean up (i = n-1)

$$T(n) = \sum_{j=0}^{i-1} 2^j + 2^i T(n-i)$$

$$T(n) = \sum_{j=0}^{(n-1)-1} 2^j + 2^{(n-1)} T(n-(n-1))$$

$$T(n) = \sum_{j=0}^{n-2} 2^j + 2^{(n-1)} T(0)$$

$$T(n) = \sum_{j=0}^{n-2} 2^j + 2^{(n-1)}$$

$$T(n) = \sum_{j=0}^{n-1} 2^j = 2^n - 1$$

Exercise

- Solve the recurrence:

$$T(0) = 1$$
$$T(n) = 2T(n-1)$$

Exercise

- Solve the recurrence:

  (assume $n = 2^k$)

$$T(1) = 0$$

$$T(n) = 1 + T\left(\frac{n}{2}\right)$$

Master's Theorem

- Generic recurrence solution patterns for divide & conquer algorithms

$$\textit{Pattern:}\quad T(n) = aT\left(\frac{n}{b}\right) + O\left(n^k\right)$$

$$\textit{If } a > b^k, \textit{ then } T(n) \in O\left(n^{\log_b a}\right)$$

$$\textit{If } a = b^k, \textit{ then } T(n) \in O\left(n^k \log n\right)$$

$$\textit{If } a < b^k, \textit{ then } T(n) \in O\left(n^k\right)$$

Recurrences in CS240

- Finding recurrences
  - Provide the recurrence in terms of T(n)
- Solving recurrences
  - Provide the closed-form solution in terms of *n*
  - Provide some indication of how you found it
    - Back substitution
    - Recognized pattern
  - Verify that it matches some actual values of T(n)

More Exercises

- See "Concise Notes" Chapter 14
  - Check your answers w/ Wolfram Alpha
- Watch for an upcoming homework
- Possible external review session

- Useful math facts:

$$\sum_{0}^{n} b^n = b^{n+1} - 1$$

$$n = b^k \Rightarrow k = \log_b n$$

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