L13: Recursion

Alex Steiger
CompSci 201: Spring 2024
2/26/2024

Person in CS: Ellen Ochoa

• BS physics ('75), PhD Elec. Eng. ('85)
• Starting working on software for optical recognition systems (computer vision)
• Applied to be an astronaut in...
  • '85...rejected
  • '87...rejected
  • '90...accepted!!!
• Worked on flight software, computer hardware, and robotics
• First Hispanic woman in space '93
• Director of NASA Johnson Space Flight Center (Houston) '13

Announcements, Coming up

• Today, Monday 2/26
  • Nothing due

• Wednesday 2/28
  • APT 5 (linked list problems) due

• Next Monday 3/4
  • Project P3: DNA (linked list project) due
Today’s outline

- Introducing Recursion
  - Counting ListNodes
  - Reversing a LinkedList

- Live Coding w/ Recursion
  - (Time permitting)

Toward Recursion by counting nodes: Iterative vs. Recursive

- Standard linked list iteration
  - Advance local pointer, do something at each node

```java
public int countIter(ListNode list) {
    int total = 0;
    while (list != null) {
        list = list.next;
        total++;
    }
    return total;
}
```

- Recursion?
  - Base Case?
  - General case?

```java
public int size(ListNode list) {
    if (list == null) return 0;
    return 1 + size(list.next);
}
```

Key ideas in recursion

1. Base case: Solve for answer when instance is “small”
2. General case:
   1. Get answer on smaller instance(s) of the same problem using recursive call(s)
   2. Do something with the result of the recursive call(s) and then return

- Note: Methods/calls stacked, like all methods
Thinking recursively

1. When is the input small enough that the answer is trivial? Base case.
2. Otherwise, suppose a magical fairy (the Recursion Fairy!) could solve the exact same problem on a smaller input.
3. Could you solve the larger problem given what the fairy told you?

The call stack: How recursion works on a machine

- Each method call gets its own call frame (local variables, etc.)
- **Eager evaluation:** Invoking method does not resume until invoked method returns.

Eager evaluation and substitution

- Return value will be substituted into the expression calling the method.
Eager evaluation and substitution

- Return value will be substituted into the expression calling the method.
Counting Nodes

```
public int size(ListNode list) {
    if (list == null) return 0;
    return 1 + size(list.next);
}
```

Recursive runtime

- Concept is the same: Count the number of constant time operations...across all recursive calls!

- **Ensure each recursive call gets closer to the base case**, else code may run forever.

- Moves one node toward the base case at each step.
- List of N nodes, makes O(N) total recursive calls, each takes O(1) time
- Overall O(N) runtime complexity.

Recall the reverse problem

- How do we reverse nodes in a linked list
  - Go from A->B->C to C->B->A
  - Typical interview style question
    - [https://leetcode.com/problems/reverse-linked-list/](https://leetcode.com/problems/reverse-linked-list/)
    - [https://www.hackerrank.com/challenges/reverse-a-linked-list](https://www.hackerrank.com/challenges/reverse-a-linked-list)
Base case, words and code

• Base case: When is there nothing to do?
  • A list with 0 or 1 nodes is its own reverse

```java
3  public static ListNode reverse(ListNode list) {
4    if (list == null || list.next == null) {
5      return list;
6    }
```

Recursive step in words

• Suppose the Recursion Fairy (a recursive call) reverses the list after the first node.

• How to use? Just put the first node at the end!

• Restated: The reverse of a list is the reverse of all but the first element, with the first element added to the end.

Recursive step in pictures

- Returned by recursive call on list.next
- Make reversedLast point to what list points to
- Return reversedFirst
Recursive step in code

```java
list
 A    B    C

7    ListNode reversedLast = list.next;
8    ListNode reversedFirst = reverse(list.next);

list
 A    C    B

Note that list.next still refers to reversedLast
```

Recursive step in code (continued)

```java
list
 A    C    B

9    reversedLast = list;
10   list.next = null;
11   return reversedFirst;

C    B    A

Make B point to A
Make A point to null
Return overall reversed list
```

Putting it all together

```java
3    public static ListNode reverse(ListNode list) {
4        if (list == null || list.next == null) {
5            return list;
6        }
7        ListNode reversedLast = list.next;
8        ListNode reversedFirst = reverse(list.next);
9        reversedLast.next = list;
10       list.next = null;
11       return reversedFirst;
12    }
```

Revisiting the call stack: How it really works

Back to the case we considered first
Consider the `rec` method. If the input list is `['A', 'B', 'C']`, what will be returned by `rec(list)`?

```java
public static ListNode rec(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode after = rec(list.next);
    if (list.info <= after.info) {
        list.next = after;
        return list;
    }
    return after;
}
```

Answer: `['A', 'B', 'C']`

If the input list is `['C', 'B', 'A']`, what will be returned by `rec(list)`?

```java
public static ListNode rec(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode after = rec(list.next);
    if (list.info <= after.info) {
        list.next = after;
        return list;
    }
    return after;
}
```

Answer: `['A']`

For an input list with N nodes, the best characterization of the runtime complexity of `rec(list)` is...

```java
public static ListNode rec(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode after = rec(list.next);
    if (list.info <= after.info) {
        list.next = after;
        return list;
    }
    return after;
}
```

Answer: `O(N)`
Consider the mystery method. Note that it is the same as rec except for lines 24-29. If the input list is ['C', 'B', 'A'], what will be returned by mystery(list)?

```java
15 public static ListNode mystery(ListNode list) {
16     if (list == null || list.next == null) {
17         return list;
18     }
19     ListNode after = mystery(list.next);
20     if (list.info == after.info) {
21         list.next = after;
22         return list;
23     }
24     ListNode current = after;
25     while (current.next != null && list.info > current.next.info) {
26         current = current.next;
27     }
28     list.next = current.next;
29     current.next = list;
30     return after;
31 }
```

Answer: ['A', 'B', 'C']

Same mystery method. For an input list with N nodes, the best characterization of the runtime complexity of mystery(list) is...

```java
15 public static ListNode mystery(ListNode list) {
16     if (list == null || list.next == null) {
17         return list;
18     }
19     ListNode after = mystery(list.next);
20     if (list.info == after.info) {
21         list.next = after;
22         return list;
23     }
24     ListNode current = after;
25     while (current.next != null && list.info > current.next.info) {
26         current = current.next;
27     }
28     list.next = current.next;
29     current.next = list;
30     return after;
31 }
```

Answer: O(N²)