Recursive Algorithms

Part 1

Roadmap

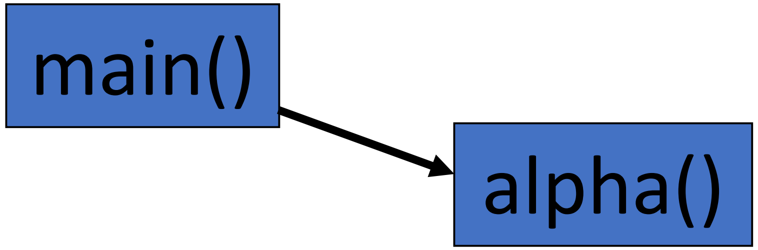
- Iteration and recursion
- Recursion traces
- Stacks and recursion
- Types of recursion
- Rules for designing recursive algorithms

Iteration and recursion

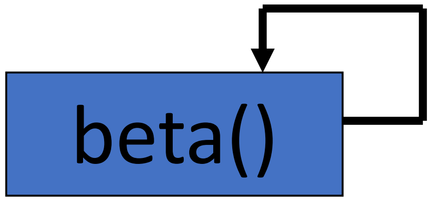
- For tasks that must be repeated, up until now we have considered iterative approaches only
- Recap: iteration allows some sequence of steps (or block of code) to be executed repeatedly, e.g. using a for loop or a while loop
- Recursion is another technique which may be applied to complete tasks which are repetitive in nature

Recursion

- "Normally", procedures (or methods) call other procedures
  - E.g. the main() procedure calls the alpha() procedure



- A recursive procedure is one which calls itself
  - E.g. the beta() procedure contains a call to beta()



Simple recursion program

- You can see that the count method calls itself
- This program would output the values 0 1 2 to the console if run

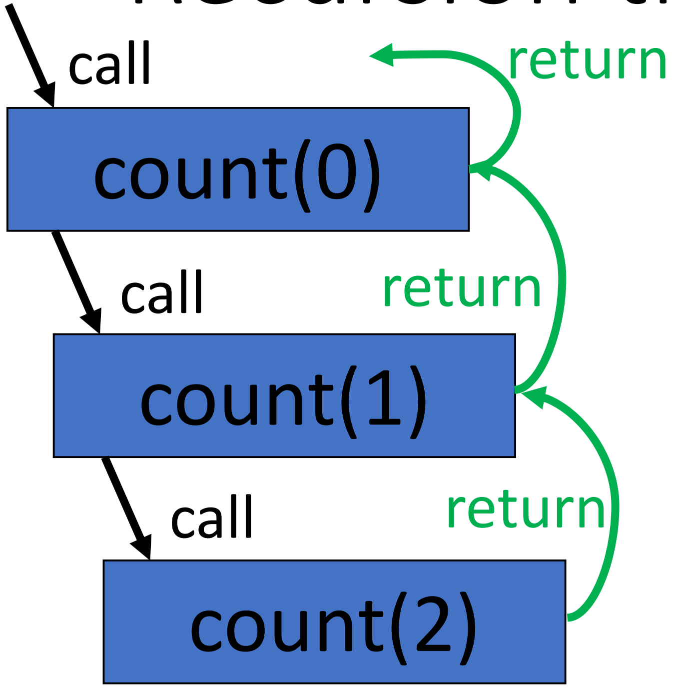
Java

```java
void main() {
  count(0);
}

void count(int index) {
  print(index);
  if(index<2) {
    count(index+1);
  }
}
```

Python

```python
def count(index):
    print(index)
    if index < 2:
        count(index + 1)


count(0)  # outputs 0 1 2
```

Recursion trace for the call count(0)



Java

```
void main() {
  count(0);
}

void count(int index) {
  print(index);
  if(index<2) {
    count(index+1);
  }
}
```

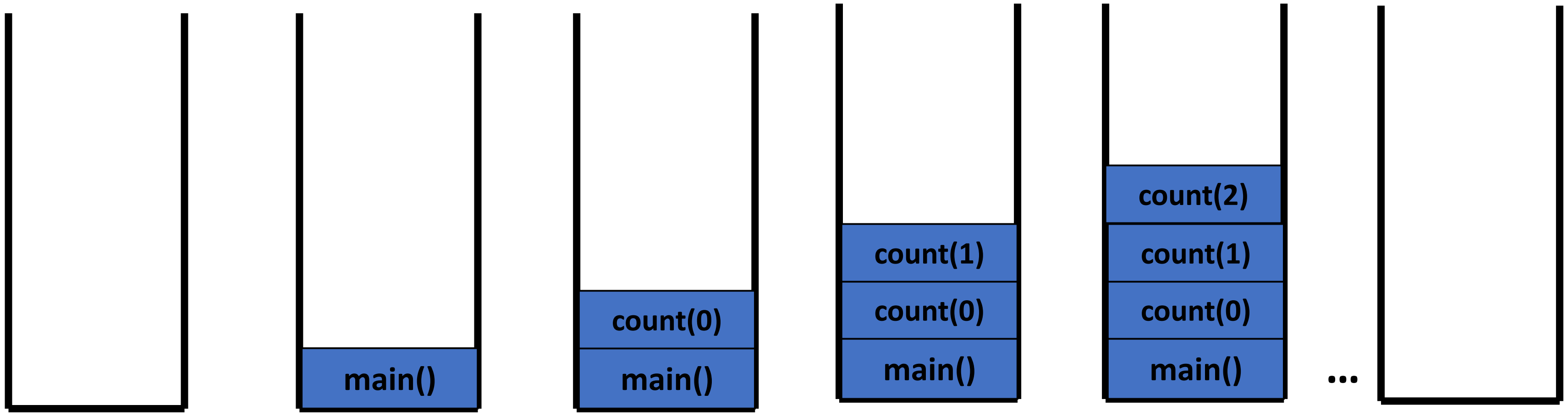
Python

```python
def count(index):
    print(index)
    if index < 2:
        count(index + 1)


count(0)   # outputs 0 1 2
```

Stacks

- A program stack basically operates like a container of trays in a cafeteria. It has only two operations:
- Push: push something onto the stack.
- Pop: pop something off the top of the stack.
- When the method returns or exits, the method’s activation frame is popped off the stack.
- Each time a method is invoked, the method’s activation frame (record) is placed on top of the program stack.

Stacks and recursion



Time: 0

Empty Stack

Time 1:

Push: main()

Time 2:

Push: count(0)

Time 3:

Push: count(1)

Time 4:

Push: count(2)

Inside count(0):
print (index) → 0
if (index < 2)
    count(index+1)

Inside count(1)
print (index) → 1
if (index < 2)
    count(index+1)

Inside count(2)
print (index) → 2
if (index < 2)
    count(index+1)

This condition now fails!

Hence, recursion stops, and method activation frames are popped off the stack.

Why use recursion?

- With the technique of recursion, a problem may be solved by solving smaller instances of the same problem
- Some problems are more easily solved by using a recursive approach
- E.g.
  - Traversing through directories of a file system
  - Traversing through a tree of search results
  - Some sorting algorithms are recursive in nature
- Recursion often leads to cleaner and more concise code which is easier to understand

Recursion vs. iteration

- Note: any set of tasks which may be accomplished using a recursive procedure may also be accomplished by using an iterative procedure
- Recursion is "expensive". The expense of recursion lies in the fact that we have multiple activation frames and the fact that there is overhead involved with calling a method.
- If both of the above statements are true, why would we ever use recursion?
- In many cases, the extra "expense" of recursion is far outweighed by a simpler, clearer algorithm which leads to an implementation that is easier to code.
- Ultimately, the recursion is eliminated when the compiler creates assembly language (it does this by implementing the stack).
- If the recursion tree has a simple form, the iterative version may be better.
- If the recursion tree appears quite “bushy”, with very few duplicate tasks, then recursion is likely the natural solution.

Types of recursion

- Linear recursion: the method makes a single call to itself
- Tail recursion: the method makes a single call to itself, as the last operation
- Binary recursion: the method makes two calls to itself
- Exponential recursion: the method makes more than two calls to itself

Tail recursion

- Tail recursion is when the last operation in a method is a single recursive call.
- Each time a method is invoked, the method's activation frame (record) is placed on top of the program stack.
- In this case, there are multiple active stack frames which are unnecessary because they have finished their work.
- Can be expensive and inefficient, so use carefully!

Infinite recursion

- Infinite recursion occurs when a recursive method does not have a base case
- Consider the method to the right:
- If we call infinite(1), the next call will be infinite(0), then infinite(-1), then infinite(-2) etc…
- In Java, this method will keep making recursive calls to itself until a StackOverflowError occurs (recursive calls have taken up all available memory)
- In Python, this function will continue calling itself until it exceeds the limit for recursion depth (1000 by default)

Java

```java
void infinite(int x) {
  infinite(x-1);
}
```

Python

```python
def infinite(x):
    infinite(x-1)


infinite(1)
# RecursionError:
# maximum recursion depth exceeded
```

Circular recursion

- Circular recursion occurs when recursive calls stop making progress towards the base case
- Consider the method to the right:
- If we call circular(1), the next call will be circular(2), then circular(1), then circular(2) etc…
- As with the infinite recursion example, this method will keep making recursive calls to itself until a StackOverflowError occurs (recursive calls have taken up all available memory)

Java

```java
void circular(int x) {
  if(x==1) {
    circular(x+1);
  }
  circular(x-1);
}
```

Python

```python
def circular(x):
    if x == 1:
        circular(x + 1)
    circular(x - 1)


circular(1)  # RecursionError:
# maximum recursion depth exceeded
# in comparison
```

Rules for recursive algorithms

1. **Base case**: a recursive algorithm must always have a base case which can be solved without recursion. Methods without a base case will result in infinite recursion when run.
2. **Making progress**: for cases that are to be solved recursively, the next recursive call must be a case that makes progress towards the base case. Methods that do not make progress towards the base case will result in circular recursion when run.
3. **Design rule**: Assume that all the recursive calls work.
4. **Compound interest rule**: Never duplicate work by solving the same instance of a problem in separate recursive calls.

Designing recursive algorithms

- Think about the task which you wish to accomplish, and try to identify any recurring patterns, e.g. similar operations that must be conducted, like traversing through nested directories on a file system
- Divide the problem up using these recurring operations
- Then:
  - Identify cases you know can be solved without recursion (base cases). Avoid ending with a multitude of special cases; rather, try to identify a simple base case
  - Invoke a new copy of the method within each recursive step
  - Each recursive step resembles the original, larger problem
  - Make progress towards the base case(s) with each successive recursive step/call

Recap

- A recursive method is one which calls itself within its method body
- Recursion allows us to solve a problem, by breaking it up into smaller instances of the same problem
- Recursive methods must always have a base case which may be solved without recursion
- In the next lecture we will consider some example problems which may be solved using recursion