Recursion and termination

```
static void f(int n)
{
  System.out.println(n);
  f(n);
}
```

Recursion and termination

```
static int f(int n)
{
  System.out.println(n);
  return f(n-1);
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n);
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n-1);
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  else if (n % 2 == 0) return n / 2;
  else return f( f( 3*n+1 ) );
}
```

Recursion and termination

If n is odd then the third case applies, and we have to compute

$$f(f(3n+1))$$

But, since n is odd, then $n = 2k+1$ for some $k \geq 0$, which implies that

$$\begin{aligned} 3n+1 &= 3(2k+1)+1 \\ &= 6k+3+1 \\ &= 6k+4 \\ &= 2(3k+2) \end{aligned}$$

so $3n+1$ is even!

Therefore

$$f(3n+1) = \frac{3n+1}{2}$$

so we can rewrite

$$f(f(3n+1))$$

as

$$f(\frac{3n+1}{2})$$

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  else if (n % 2 == 0) return n / 2;
  else return f( (3*n+1)/2 );
}
```

Recursion and termination

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n/2 & \text{if } n \text{ is even} \\ f(\frac{3n+1}{2}) & \text{if } n \text{ is odd} \end{cases}$$

Question: for all $n > 0$, is $\frac{3n+1}{2} < n$?

Answer: no! For $n > 0$

$$3n > 2n$$

so

$$3n + 1 > 2n$$

therefore

$$\frac{3n+1}{2} > n$$

Hence the argument of the recursive call increases!

Does this mean, then that f cannot terminate?

No! eventually $\frac{3n+1}{2}$ will be even, thus reaching a base case (but the proof is hard!)

Recursion and termination

Determining whether a program might terminate is in general hard

Furthermore, it is impossible to write a program that takes as input any program P and decides whether P terminates or not.

Continuation-passing and tail recursion

```
static int factorial_2(int n)
{
  return cont_factorial(n, 1);
}

static int cont_factorial(int n, int result)
{
  if (n == 0) return result;
  return cont_factorial(n - 1, result * n);
}
```

Continuation-passing and tail recursion

`factorial_2(4)`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 1 * 4);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 4 * 3);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 12 * 2);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns `cont_factorial(0, 24 * 1);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns `cont_factorial(0, 24);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns `cont_factorial(0, 24);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns 24

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2					
3					
4					

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6				
3	9				
4	12				

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6	8	10	12	14
3	9	17	27	39	53
4	12	29	56	95	148

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6	8	10	12	14
3	9	17	27	39	53
4	12	29	56	95	148

Recursion and termination

Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \text{ and } n \geqslant 0 \\ A(m-1,1) & \text{if } m \geqslant 1 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m \geqslant 1 \text{ and } n \geqslant 1 \end{cases}$$

m\n	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125

Recursion and termination

Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \text{ and } n \geqslant 0 \\ A(m-1,1) & \text{if } m \geqslant 1 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m \geqslant 1 \text{ and } n \geqslant 1 \end{cases}$$

m\n	0	1	2	3	4	n
0	1	2	3	4	5	n+1
1	2	3	4	5	6	n+2
2	3	5	7	9	11	2n+3
3	5	13	29	61	125	$2^{(n+3)} - 3$
4	13	65533	$2^{65536} - 3$			$2^{2^{\cdots^{2}}} - 3$

A(4,2) is greater than the number of particles in the universe raised to the power 200

A(5,2) cannot be written as a decimal expansion in the physical universe.

Recursion and termination

```
static int ackermann(int m, int n)
{
  if (m == 0 && n >= 0)
    return n + 1;
  else if (m >= 1 && n == 0)
    return ackermann(m - 1, 1);
  else
    return ackermann(m - 1, ackermann(m, n - 1));
}
```

Double recursion

- Problem: Compute the n-th Fibonacci number
- Analysis: The Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ...is defined by:

$$fib(n) = \begin{cases} 1 & \text{if } n \leqslant 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Implementation:

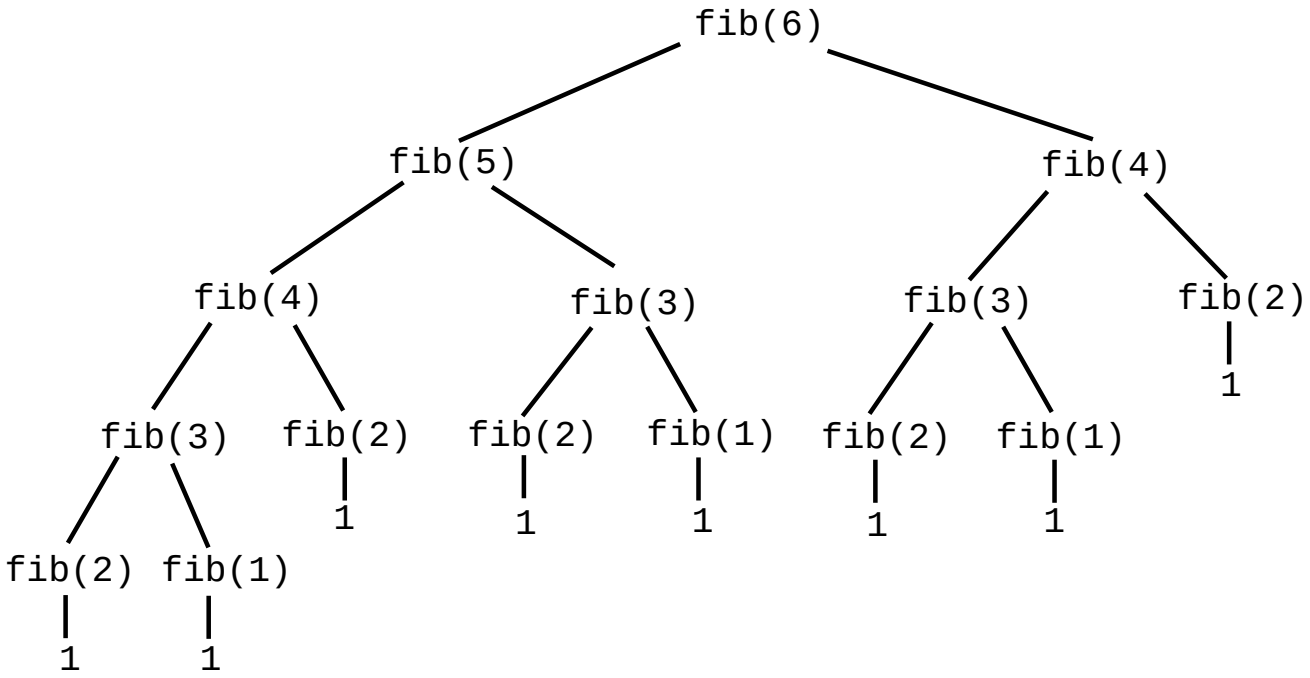
```
static int fib(int n)
{
    if (n <= 2) {
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
```

Iteration vs recursion

- Iterative solution to the Fibonacci problem:

```
static int fib(int n)
{
    int a, b, c, i;
    a = 1;
    b = 1;
    c = 1;
    i = 3;
    while (i <= n) {
        c = a + b;
        a = b;
        b = c;
        i++;
    }
    return c;
}
```

Execution trees



Memoization

- We can write imperative versions of recursive programs
- A memoized algorithm is an algorithm which uses an array to keep track of partially computed solutions.
- By remembering a previous solution (using memoization,) a recursive algorithm can be rewritten so that if at any point a solution has already been computed, and the recursion requires it again, then it is looked up in the array instead of being recomputed.
- Memoization can be applied when the arguments of the recursive function are natural numbers, or can be associated with natural numbers.
- In the memoized solution, the arguments are going to be indices of the array storing the partial solutions to the recursion.

Memoization

```
int memoized_fib(int n)
{
  if (n <= 1) return 1;
  int[] mem = new int[n+1];
    // mem[i] will contain fib(i)
  mem[0] = 1;
  mem[1] = 1;
  int i = 2;
  while (i <= n) {
    mem[i] = mem[i-1]+mem[i-2];
    i++;
  }
  return mem[n];
}
```

Memoization

- Memoization implies a trade-off: efficiency is gained, but at the cost of taking up more memory space, but the recursive definition might also take up a lot of space, since each recursive call generates a new frame.

- If the recursive function has two parameters, then the memoized version uses a two-dimentional array. In general with n-parameters, the memoized version needs an n-dimensional array.

```
double f(int a, int b)
{
  if (a <= 0) return 2.0;
  else if (b <= 0) return 3.0 * a;
  else return f(a - 1, b) + 5.0 * f(a, b - 1);
}
```

Memoization

```
double memo_f(int a, int b)
{
  double[][] table = new double[a+1][b+1];
    //table[i][j] will contain f(i,j)
  int row = 0, col = 0;
  while (row <= a) {
    col = 0;
    while (col <= b) {
      if (a <= 0) table[row][col] = 2.0;
      else if (b <= 0)
        table[row][col] = 3.0 * row;
      else
        table[row][col] = table[row-1][col]
                        + 5.0 * table[row][col-1];
      col++;
    }
    row++;
  }
  return table[a][b];
}
```

Memoization

We can use any part of the solution which has already been computed:

Generalized fibonacci: gf(n) is the sum of the first n-1 gf numbers: 1, 1, 2, 4, 8, 16, 32, ...

```
int gf(int n)
{
  if (n <= 1) return 1;
  int sum;
  for (int i = 0; i < n; i++)
    sum = sum + gf(i);
  return sum;
}
```

Memoization

```
int memo_gf(int n)
{
  if (n <= 1) return 1;
  int sum;
  int[] table = new int[n+1];
  table[0] = 1;
  table[1] = 1;
  for (int i = 0; i < n; i++)
    sum = sum + table[i];
  return sum;
}
```

Memoization

The Ackermann function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

A(4,2) has 19729 digits!

```
int ack(int m, int n)
{
  if (m == 0) return n + 1;
  if (m > 0 && n == 0) return ack(m-1,1);
  return ack(m-1,ack(m,n-1));
}
```

Memoization

```
int memo_ack(int m, int n)
{
  int[][] table = new int[m+1][n+1];
  for (int j = 0; j <= m; j++) {
    for (int i = 0; i <= n; i++) {
      if (j == 0) table[j][i] = i + 1;
      else if (i == 0)
        table[j][i] = table[j-1][1];
      else
        table[j][i] = table[j-1][table[j][i-1]];
    }
  }
  return table[m][n];
}
```

The end