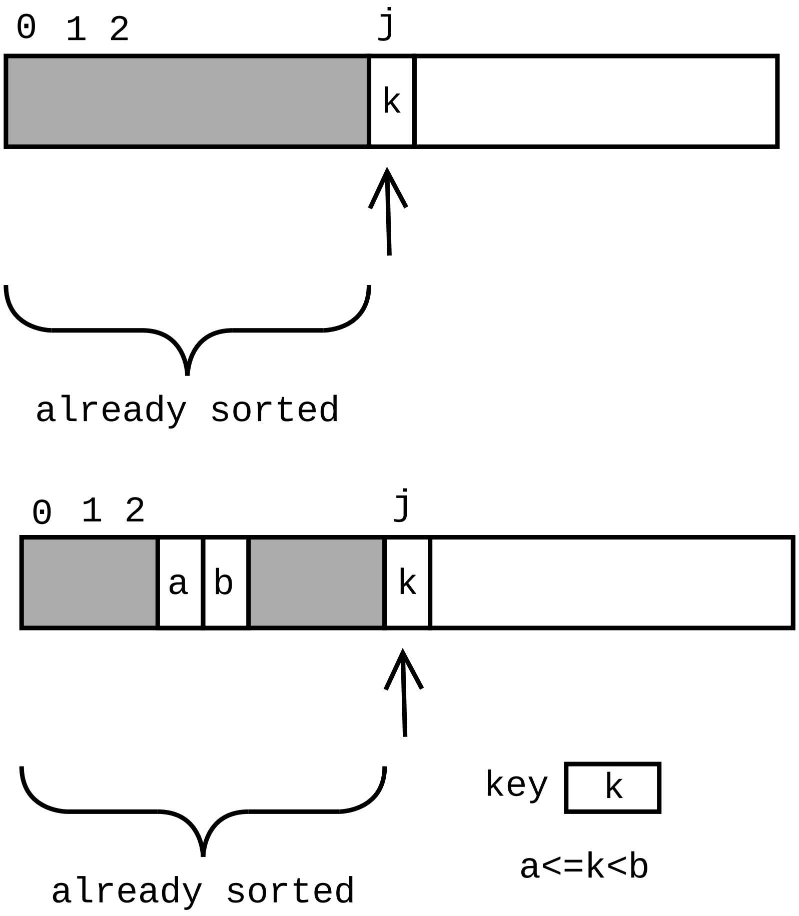
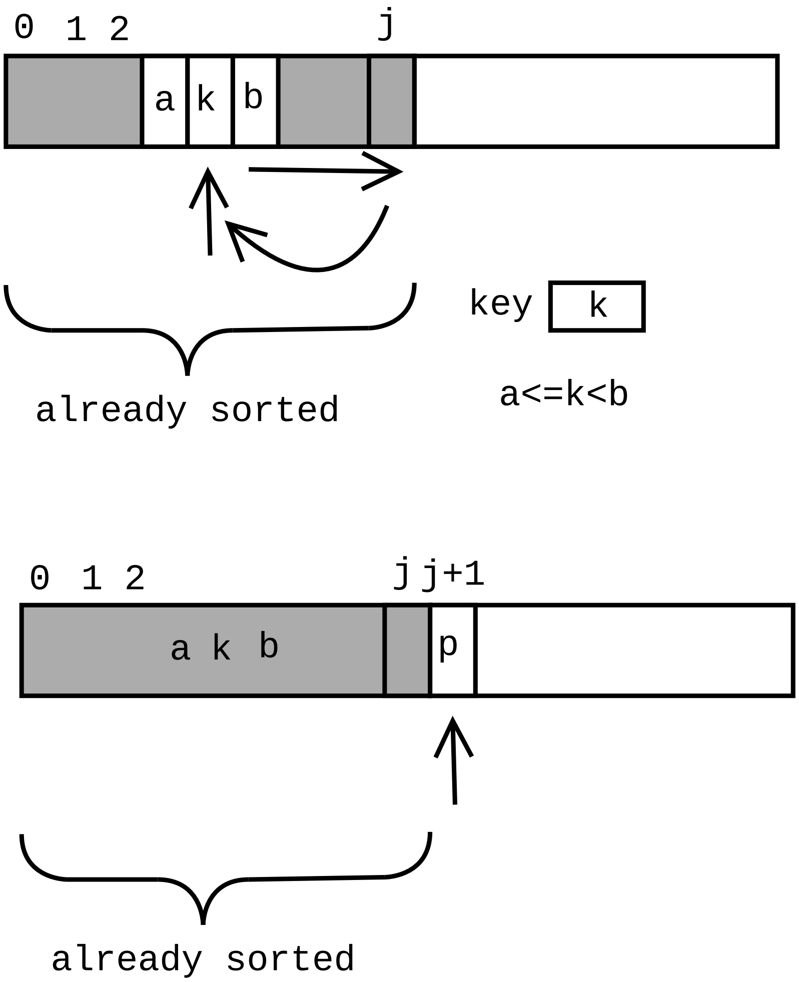
# Review

- Growing arrays:
  - Create new bigger array,
  - Copy contents of the old in the new
  - Adjust references
- Adding an object to an array
  - Check if the array is full
  - If not, add the object to the first available cell
  - Otherwise, grow the array and add it in the first available cell
- Sorting

# Insertion sort



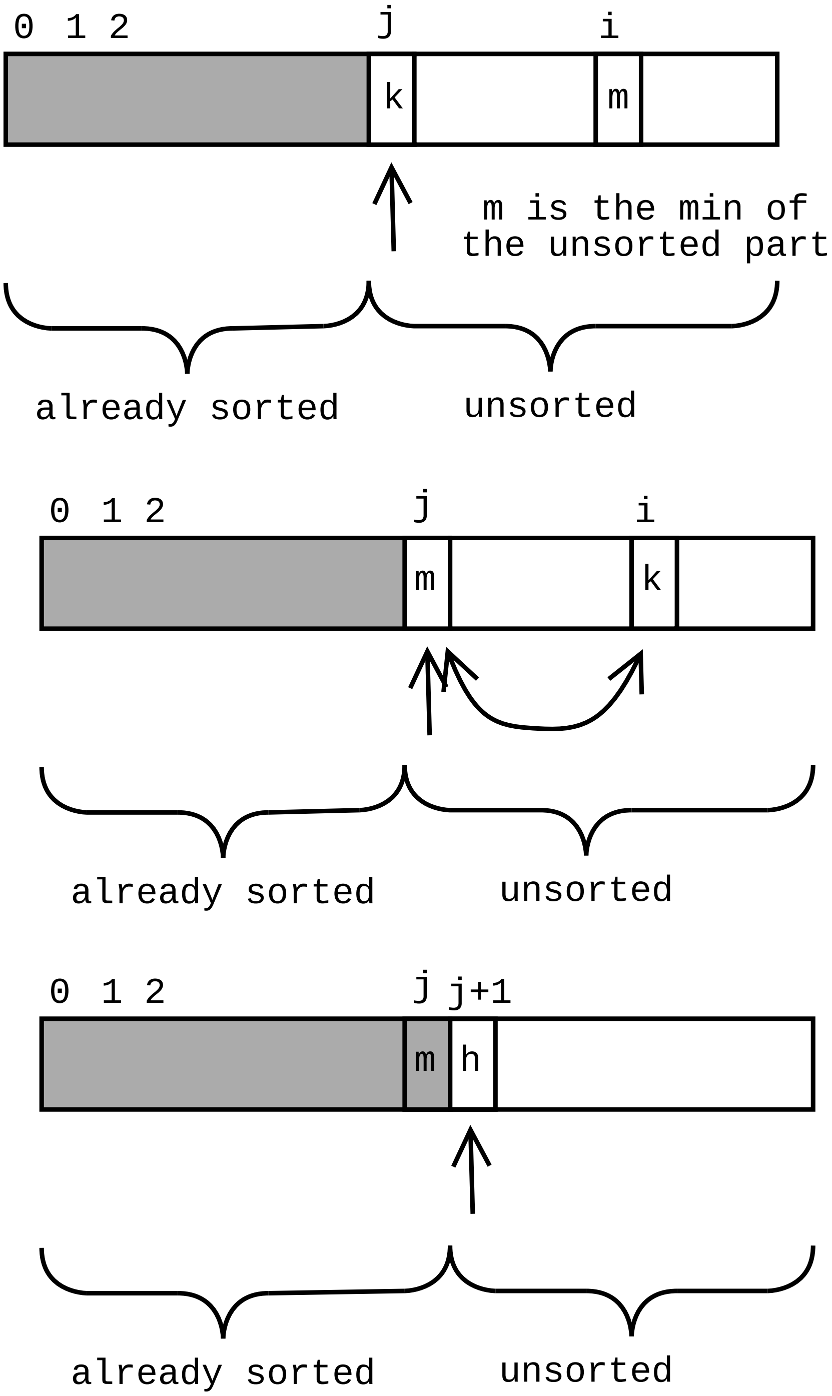
# Insertion sort



# Insertion sort

```
void insertion_sort(int[] a)
{
  int i, j, key;
  for (j = 1; j < a.length; j++) {
    key = a[j];
    i = j - 1;
    while (i >= 0 && key < a[i]) {
      a[i+1] = a[i];
      i--;
    }
    a[i+1] = key;
  }
}
```

# Selection sort



# Selection sort

```
void selection_sort(int[] a)
{
  int minimum, min_index, temp;
  for (int j = 0; j <= a.length - 2; j++) {
    minimum = a[j];
    min_index = j;
    for (int i = j + 1; i <= a.length - 1; i++) {
      if (a[i] < minimum) {
        minimum = a[i];
        min_index = i;
      }
    }
    temp = a[j];
    a[j] = a[min_index];
    a[min_index] = temp;
  }
```

# Sorting arrays of Objects

Choose a key, which can be compared.

```
class Movie {
  private String title, director;
  //...
  public String get_title() { return title; }
  public String get_director() { return director; }
  //...
}
```

# Sorting arrays of Objects

- Comparing strings: Lexicographical order
- The `compareTo` method from the `String` class
- `s1.compareTo(s2)` returns a negative integer if `s1` is lexicographically before `s2`, 0 if they are equal, and a positive integer if `s1` is lexicographically after `s2`.

```
String s1 = “aac”, s2 = “aaf”;
int n = s1.compareTo(s2); // n = -3;
String s3 = “aacgg”;
int m = s3.compareTo(s2); // n = -3
int k = s3.compareTo(s1); // n = 2
```

# Sorting arrays of Objects

```
void insertion_sort(Movie[] a)
{
  int i, j;
  String key;
  for (j = 1; j < a.length; j++) {
    key = a[i].get_title();
    i = j - 1;
    while (i >= 0
      && key.compareTo(a[i].get_title()) < 0 ) {
      a[i+1] = a[i];  // copy the reference
      i--;
    }
    a[i+1] = key;
  }
}
```

# Sorting arrays of Objects

```
void selection_sort(Movie[] a)
{
  int min_index;
  String minimum;
  Movie temp;
  for (int j = 0; j <= a.length - 2; j++) {
    minimum = a[j].get_title();
    min_index = j;
    for (int i = j + 1; i <= a.length - 1; i++) {
      String current_key = a[i].get_title();
      if (current_key.compareTo(minimum) < 0) {
        minimum = current_key;
        min_index = i;
      }
    }
    temp = a[j];
    a[j] = a[min_index];
    a[min_index] = temp;
  }
```

# Linear search

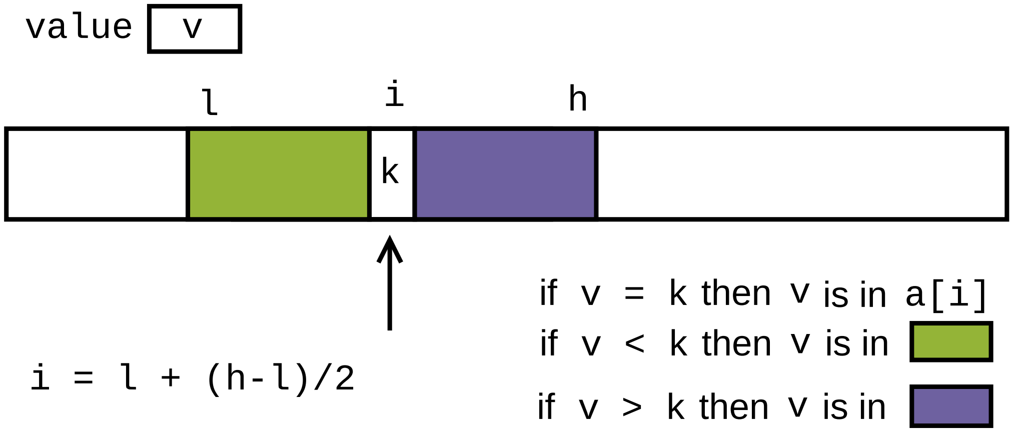
```
int linear_search(int[] a, int value)
{
  int index = 0;
  while (index < a.length) {
    if (value == a[index]) {
      return index;
    }
    index++;
  }
  return -1;  // Not found
}
```

- This works for unsorted arrays

# Binary search

- But if we know that the array is sorted, we can improve the speed of searching by ignoring parts which do not need to look at.

- If we are looking for a value `v` in an array `a`, and we have already narrowed down the search space to `a[l..h]`, then



i = l + (h-l)/2

if v = k then v is in a[i]

if v < k then v is in 🟩

if v > k then v is in 🟪

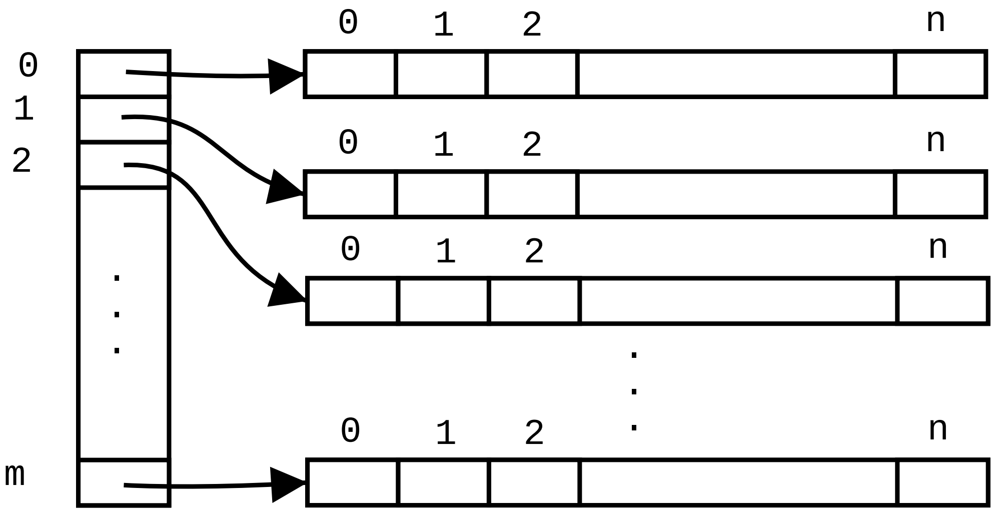
# Binary search

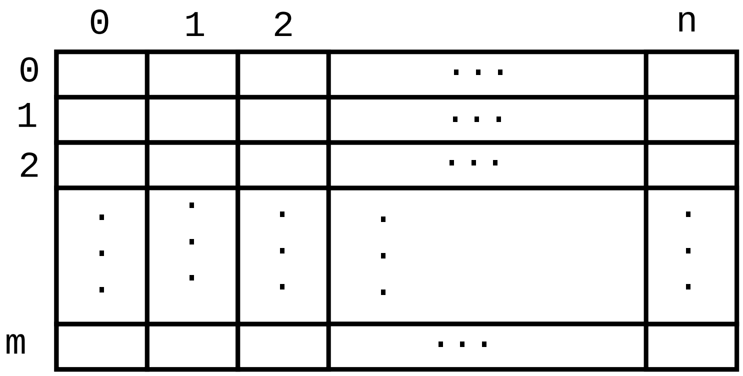
```
int binary_search(int[] a, int value)
{
  int lower = 0, higher = a.length - 1, index;
  while (lower <= higher) {
    index = lower + (higher - lower) / 2;
    if (value == a[index]) {
      return index;
    }
    else if (value < a[index]) {
      higher = index - 1;
    }
    else {  // value > a[index]
      lower = index + 1;
    }
  }
  return -1;  // Not found
}
```

# Binary search

```
int binary_search(Movie[] a, String title)
{
  int lower = 0, higher = a.length - 1, index;
  String current_title;
  int comparison;
  while (lower <= higher) {
    index = lower + (higher - lower) / 2;
    current_title = a[index].get_title();
    comparison = title.compareTo(current_title);
    if (comparison == 0) {
      return index;
    }
    else if (comparison < 0) {
      higher = index - 1;
    }
    else {  // comparison > 0
      lower = index + 1;
    }
  }
  return -1;  // Not found
}
```

# Multidimensional arrays





# Multidimensional arrays

- A two-dimensional array is an array of arrays.

  ```
  int[][] table = new int[5][10];

  for (int row=0; row < table.length; row++)
    for (int col=0; col < table[row].length; col++)
      table[row][col] = row * 10 + col;

  for (int row = 0; row < table.length; row++) {
    for (int col=0; col < table[row].length; col++)
      System.out.print(table[row][col]+"\t");
    System.out.println();
  }
  ```

- A multidimensional array is an n-dimensional array, i.e. an array of arrays of arrays of ...

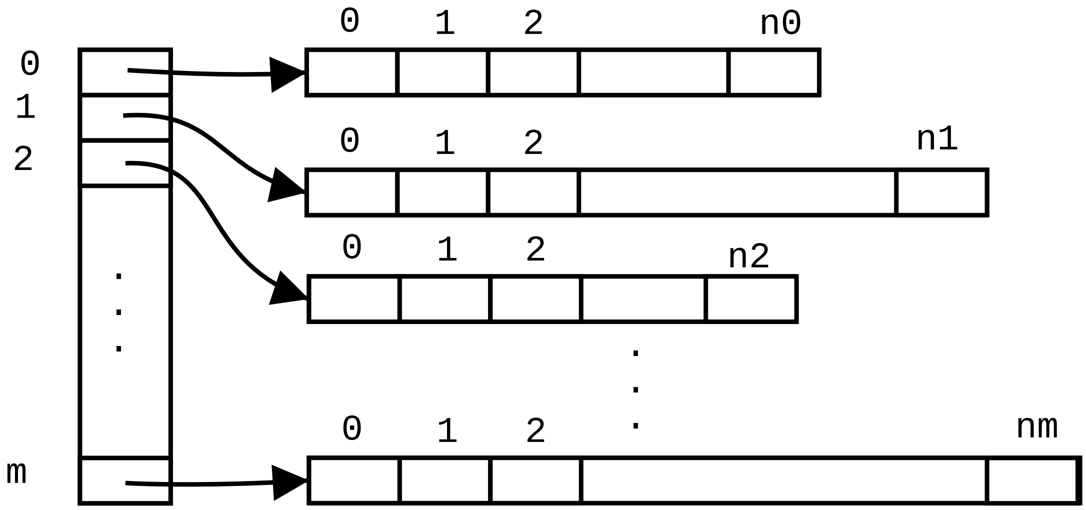
- Processing nested arrays is commonly done with nested loops.

# Multidimensional arrays

- A two-dimensional array can be an array of objects

```
class A { int x; }

// and in the client
A[][] table = new A[5][10];
for (int row=0; row < table.length; row++)
  for (int col=0; col < table[row].length; col++)
    table[row][col] = new A();
    table[row][col].x = row * 10 + col;
  }

for (int row = 0; row < table.length; row++) {
  for (int col=0; col < table[row].length; col++)
    System.out.print(table[row][col].x+"\t");
  System.out.println();
}
```

# Multidimensional arrays



# Multidimensional arrays

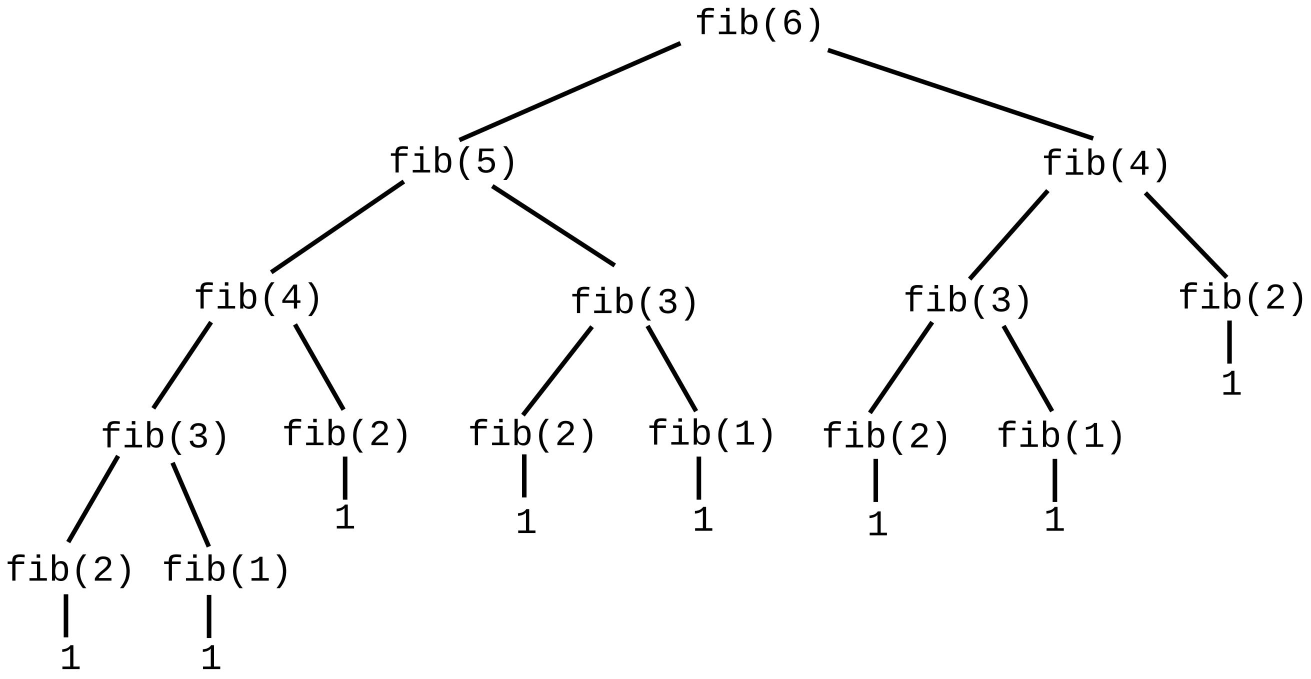
- Each row can have different length

```
class A { int x; }

// and in the client
A[][] table = new A[5][];
for (int row=0; row < table.length; row++) {
  table[row] = new A[row];
  for (int col=0; col < table[row].length; col++)
    table[row][col] = new A();
    table[row][col].x = row * 10 + col;
  }
}

for (int row = 0; row < table.length; row++) {
  for (int col=0; col < table[row].length; col++)
    System.out.print(table[row][col].x+"\t");
  System.out.println();
}
```

# Memoization

- In recursion, we often compute something many times (e.g. fibonacci.)

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



# 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];
}
```