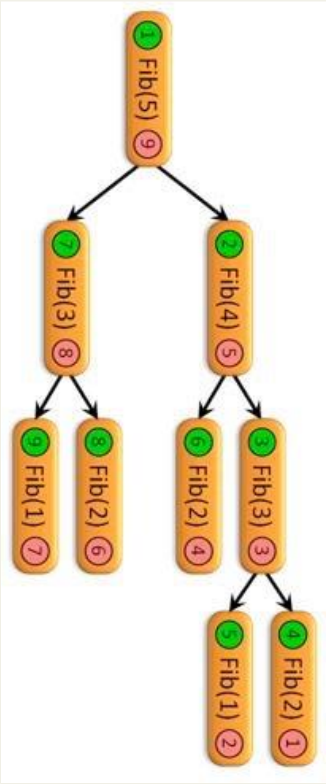
Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa



RECURSÃO

- Dividir e conquistar
- Recursão Linear
- *Tail recursion*
- Recursão Binária
- *Memoization*
- Sistemas de Recorrência

Dividir para Conquistar

- Uma técnica para resolver um problema complicado é dividi-lo em problemas mais pequenos (independentes), encontrar soluções para estes sub-problemas e depois combinar estas soluções de forma a obter a solução do problema inicial.

- No caso particular dos sub-problemas serem apenas instâncias mais pequenas do problema inicial então a aplicação desta técnica dá origem a um algoritmo recursivo

- **Exemplo**: Calcular a potência x^n de um número

Recursão Linear

```
Algorithm Power(x, n){
  if n = 0 then
      return 1
  if n is odd then
      y := Power(x, (n - 1)/2)
      return x * y * y
  else
      y := Power(x, n/2)
      return y * y
}
```

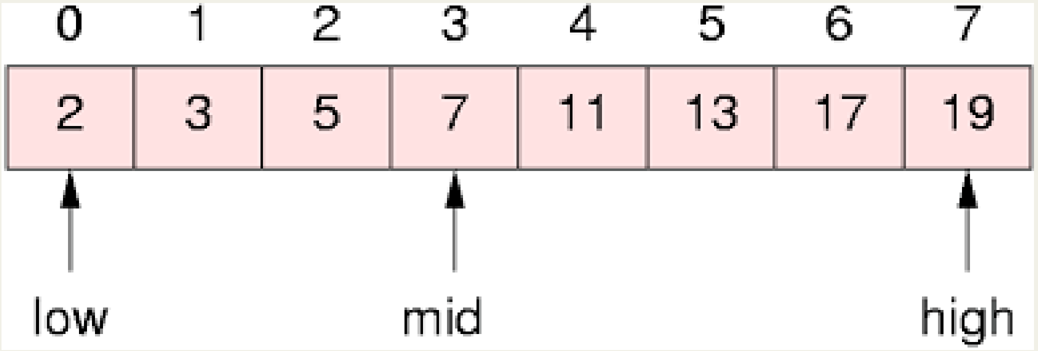
Recursão Linear

```
Algorithm Power(x, n){
  if n = 0 then
       return 1
  if n is odd then
       y := Power(x, (n - 1)/2)
       return x * y * y
  else
       y := Power(x, n/2)
       return y * y
}
```

- A análise do tempo de execução do algoritmo pode fazer-se recorrendo a traços de execução.
- O número de chamadas recursivas é *1+log n*, donde *T(n) é O(log n)*

Exemplo: Pesquisa Binária

- Para o problema da pesquisa vimos o algoritmo de pesquisa linear cujo tempo de execução é *O(n)*
- Para o caso em que o *input* é um vector X que se encontra ordenado temos um algoritmo melhor, conhecido por **pesquisa binária**



Exemplo: Pesquisa Binária

- Compara-se o elemento x com o "**elemento do meio**" X[mid]
  - se é igual, problema resolvido
  - se é menor procura-se em X[0] ... X[mid-1]
  - se é maior procura-se em X[mid+1] ... X[n-1]

- Cálculo de *mid*:

  (min+max)/2 vs min + (max-min)/2

  - apesar das expressões serem matematicamente equivalentes, quando se tem inteiros limitados a um dado intervalo a equivalência deixa de se verificar (a primeira versão pode conduzir a *overflows*)

Pesquisa Binária

```
Algorithm binarySearch(X,x,min,max){
  if min > max then
    return false
  mid := min + (max-min)/2
  if x = X[mid] then
    return true
  if x < X[mid] then
    return binarySearch(X,x,min,mid-1)
  else
    return binarySearch(X,x,mid+1,max)
}
```

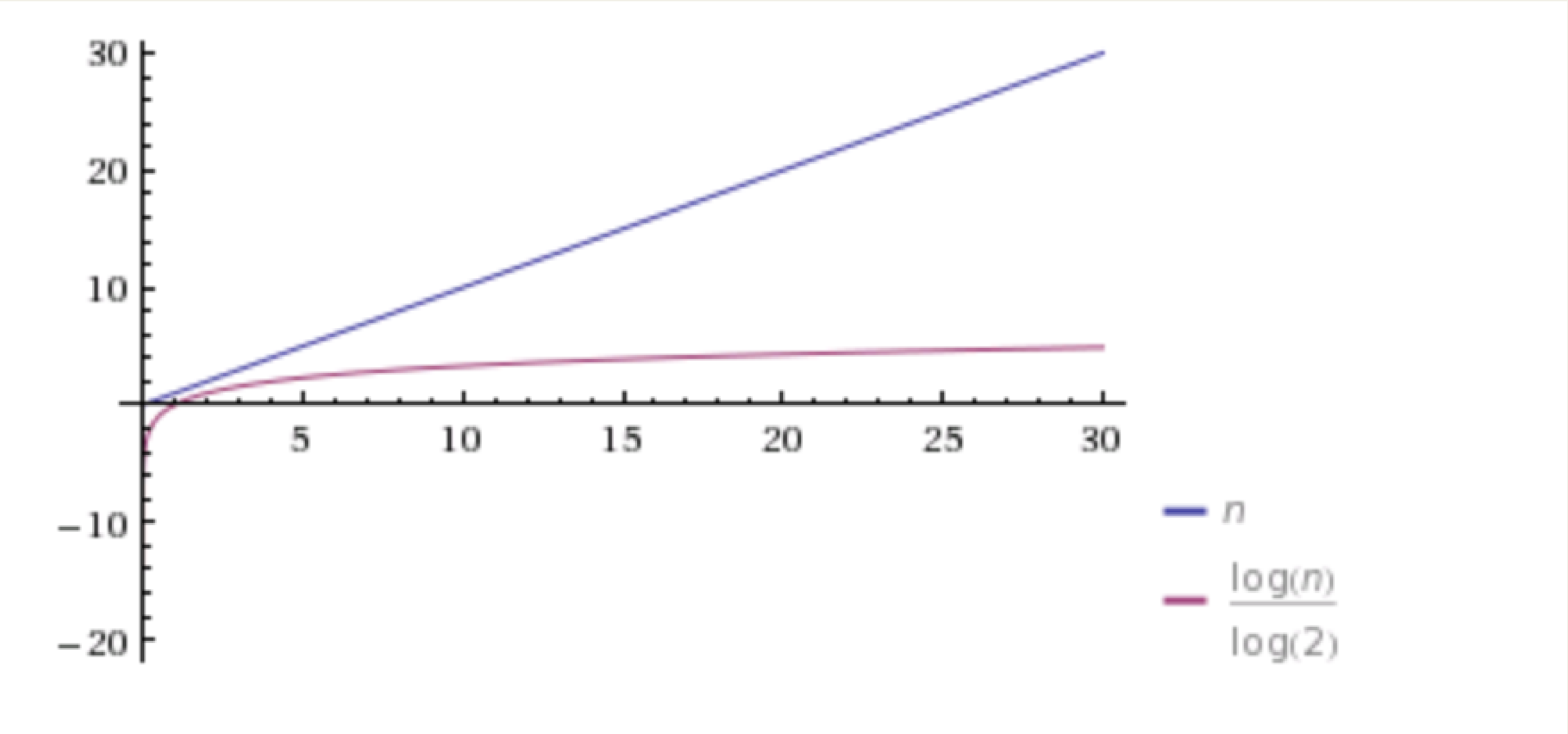
T(n) ?

Pesquisa Binária

```
Algorithm binarySearch(X,x,min,max){
  if min > max then
     return false
  mid := min + (max-min)/2
  if x = X[mid] then
     return true
  if x < X[mid] then
    return binarySearch(X,x,min,mid-1)
  else
    return binarySearch(X,x,mid+1,max)
}
```

Na execução de `binarySearch(X,x,0,max)`, o número de chamadas recursivas é, no pior caso, *1+log n*, donde *T(n) é O(log n)*

Pesquisa Linear vs Pesquisa Binária



Recursão: Impacto em termos de memória

- A utilização da recursão permite a definição de algoritmos de uma forma sucinta e elegante mas que em geral tem um custo elevado em termos de memória

- De cada vez que se faz uma chamada recursiva é necessário guardar o estado da chamada ativadora (que vai ficar suspensa)

- Esta informação é guardada numa pilha (***call stack***) e só é retirada da pilha (libertando a memória ocupada) quando a chamada termina (quando chega ao *return*)

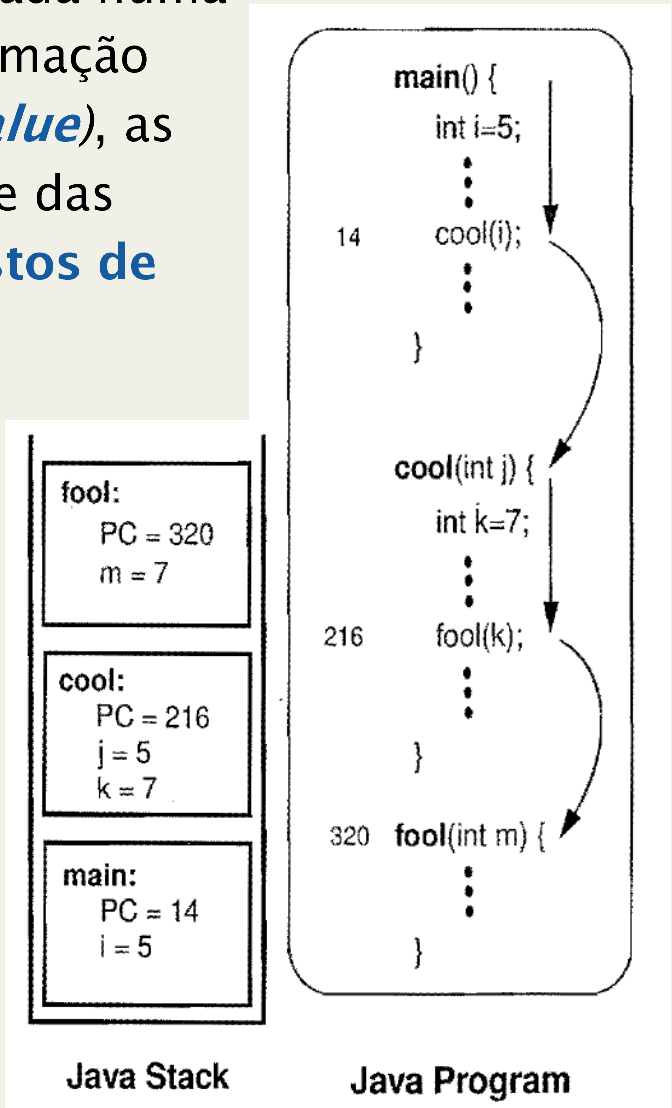
Java Stack

- Durante a execução de um programa é guardada numa pilha (*Java Method Stack* ou ***Java Stack***) informação sobre os parâmetros (no Java é por ***call by value***), as variáveis locais e outra informação importante das chamadas a métodos. São chamados os **registos de activação**.

- Quando a memória disponível para guardar esta informação se esgota, o problema é assinalado com

  StackOverflowError

- Uma das causas mais comuns deste erro é recursão infinita ou excessivamente profunda



Recursão

- **Vantagens**:
  - muitos problemas (e estruturas) são naturalmente definidos recursivamente
  - simplicidade da solução
  - fácil de perceber
- **Desvantagens**:
  - memória
  - desempenho
  - possibilidade de trabalho redundante

Tail Recursion

- Quando num algoritmo recursivo a chamada recursiva é a última coisa a ser feita (não ficam pendentes outras operações) estamos na presença de ***tail recursion***
- Este tipo de recursão é importante porque um algoritmo com *tail recursion* pode ser facilmente convertido num algoritmo não recursivo, resultando numa poupança de memória
- Os compiladores em geral geram código optimizado para situações de *tail recursion* (diz-se neste caso que implementam a ***tail-call optimization***)
  - o código gerado para as chamadas que são *tail recursive* não requerem que se gaste mais espaço da *call stack*

Exemplo

```
Algorithm ReverseArray(A, i, j){
  if i<j then
     swap A[i] and A[j]
     ReverseArray(A, i+1, j-1)
  return
}
```

versão *tail recursive*

versão iterativa *?*

Exemplo

```
Algorithm ReverseArray(A, i, j){
  if i<j then
     swap A[i] and A[j]
     ReverseArray(A, i+1, j-1)
  return
}
```

versão *tail recursive*

```
Algorithm ItReverseArray(A, i, j){
  while i<j
     swap A[i] and A[j]
     i := i+1
     j := j-1
  return
}
```

versão iterativa

Exemplo

```
Algorithm Sum(A, n){
  if n>0 then
     return A[n-1]+Sum(A, n-1)
  return 0
}
```

versão *tail recursive ?*

Exemplo

```
Algorithm Sum(A, n){
  if n>0 then
     return A[n-1]+Sum(A, n-1)
  return 0
}
```

versão *tail recursive*

```
Algorithm SumAux(A, n, acc){
  if n>0 then
     return SumAux(A, n-1, acc+A[n-1])
  return acc
}
Algorithm Sum(A, n){
  return SumAux(A, n, 0)
}
```

Exemplo

```
Algorithm SumAux(A, n, acc){
 if n>0 then
     return SumAux(A, n-1, acc+A[n-1])
  return acc
}
Algorithm Sum(A, n){
  return SumAux(A, n, 0)
}
```

versão iterativa

```
Algorithm ItSum(A, n){
  acc := 0
  while n>0
     acc := acc+A[n-1]
     n := n-1
  return acc
}
```

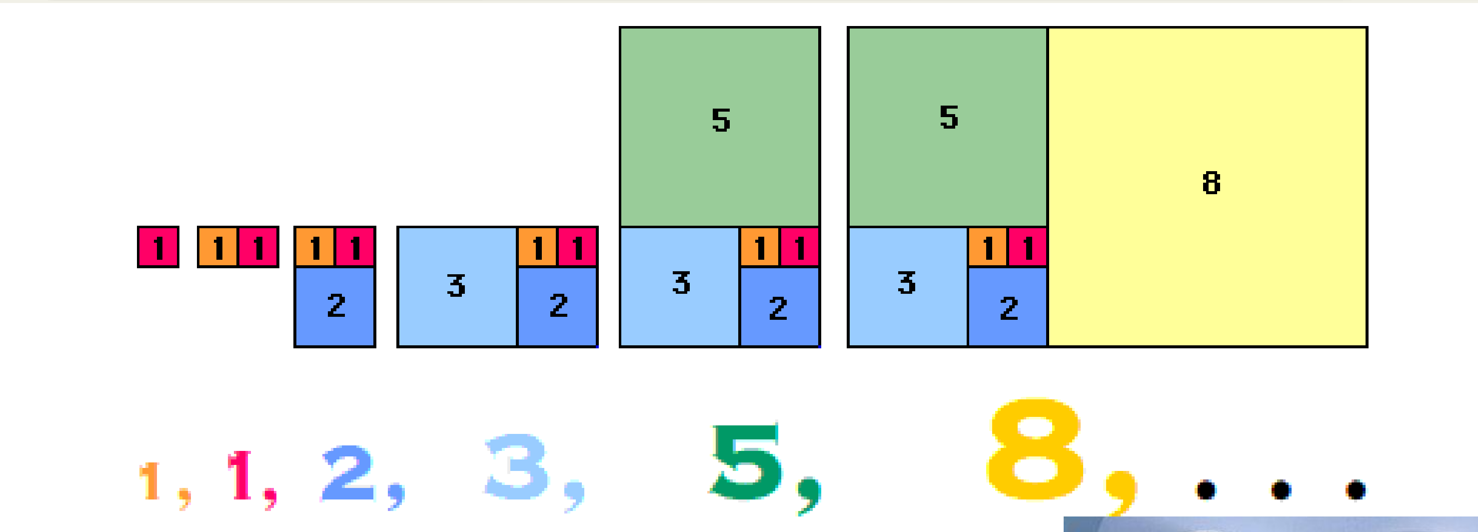
Recursão Binária

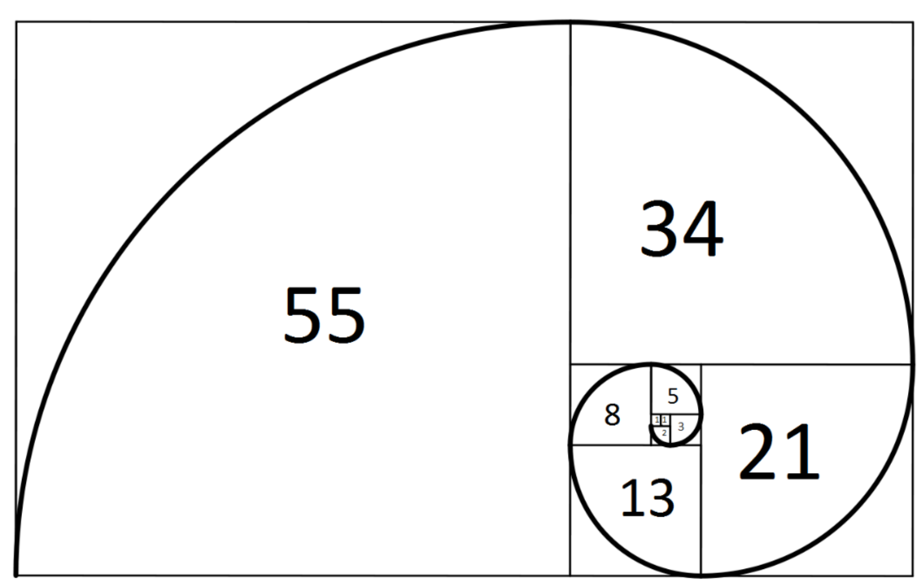
- Ocorre quando cada invocação do algoritmo (em casos que não pertencem à base) dá origem a duas chamadas recursivas

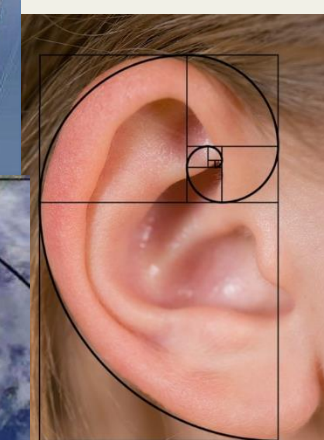
- **Exemplo**
  - Algoritmo recursivo para calcular a Sucessão de Fibonacci

```
Algorithm Fib(n){
  if n = 0 then
     return 0
  if n = 1 then
     return 1
  return Fib(n-1)+Fib(n-2)
}
```

Sequência de Fibonacci

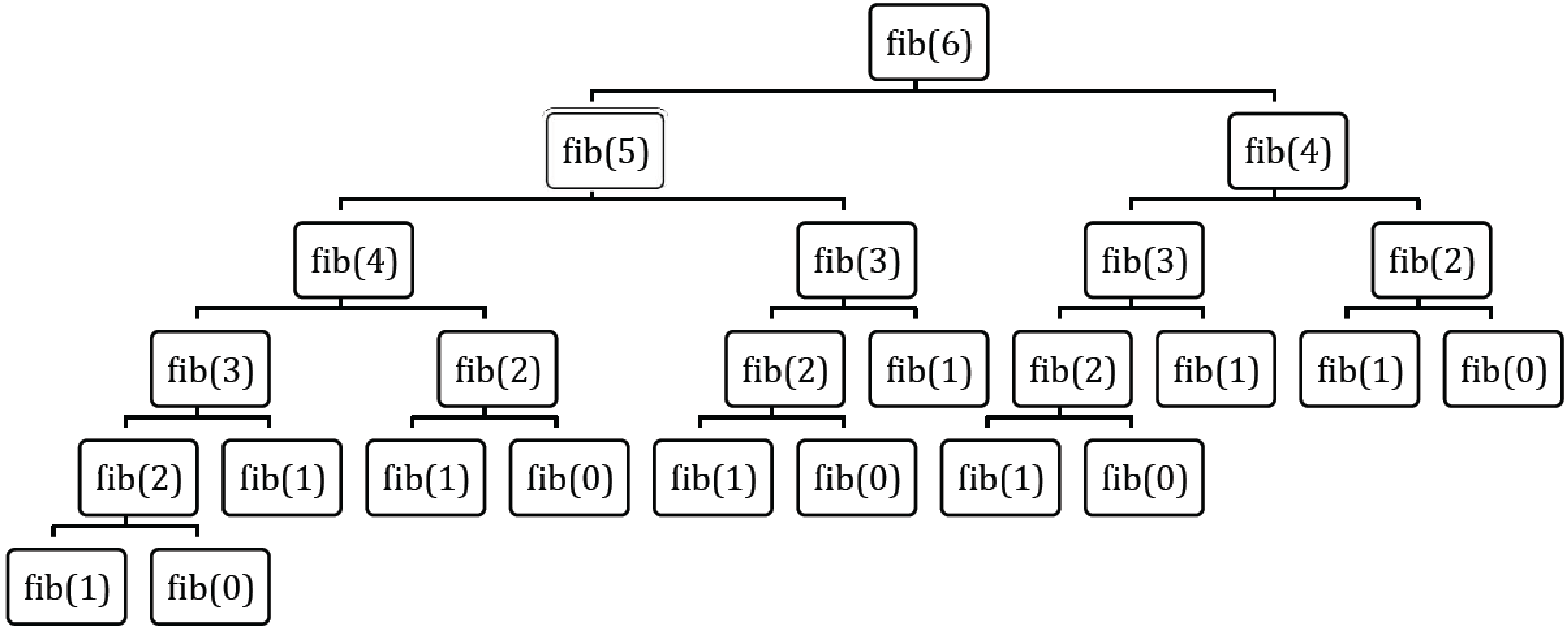






Recursão Binária

- Árvore de chamadas recursivas para calcular **Fib(6)**



Recursão Binária

```
Algorithm Fib(n){
   if n = 0 then
     return 0
   if n = 1 then
     return 1
   return Fib(n-1)+Fib(n-2)
}
```

- O número de chamadas recursivas é $O(2^n)$
- Prova-se que $T(n)$ é $\Theta(((1+\sqrt{5})/2)^n)$

  $(1+\sqrt{5})/2$ é o golden ratio ($\varphi \approx 1.618$, o número de ouro)
- Existe uma versão recursiva linear que corre em tempo $O(n)$!!

Memorização (*Memoization*)

- Uma técnica de optimização em que se memorizam resultados das chamadas a funções para evitar voltar a calcular valores já calculados
- No caso de funções recursivas, pode se guardar todos os resultados conseguidos em chamadas recursivas, da primeira vez que a respectiva chamada for activada
  - Antes da execução de uma chamada recursiva, verifica-se se o resultado foi calculado anteriormente
  - A activação de uma determinada chamada recursiva só se dá uma vez
- Esta técnica reduz um algoritmo tipicamente exponencial para a dimensão da memória necessária para guardar todos os resultados gerados pelo algoritmo
- A complexidade espacial da solução cresce

Exemplo: Fibonacci com memorização

```
Algorithm FibMem(n, mem){
  if mem[n] = -1
    if n = 0 then
      mem[n] := 0
    else if n = 1 then
      mem[n] := 1
    else
      mem[n] := FibMem(n-1,mem) + FibMem(n-2,mem)
  return mem[n]
}

Algorithm Fib(n){
  inicializar mem com uma sequencia de (n+1) -1's
  return FibMem(n, mem)
}
```

Programação Dinâmica

- A memorização é uma técnica de **programação dinâmica**, que é uma outra abordagem à resolução de problemas

  - **Divide-and-conquer** Dividir um problema em sub-problemas mais pequenos e **independentes**, encontrar soluções para estes sub-problemas e depois combinar estas soluções de forma a obter a solução do problema inicial

  - **Dynamic Programming** Dividir um problema numa série de sub-problemas que se sobrepõem e encontrar soluções para problemas cada vez maiores

Análise de Algoritmos Recursivos

- Uma forma sistemática de analisar um algoritmo recursivo é usando sistemas de equações de recorrência
- **Exemplo**

```
MERGESORT(A[1..n]):
  if (n > 1)
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])
      MERGESORT(A[m+1..n])
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

Sistemas de recorrência

- Suponha-se que um problema P de tamanho n:
  - para n≤**c** é resolvido directamente com custo inferior a **k** (constante)
  - para n>**c** é dividido em **a** problemas P de tamanho n/**b**
    - o custo desta divisão é **D(n)**
    - o custo de combinar as soluções dos sub-problemas é **C(n)**

- O custo do algoritmo recursivo que se obtém é solução do sistema de recorrência:

$$T(n) = k \qquad \text{para } n \le c$$

$$T(n) = a.T(n/b)+D(n)+C(n) \qquad \text{caso contrário}$$

Sistemas de recorrência

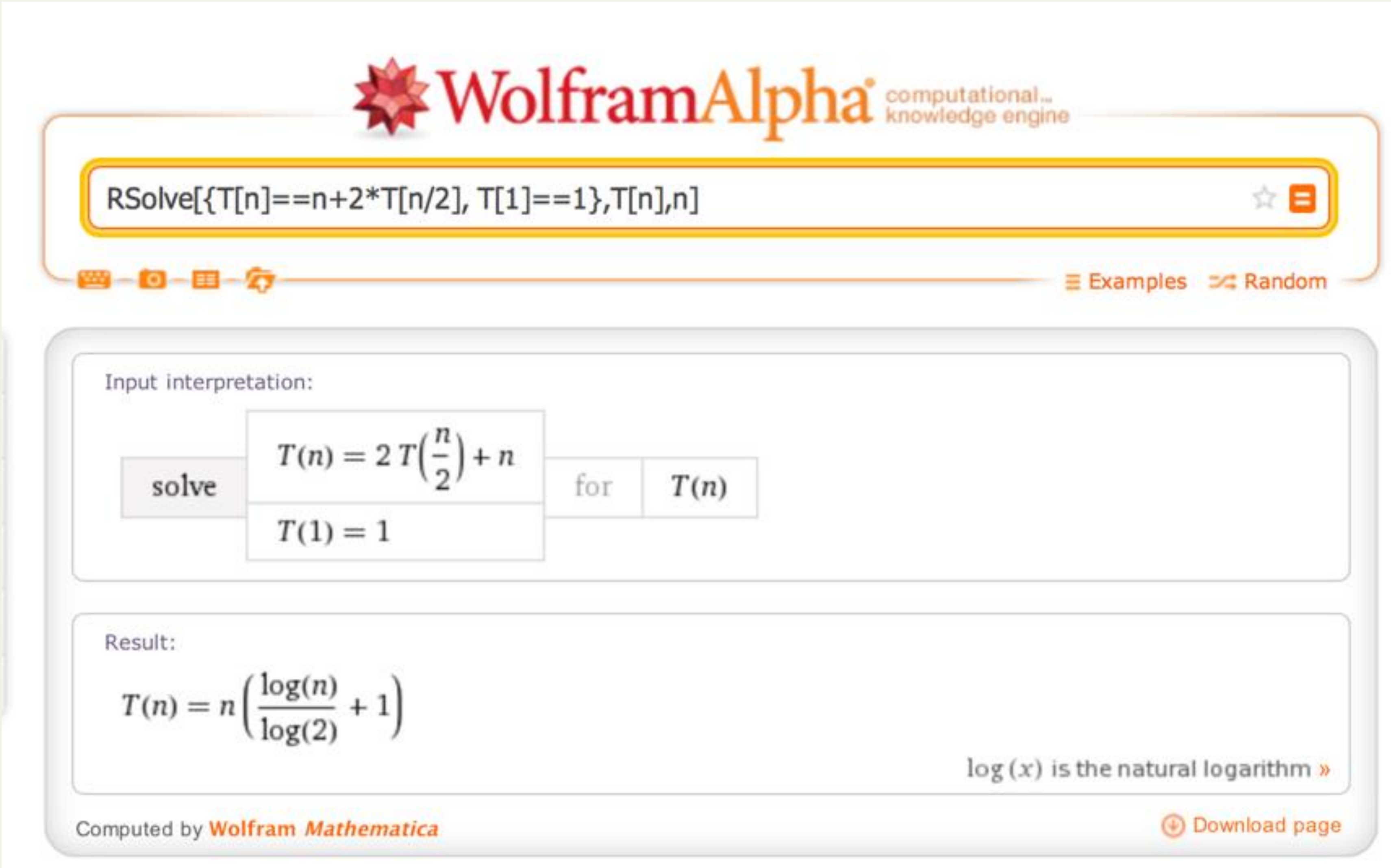
- O custo do algoritmo recursivo que se obtém é solução do sistema de recorrência:

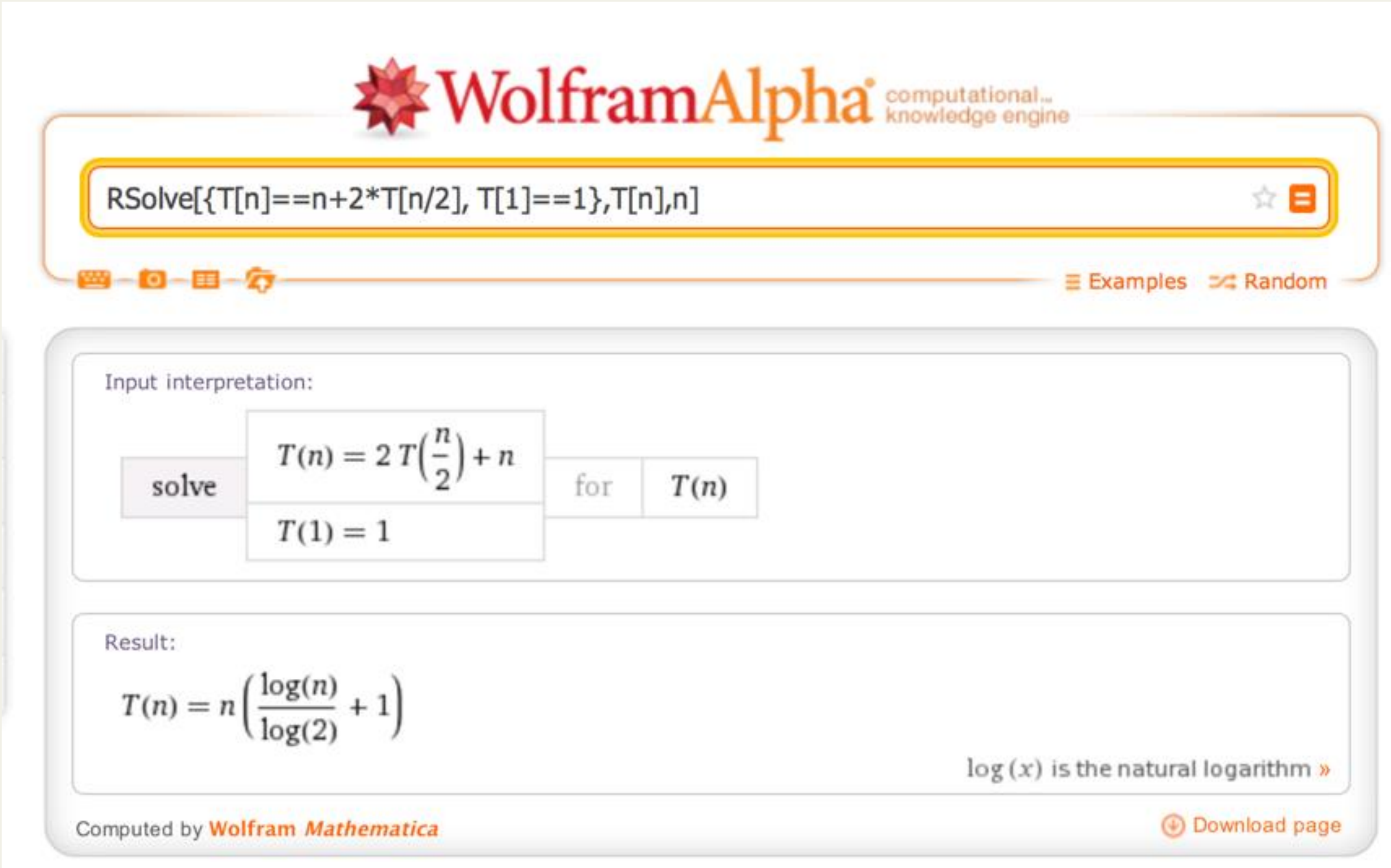
  $T(n) = k$ para $n \leq c$

  $T(n) = a.T(n/b)+D(n)+C(n)$ caso contrário

- Há vários métodos para resolver sistemas de recorrência, como o
  - *master method*
  - *iteration method*

  Em particular, há alguns sistemas de recorrência "tipo", que aparecem frequentemente, e cujas soluções são bem conhecidas.


WolframAlpha computational knowledge engine
RSolve[{T[n]==n+2*T[n/2], T[1]==1},T[n],n]
Examples
Random
Input interpretation:
solve
$T(n) = 2\,T\left(\frac{n}{2}\right) + n$
$T(1) = 1$
for
$T(n)$
Result:
$T(n) = n\left(\frac{\log(n)}{\log(2)} + 1\right)$
$\log(x)$ is the natural logarithm »
Computed by Wolfram Mathematica
Download page


WolframAlpha computational knowledge engine
RSolve[{T[n]==n+2*T[n/2], T[1]==1},T[n],n]
Examples
Random
Input interpretation:
solve $T(n) = 2\,T\left(\frac{n}{2}\right) + n$, $T(1) = 1$ for $T(n)$
Result:
$T(n) = n\left(\frac{\log(n)}{\log(2)} + 1\right)$
$\log(x)$ is the natural logarithm »
Computed by Wolfram Mathematica
Download page