MAC0219 - Relatório do Mini-EP 3

Nathalia Yukimi Uchiyama Tsuno

September 2024

1 Parte I - Buscando o mal pela raíz

A priori, no terminal, executamos a sequência de comandos:

```
$ gcc -pg -g mini_ep3.c -o mini_ep3
$ ./mini_ep3
$ gprof ./mini_ep3
```

Conseguimos o seguinte diagnóstico:

```
Flat profile:

Each sample counts as 0.01 seconds.

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 53.26      0.38     0.38        1   383.51   383.51  slowsort
 39.25      0.67     0.28      200     1.41     1.41  fibonacci
  8.41      0.73     0.06                             frame_dummy
```

```
          Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 1.18% of 0.85 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     85.7    0.00    0.73                 main [1]
                0.41    0.00       1/1           slowsort [2]
                0.31    0.00     200/200         fibonacci [3]
-----------------------------------------------
                                342019695             slowsort [2]
                0.41    0.00       1/1           main [1]
[2]     48.8    0.41    0.00       1+342019695 slowsort [2]
                                342019695             slowsort [2]
-----------------------------------------------
                                458565084             fibonacci [3]
                0.31    0.00     200/200         main [1]
[3]     36.9    0.31    0.00     200+458565084 fibonacci [3]
                                458565084             fibonacci [3]
-----------------------------------------------
                                                 <spontaneous>
[4]     14.3    0.12    0.00                 frame_dummy [4]
```

```
-----------------------------------------------

Index by function name

   [3] fibonacci               [4] frame_dummy             [2] slowsort
```

E quanto ao uso do perf:

```
$ perf record -g ./mini_ep3
$ perf report
```

Temos um diagnóstico semelhante:

```
Samples: 6K of event 'cycles', Event count (approx.): 6394890089
  Children      Self  Command   Shared Object      Symbol
+   99,37%     0,00%  mini_ep3  mini_ep3           [.] main
+   99,32%     0,00%  mini_ep3  libc-2.31.so       [.] __libc_start_main
+   52,35%    52,35%  mini_ep3  mini_ep3           [.] fibonacci
+   47,62%    47,58%  mini_ep3  mini_ep3           [.] slowsort
```

Um grande gargalo observado se concentra nas chamadas recursivas dos algoritmos de slowsort e fibonacci.

2 Parte II - Fibonacci e a DP Memoizada

De modo geral, a sequência de Fibonacci pode ser facilmente computada por meios recursivos. E, talvez, esta seja a forma mais didática de ensino da computação dessa instigante sequência numérica.

Contudo, em razão de sua natureza FORTEMENTE recursiva, o algoritmo pode ser tratado como extremamente ineficiente. Note que o cômputo do n-ésimo fibonaccico exige os valores do $n-1$ e do $n-2$-ésimos fibonaccicos anteriores. E estes dependem dos seus antecessores. Então, para um n expressivamente grande, a recursão exige o retorno até a base n = 1 e n = 0 para estruturar os cálculos, carregando consigo uma árvore recursiva pesada.

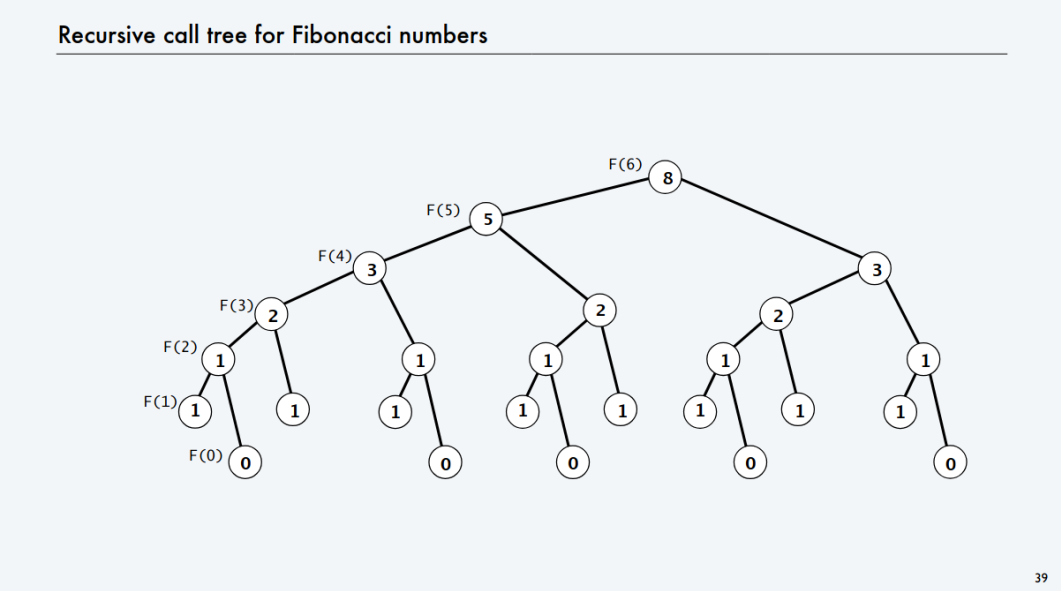

Figura 1: Fib(6)

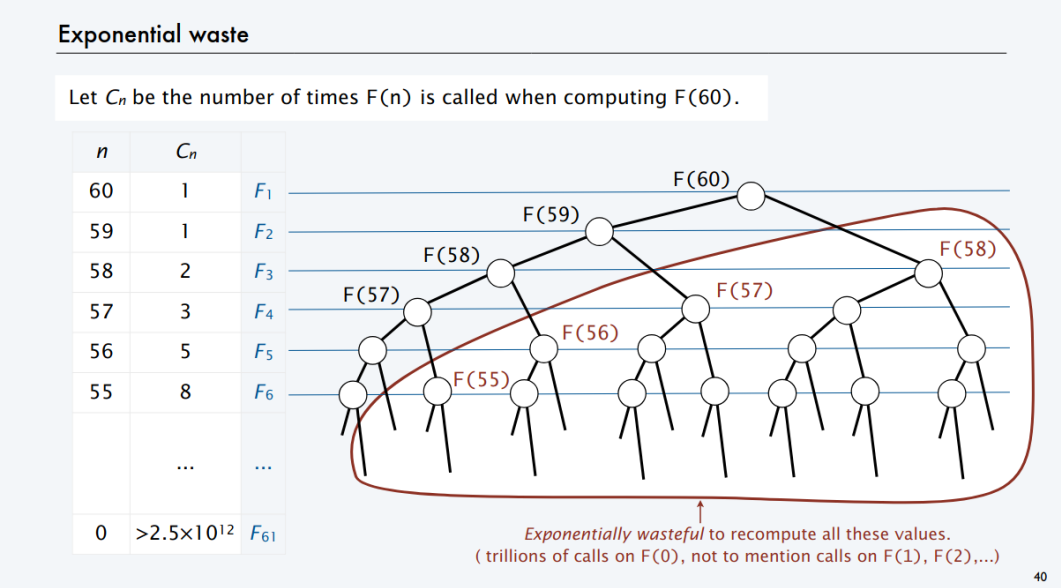

Figura 2: F(60)

As figuras acima ilustram que muito cômputo é repetido e, também, muito desperdiçado. Então, uma solução de otimização é o reaproveitamento de recursos.

Para este fim, podemos aplicar a DP memoizada, em que, para cada fibonaccico, o armazenamos numa posição de um array de memoização. E, antes de computar os valores do $n-1$ e do $n-2$-ésimos fibonaccicos anteriores para Fib(n), verificamos se estes já existem no nosso array. Reduzimos o número de chamadas recursivas e de processamento computacional.

Ademais, pré-computamos, à mão, alguns valores de base para auxiliar e reduzir cálculos.

3 Parte III - S ...L ...O ...W ...S ...O ...R ...T

De nome fortemente sugestivo, o slowsort é um algortimo **FORTEMENTE** recursivo. Apesar da aplicação do algoritmo de "Divisão e Conquista"(praticada pelos melhores sorts como MergeSort e QuickSort), sua forma de ordenação é extremamente ineficiente. Este algoritmo ordena dois elemento por chamada recursiva e aplica uma chamada de complexidade $T(n-1)$. (O principal mal desse algoritmo)

Em outras palavras, o algoritmo tem complexidade $T(n) = 2T(n/2) + T(n-1) + 1$, sendo não polinomial e sendo pior que o BubbleSort no melhor caso.

Nesse viés, para garantir um bom desempenho, podemos trocar o SlowSort pelo QuickSort 3-way-partition, que garante, no tempo médio, um desempenho assintótico de $O(nlg(n))$ e que lida bem com repetições. Afinal, no fim, ambos os sorts desempenham a mesma finalidade.

Evitamos as pesadas chamadas recursivas e processamento desnecessário.

4 Parte IV - BitWise

Apesar de quase silencioso, a operação de resto por 2 pode facilmente ser substituída por sua versão bitwise (um pouco mais eficiente).

5 Parte V - O voto de Minerva

Finalmente, aplicando o gprof

```
$ gcc -pg -g mini_ep3_ot.c -o mini_ep3_ot
$ ./mini_ep3_ot
$ gprof ./mini_ep3_ot
```

Conseguimos o seguinte diagnóstico:

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       31     0.00     0.00  partition
  0.00      0.00     0.00       30     0.00     0.00  fibonacci
  0.00      0.00     0.00        1     0.00     0.00  quicksort
```

```
Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00      31/31          quicksort [3]
```

```
[1]      0.0    0.00    0.00      31          partition [1]
-----------------------------------------------
                                  62              fibonacci [2]
                0.00    0.00      30/30           main [9]
[2]      0.0    0.00    0.00      30+62       fibonacci [2]
                                  62              fibonacci [2]
-----------------------------------------------
                                  62              quicksort [3]
                0.00    0.00       1/1            main [9]
[3]      0.0    0.00    0.00       1+62        quicksort [3]
                0.00    0.00      31/31           partition [1]
                                  62              quicksort [3]
-----------------------------------------------

Index by function name

   [2] fibonacci               [1] partition               [3] quicksort
```

E quanto ao perf, nenhum dos eventos mais críticos se referem às funções mais pesadas e problemáticas.

Portanto, reduzimos a quantidade de recursões e reciclamos computações, gerando um programa de mesmo resultado, mas, mais eficiente.