Atividade Prática 02

Aluno: Raphael Henrique Braga Leivas

Matrícula: 2020028101

1 OBJETIVO

O objetivo dessa atividade prática é avaliar e comparar o desempenho de códigos recursivos e iterativos para a execução de uma mesma tarefa.

Além disso, durante as atividades, serão desenvolvidas habilidades e conhecimentos para mensurar o desempenho de um código em C - isto é, quanto tempo é gasto para compilar e executar o código.

2 METODOLOGIA

2.1 CÓDIGOS DESENVOLVIDOS

Foram codificadas 4 funções, disponíveis nas bibliotecas de autoria própria fibonacci.h e factorial.h.

```
unsigned int getRecursiveFactorial(unsigned int n);
unsigned int getIteractiveFactorial(unsigned int n);
unsigned int getRecursiveFibonacci(unsigned int n);
unsigned int getIteractiveFibonacci(unsigned int n);
```

As especificações de cada uma delas estão descritas abaixo.

- getRecursiveFactorial: retorna o fatorial $n!$ do inteiro n passado no argumento. O cálculo é feito via recursões;
- getIteractiveFactorial: retorna o fatorial $n!$ do inteiro n passado no argumento. O cálculo é feito via métodos iterativos;
- getRecursiveFibonacci: retorna o n-ésimo termo da sequência de Fibonacci via métodos recursivos;
- getIteractiveFibonacci: retorna o n-ésimo termo da sequência de Fibonacci via métodos iterativos.

Além dessas funções, foram desenvolvidas mais três funções para o cálculo de tempo de execução. Elas estão disponíveis na biblioteca de autoria própria customTime.h.

```
int64_t getUnixTimestamp(int option);
float getUserTimeDiff(struct rusage *start, struct rusage *end);
float getSystemTimeDiff(struct rusage *start, struct rusage *end);
```

As especificações de cada uma delas estão descritas abaixo.

- getUnixTimestamp: retorna o número de milissegundos, microssegundos ou nanosegundos desde a época Unix (1 de janeiro de 1970), conforme o argumento option;
- getUserTimeDiff: retorna o tempo de usuário transcorrido entre dois objetos rusage;
- getSystemTimeDiff: retorna o tempo de sistema transcorrido entre dois objetos rusage.

Todo o código usado está disponível na pasta .zip enviada pelo moodle.

2.2 PLANO DE EXPERIMENTOS

Para testar o desempenho do algoritmo de Fibonacci, o vetor A foi arbitrariamente escolhido para ser um vetor de 10 inteiros, dado por

$$A = [1, 3, 5, 10, 15, 20, 25, 30, 35, 40]$$

Esse valores de A permitem testar uma faixa de tempo de execução variada, sem tomar tempo demais esperando códigos muito lentos (por exemplo: termo de posição 80 da sequência de Fibonacci) terminarem.

Para o teste de desempenho do algoritmo do fatorial, é usado o vetor B dado por

$$B = 100\,A = [100, 300, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000]$$

Como o algoritmo de recursividade do fatorial é menos complexo que o algoritmo recursivo de Fibonacci (veja seção de Discussão para mais detalhes), usamos valores maiores para ver diferenças mais expressivas nos resultados.

O plano de experimentos consiste em iterar sobre todos os elementos do vetor A ou B, e em cada iteração será calculado o i-ésimo termo da sequência de fibonacci ou o valor de $i!$ com as funções exibidas acima. Para cada operação, será calculado os seguidos valores de tempo transcorridos desde o início e fim da chamada da função que calcula o Fibonacci ou o Fatorial:

- Tempo de relógio transcorrido: a diferença em microssegundos entre o início da operação e o fim da operação. Calculado via função getUnixTimestamp();
- Tempo de usuário transcorrido: a diferença em microssegundos entre o tempo de usuário do início da operação e o fim da operação. Calculado via função getUserTimeDiff();
- Tempo de sistema transcorrido: a diferença em microssegundos entre o tempo de sistema do início da operação e o fim da operação. Calculado via função getSystemTimeDiff();

É importante destacar a diferença entre o tempo de sistema e o tempo de usuário. O tempo de usuário corresponde à quantidade de tempo que o programa gastou efetuando cálculos e operações no código, enquanto o tempo de sistema é a quantidade de tempo que o programa ficou esperando o kernel do sistema operacional responder e efetuar operações de sistema, tais como printar conteúdos na tela, ler arquivos salvos em disco, etc. Assim, o tempo total gasto pela operação será a soma do tempo de usuário com o tempo de sistema.

3 RESULTADOS

A Tabela 1 mostra os resultados obtidos para os cálculos de Fibonacci, enquanto a Tabela 2 mostra os resultados obtidos para os cálculos de Fatorial.

Tabela 1. Medições de tempo para os algoritmos recursivo e iterativo de Fibonacci.

i	Tempo de relógio iterativo (µs)	Tempo de usuário iterativo (µs)	Tempo de sistema iterativo (µs)	Tempo de relógio recursivo (µs)	Tempo de usuário recursivo (µs)	Tempo de sistema recursivo (µs)
1	0	0	0	0	0	0
3	0	0	0	0	1	0
5	1	0	0	0	1	0
10	0	0	0	1	2	0
15	0	0	0	16	18	0
20	0	1	0	184	255	0
25	0	0	0	2182	2212	0
30	0	1	0	23070	22792	0
35	0	0	0	257664	299535	0
40	0	0	0	3003337	2806024	0

Fonte: elaboração própria.

Tabela 2. Medições de tempo para os algoritmos recursivo e iterativo de cálculo do Fatorial.

i	Tempo de relógio iterativo (µs)	Tempo de usuário iterativo (µs)	Tempo de sistema iterativo (µs)	Tempo de relógio recursivo (µs)	Tempo de usuário recursivo (µs)	Tempo de sistema recursivo (µs)
100	0	1	0	1	2	0
300	1	1	0	6	6	0
500	1	2	0	7	10	0
1000	2	2	0	16	18	0
1500	2	3	0	25	27	0
2000	4	4	0	31	35	0
2500	4	5	0	37	42	0
3000	6	7	0	44	49	0
3500	7	8	0	51	57	0
4000	8	9	0	56	64	0

Fonte: elaboração própria.

4 DISCUSSÃO

O fato do tempo de sistema ser zero em todos os casos é condizente com o esperado: as funções de Fibonacci e fatorial não executam chamadas de sistema e apenas executam operações simples de cálculo, não havendo leitura de dados em disco ou operações com o display da máquina que possam exigir ações do sistema operacional.

É interessante notar que o algoritmo iterativo é mais rápido que o algoritmo recursivo, especialmente no caso de Fibonacci. De fato, os algoritmos iterativos possuem apenas um loop em sua execução, e assim possuem complexidade linear $O(n)$. No entanto, o algoritmo recursivo de Fibonacci possui complexidade exponencial de $O\left(\varphi^{n}\right)$, onde $\varphi = 1.6180$ é o número da razão de ouro. Assim, o tempo gasto aumenta

exponencialmente com o aumento do argumento de entrada, e com $i = 40$ o algoritmo já gasta mais de 3 segundos para ser executado e retornar a resposta.

No caso do algoritmo recursivo de fatorial, ele também possui complexidade linear $O(n)$, mas ele gasta mais tempo que o interativo (que tem mesma complexidade) pois as chamadas recursivas de função gastam mais tempo na stack do programa principal que um simples loop sobre um vetor, que no final contribuem para a diferença de tempo observada.

É interessante notar que, no algoritmo recursivo de fatorial, caso seja adicionado uma operação intensa (por exemplo, calcular o seno de 1 a 1000 em cada chamada recursiva), o tempo de execução aumenta significativamente, como mostra a Tabela 3.

Tabela 3. Comparação de tempo de relógio do algoritmo fatorial recurso com e sem uma operação intensa em cada chamada.

i	Tempo de relógio normal (μs)	Tempo de relógio com operação intensa (μs)
100	1	217
300	6	658
500	7	1018
1000	16	1938
1500	25	2952
2000	31	3891
2500	37	7072
3000	44	6279
3500	51	7176
4000	56	7756

Fonte: elaboração própria.

Por fim, é importante lembrar que a própria medição gasta um tempo para ser executada e causa uma interferência, mas ela não foi significativa pois os valores medidos estão conforme o esperado. Portanto, nessa aplicação, podemos concluir que essa interferência pode ser desprezada.

5 REFERÊNCIAS

GEEKS FOR GEEKS. Time Complexity of Recursive Fibonacci Program (2017). Disponível em: <https://www.geeksforgeeks.org/time-complexity-recursive-fibonacci-program/>. Acesso em 30 de mar. de 2023.