MAC0121 - Estruturas de Dados Eduardo Hashimoto nUSP 6514136

Foram criadas duas funções para executar a tarefa.

1) A função **collatz** cálculo o número de passos necessários para se chegar a situação estacionária (n = 1) dado um determinado inteiro positivo.

Observações:

- não há testes para verificar se o valor dado é válido;
- a função dá erros caso o valor não seja válido;

2) A segunda função, **test_collatz**, é uma função auxiliar que executa a função collatz para os números entre o intervalo especificado e imprimi o resultado.

Observa-se em relação a distribuição do número de passos para chegar ao 1:

- os números de passos costumam se repetir para diferentes valores de entrada;
- entradas potência de 2 consomem poucos passos (exatamente o valor da potência, obviamente);
- para a grande maioria dos números não é preciso mais que 100 passos (sendo que a distribuição vai aumentando em direção à 100);

A primeira tentativa para executar a tarefa designada está representada abaixo:

```
void test_collatz(int i, int f){
    for ( ;i < f; i++) {
        int p = collatz(i);
        printf("para o número %d foram necessários %d passos\n", i, p);
    }
}
```

1º algoritmo - função collatz

```
int collatz(int n){
    int passo = 0;
    while (n != 1){
        if (n % 2) n = 3*n + 1;
        else
            n = n/2;
        passo++;
    }
    return passo;
}
```

Esse algoritmo é a implementação mais direta da especificação do problema.

Se, no entanto, notarmos que se um número é impar o próximo passo terá um número par (pois se n é impar, n = 2*k+1 e então 3*n+1 = 3*(2*k+1)+1 = 6*k+4 = 2*(3*k+2) e portanto par). Se no entanto, o número for par pode se ter um número par ou impar no passo seguinte (pois se n é par, n = 2*k e então 2*k/2 = k, mas k é um inteiro positivo). Isso nos permite 'pular' um passo, tornando o algoritmo ligeiramente mais eficiente.

2º algoritmo - função collatz

```
int collatz(int n){
    int passo = 0;
    while (n != 1){
        if (n % 2) {
            n = (3*n + 1)/2;
            passo += 2;
        }
        else {
            n /= 2;
            passo++;
        }
    }
    return passo;
}
```

Podemos, então, repetir as ações enquanto elas satisfazerem uma das condição de paridade.

3º algoritmo - função collatz

```
int collatz(int n){
    int passo = 0;
    while (n != 1){
        while(n % 2) {
            n = (3*n + 1)/2;
            passo += 2;
        }
        while (!(n%2)){
            n /= 2;
            passo++;
        }
    }
    return passo;
}
```

Melhorado o algoritmo da função **collatz**, podemos pensar em como melhorar o algoritmo da função **test_collatz**. Observa-se que os resultados da função **collatz** são descartados após serem impressos e, portanto, não há aproveitamento do fato de já se ter computado o número de passos para determinada entrada. Assim, uma maneira de melhorar a eficiência desse algoritmo seria armazenar os resultados

computados de modo que um eventual número que recaia num caso conhecido tenha seu número de passos calculados mais rapidamente.

1º Algoritmo - função test_collatz

```
void test_collatz(int i, int f){

    for ( ;i < f; i++){
        int p = collatz(i);
        printf("Para o número %d foram necessários %d passos\n", i, p);
    }
}
```

Essa implementação é a mais direta e não usa o fato mencionado anteriormente.

Devemos mencionar também que para números muito grandes, o número n da função **collatz** chega a valores muito maiores. Para resolver isso, basta usar o tipo *long* ao invés de *int*, que pode armazenar números muito maiores.

2º Algoritmo - função collatz e test_collatz

```
int collatz(long n, int *cache){

long num = n;

int passos = 0;

while (num != 1){

    if (num % 2 == 0)
        num /= 2;
    else
        num = 3*num + 1;

    passos++;

    if (num < n){
        passos += cache[num];
        break;
    }
}

cache[n] = passos;

return passos;
}
```

.

```
void test_collatz(int i, int f){

int *cache;
cache = malloc(f*sizeof(int));

for (; i < f; i++) {
    int p = collatz( (long) i, cache);
    // printf("número: %d : passos: %d\n", i, p);
}

free(cache);
```

}

Para levar em consideração os números já calculados e sobre os quais outros números podem recair, foram feitas alterações nas duas funções.

A função **collatz** passa a receber um ponteiro que armazena o número de passos para cada número testado. Esse ponteiro será acessado toda vez que a função recair sobre um número que já foi calculado. Como devemos verificar o mais rapidamente possível se já temos o número armazenado no ponteiro o laço das condições de paridade voltaram a ser uma condicional.

Há um ganho de eficiência em relação a função sem o ponteiro, como se pode observar nos resultados abaixo:

i f tcom tsem

1 100000000 6.5 65.0

1 10000000 2.8 8.6

1 1000000 2.5 3.0

1 100000 2.4 2.6

Contudo, esse algoritmo apresenta uma falha em relação a especificação: ele funciona se i for igual a 1.

Para poder usá-lo com qualquer intervalo, deve-se pré-calcular todos os resultados até i, levando ao algoritmo final abaixo.

```
int collatz(long n, int *cache){

long num = n;

int passos = 0;

while (num != 1){

    if (num % 2 == 0)
        num /= 2;
    else
        num = 3*num + 1;

    passos++;

    if (num < n){
        passos += cache[num];
        break;
    }
}

cache[n] = passos;

return passos;
}
```

.

```
void test_collatz(long i, long f){

int *cache;
cache = malloc(f*sizeof(int));

long k;
for (k = 1; k < i; k++)
    collatz(k, cache);

for (; i < f; i++) {
    int p = collatz(i, cache);
    // printf("número: %d : passos: %d\n", i, p);
}

free(cache);
}
```

O "overhead" de se calcular previamente os números até i se constatou inferior ao custo de não se armazenar nenhum resultado, conforme expõe a tabela abaixo:

i f tcom tsem

100000 1000000 2.4 3.0

1000000 10000000 2.9 8.1

10000000 100000000 6.5 64.7

O máximo intervalo que consegui resolver foi de 1 a 500 milhões para o qual ele levou 76 segundos.

Portanto, este último algoritmo apresentado foi o que melhor consegui fazer para cumprir a especificação do EP1.

Observações finais

O resultado de alguns números foram conferidos contra valores encontrados em sites. Por exemplo, o número 270271 precisa de 406 passos para chegar a 1.

O programa foi compilado e executado no editor Sublime com o seguinte Build System:

*{ "cmd" : ["gcc", "$file*name", "-o", "${file*base*name}", "-lm", "-Wall", "-ansi", "-pedantic", "-O2"], "selector" : "source.c", "shell":false, "working*dir" : "$file_path",

"variants": [{ "name": "Run", "cmd": ["bash", "-c", "/usr/bin/gcc '${file}' -Wall -o '${file*path}/${file*base*name}' && '${file*path}/${file*base*name}'"] }] }*