Avaliação 1

Luiz Fernando Rabelo – 11796893

1. Análise Algoritmo Original

Abaixo, com algumas adaptações (a fim de facilitar a contagem), estão representadas as funções main e funcao da implementação original:

```
int main(int argc, char *argv[]){
    int x, n;
    scanf("%d", &n);  // 1 atribuição(1a)
    for(int i = 0; i < n; i++){
        scanf("%d", &x);  // 1 atribuição (1a)
        int p = 1, s = 0;  // 2 atribuições (2a)
        funcao(x, x-1, &p, &s);
        if(p == 1) printf("%d Primo\n", x);  // 1 comparação (1c)
        else printf("%d Nao primo\n", x);
    }
    return 0;
}
```

```
void funcao(int x, int y, int *primo, int *soma){
    if(y <= 1) return;  // 1 comparação (1c)
    if(x % y == 0){  // 1 comparação (1c)
        *primo = 0;  // 1 atribuição (1a)
    }
    for(int a = 0; a <= 10000; a++){
        if (a % y == 0) *soma += a;  // 1 comparação e 1 atribuição (1c + 1a)
    }
    funcao(x, y-1, primo, soma);
}
```

Nota-se, logo no início da função funcao, 2 comparações e 1 atribuição, ou seja $2c + a$ operações. Após isso, o escopo do laço for é executado 10001 vezes, o que nos dá $10001 \cdot (c + a)$ operações.

A função se repetirá recursivamente até chegar ao caso base $f(1) = 1$, em que há apenas uma comparação. Assim, se consideramos $u = 2c + a + 10001 \cdot (c + a)$ em função de y, teremos que:

$$f(y) = u + f(y - 1)$$

$$f(y-1) = u + f(y-2)$$
$$f(y-2) = u + f(y-3)$$
$$f(y-3) = u + f(y-4)$$
$$\cdots$$

Dessa forma, para a k-ésima chamada da função, existirão $k \cdot u + f(y-k)$ operações. Tomando o caso base, temos que $y - k = 1 => k = y - 1$. Assim, por substituição, temos:

$$f(y) = (y-1) \cdot u + \ f(1)$$
$$f(y) = (y-1) \cdot (2c + a + 10001c + 10001a) + 1$$
$$f(y) = (y-1) \cdot (10003c + 10002a) + 1$$

Na primeira chamada da função, $y = x - 1$. Dessa forma, podemos escrever o resultado encontrado em função de x: $f(x) = (x-2) \cdot (10003c + 10002a) + 1$.

Já na função main, após a atribuição inicial à variável n, o laço de repetição for faz com que os comandos em seu escopo sejam executados n vezes, sendo n o número total de números primos a serem analisados.

Dessa forma, como $f(x)$ representa o número de operações realizados pela função funcao e como existem outras 3 atribuições e 1 comparação no escopo do laço for, temos:

$$f(n) = n \cdot (3a + c + f(x))$$
$$f(n) = n \cdot [3a + c + (x-2) \cdot (10003c + 10002a) + 1] + \text{a}$$

2. Análise Algoritmo Otimizado

O algoritmo anterior, além de consumir bastante tempo de execução, apresenta um erro: considera que todos os números menores que 2 são primos. Isso é corrigido no novo algoritmo, cujas funções estão apresentadas abaixo:

```
int main(int argc, char *argv[]){
      int n;
      scanf("%d", &n);  // 1 atribuição (1a)
      for(int i = 0, x = 0; i < n; i++){
            scanf("%d", &x);  // 1 atribuição (1a)
            int p = 1;  // 1 atribuiçãp (1a)
            funcao(x, x-1, &p);
            if(p == 1) printf("%d Primo\n", x);  // 1 comparação (1c)
            else printf("%d Nao primo\n", x);
   }
   return 0;
}
```

```c
void funcao(int x, int y, int *primo){
	if(x < 2) *primo = 0;  // 1 comparação (1c)
	else if(x == 2) *primo = 1;  //1 comparação (1c)
	else verificaPrimo(x, y, primo);
}
```

```c
void verificaPrimo(int x, int y, int *primo){
	if(y <= 1 || *primo == 0) return; // 2 comparações (2c)
	if(x % y == 0) *primo = 0; // 1 comparação e 1 atribuição (1c + 1a)
	verificaPrimo(x, y-1, primo);
}
```

Considerando o pior caso, são executadas as 2 comparações na função funcao e a função verificaPrimo é chamada recursivamente, até que se chegue no caso base $f(1) = 2$ em que são executadas 2 comparações.

Assim, se tomarmos $u = 3c + a$ como as 3 comparações e 1 atribuição existentes na função verificaPrimo, teremos que:

$$f(y) = u + f(y-1)$$
$$f(y-1) = u + f(y-2)$$
$$f(y-2) = u + f(y-3)$$
$$f(y-3) = u + f(y-4)$$
$$\cdots$$

Dessa forma, para a k-ésima chamada da função, existirão $k \cdot u + f(y-k)$ operações. Tomando o caso base, temos que $f(y-k) = 1 => k = y - 1$. Assim, por substituição, temos:

$$f(y) = (y-1) \cdot u + f(1)$$
$$f(y) = (y-1) \cdot (3c + a) + 2$$

Da mesma maneira que escrevemos $f(y)$ em função de x no algoritmo original, podemos escrever $f(y)$ em função de x no algoritmo otimizado, pois na primeira chamada da função $y = x - 1$. Assim: $f(x) = (x-2) \cdot (3c + a) + 2$.

Na função main, o escopo do laço for(o qual possui duas atribuições, 1 comparação e a função funcao) é executado n vezes, sendo n o total de números a serem analisados.

Portanto, considerando a atribuição inicial de n na função main como a; as operações de atribuição e comparação no escopo do laço for como v; as duas comparações na função funcao como w e o número de operações da função verificaPrimo como $f(x)$, temos:

$$f(n) = n \cdot [v + w + f(x)] + a$$
$$f(n) = n.[2a + c + 2c + (x-2) \cdot (3c + a) + 2] + a$$

$$f(n) = n \cdot [2a + 3c + (x - 2) \cdot (3c + a) + 2] + a$$

Pode-se notar uma pequena melhora na eficiência do algoritmo otimizado para o pior caso. Entretanto, há uma melhoria bastante significativa no algoritmo otimizado para os casos em que o número x buscado não é primo pois, para qualquer divisor encontrado entre 1 e x, o algoritmo otimizado retorna, na próxima execução, o valor 0, enquanto o algoritmo original ainda percorreria o restante dos divisores até chegar em 1 para depois retornar 0.

3. Comparação de Tempo Casos de Teste:

Figura 1: Casos e Tempo de Execução – Algoritmo Original

Resultado dos Casos:

Caso	Status	Tempo de CPU	Mensagem
Caso 1	Correto	0.0018 s	Resposta Correta
Caso 2	Correto	0.0063 s	Resposta Correta
Caso 3	Correto	0.7203 s	Resposta Correta
Caso 4	Incorreto	1.0007 s	Tempo de Execução Excedido

Figura 2: Casos e Tempo de Execução – Algoritmo Otimizado

Resultado dos Casos:

Caso	Status	Tempo de CPU	Mensagem
Caso 1	Correto	0.0011 s	Resposta Correta
Caso 2	Correto	0.0010 s	Resposta Correta
Caso 3	Correto	0.0018 s	Resposta Correta
Caso 4	Correto	0.0029 s	Resposta Correta