Relatório - Aula 2
Giovanna Naves Ribeiro
2022043647

Começamos o processo criando as pastas e escrevendo os códigos para Fibonacci iterativo, Fibonacci recursivo, fatorial iterativo e fatorial recursivo nos arquivos fib.c e fat.c.
O arquivo fatop.c executa as operações recursiva e iterativa de fatorial e o fibop.c executa as operações recursiva e iterativa de Fibonacci.

Após organizar os códigos, testamos cada um deles na faixa de valores de 1 a 55 e imprimimos o tempo de execução de usuário, de sistema e de relógio (respectivamente) para cada um dos valores da faixa testada.

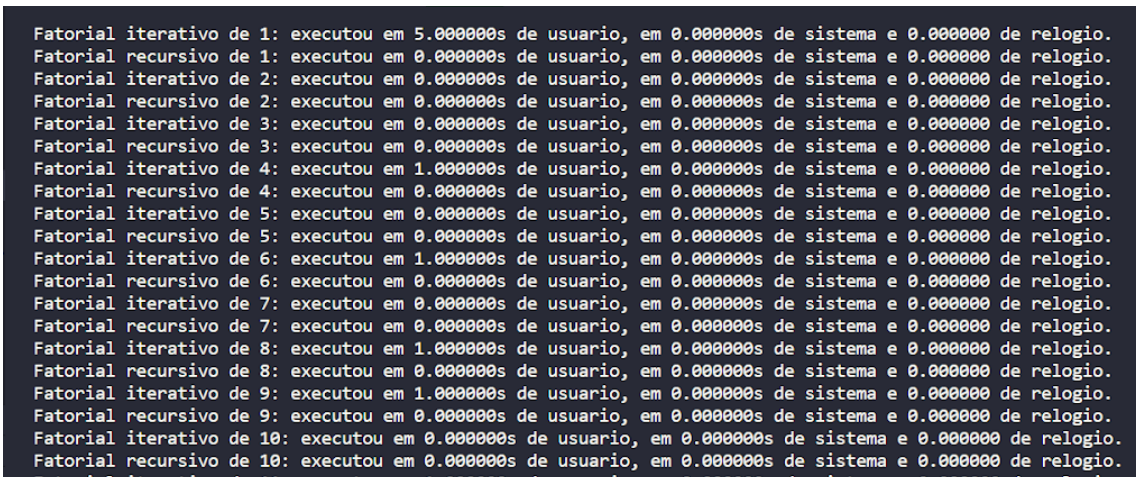
Linhas de comando:

- make fat
- make fib

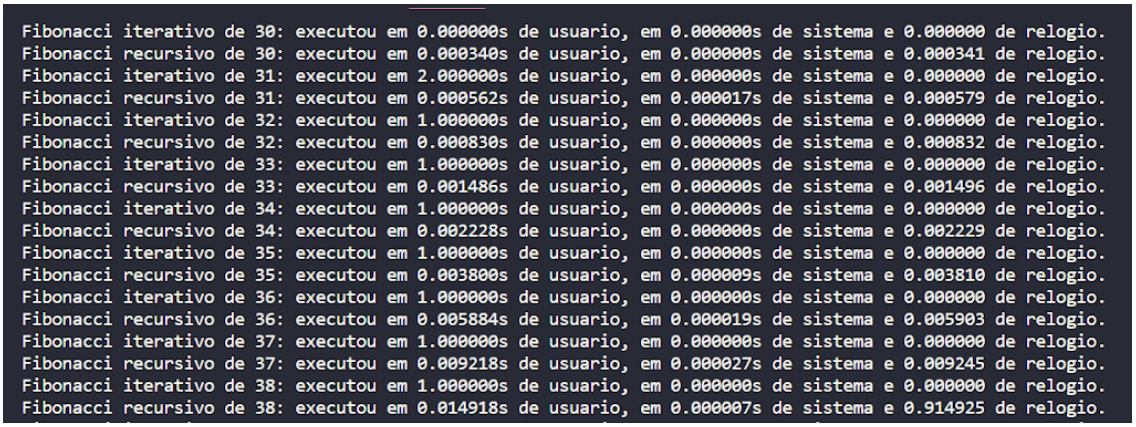
```
CC = c
LIBS = -lm
SRC = src
OBJ = obj
INC = include
BIN = bin
OBJSFAT = $(OBJ)/fatop.o $(OBJ)/fat.o
HDRSFAT = $(INC)/fat.h
OBJSFIB = $(OBJ)/fibop.o $(OBJ)/fib.o
HDRSFIB = $(INC)/fib.h
CFLAGS = -Wall -c -pg -ftest-coverage -I$(INC)

EXEFAT = $(BIN)/fatop
EXEFIB = $(BIN)/fibop

fat:  $(EXEFAT)
    $(EXEFAT)

fib:  $(EXEFIB)
    $(EXEFIB)


$(BIN)/fatop: $(OBJSFAT)
    $(CC) -pg -o $(BIN)/fatop $(OBJSFAT) $(LIBS) -pg
      You, yesterday • updates
$(OBJ)/fatop.o: $(HDRSFAT) $(SRC)/fatop.c
    $(CC) $(CFLAGS) -o $(OBJ)/fatop.o $(SRC)/fatop.c

$(OBJ)/fat.o: $(HDRSFAT) $(SRC)/fat.c
    $(CC) $(CFLAGS) -o $(OBJ)/fat.o $(SRC)/fat.c


$(BIN)/fibop: $(OBJSFIB)
    $(CC) -pg -o $(BIN)/fibop $(OBJSFIB) $(LIBS) -pg

$(OBJ)/fibop.o: $(HDRSFIB) $(SRC)/fibop.c
    $(CC) $(CFLAGS) -o $(OBJ)/fibop.o $(SRC)/fibop.c

$(OBJ)/fib.o: $(HDRSFIB) $(SRC)/fib.c
    $(CC) $(CFLAGS) -o $(OBJ)/fib.o $(SRC)/fib.c


cleanfat:
    rm -f $(EXE) $(OBJSFAT) gmon.out

cleanfib:
    rm -f $(EXE) $(OBJSFIB) gmon.out
```

Segue o resultado para alguns valores dos testes:

Fatorial iterativo e recursivo de 1 a 10:

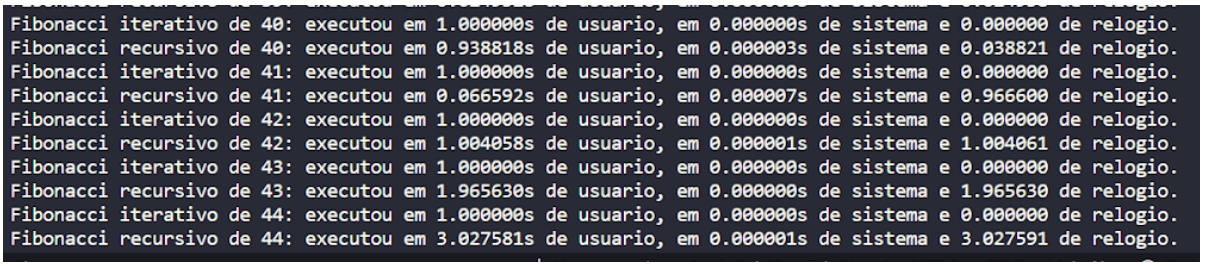


Fibonacci iterativo e recursivo de 30 a 38:



Observação 1:

Podemos perceber, pelo tempo de execução dos algoritmos de Fibonacci recursivo e iterativo, que a partir do número 42, o algoritmo recursivo se mostra menos eficiente em tempo que o iterativo para encontrar o resultado. Como é possível analisar abaixo, se assumirmos TI e TR como os tempos de usuário para os algoritmos iterativo e recursivo respectivamente, TI>TR para números menores que 42. Caso o número seja maior ou igual a 42, TI<TR.



Observação 2:

Apesar de os tempos de sistema e de relógio do algoritmo de Fibonacci tenderem a zero para os primeiros números dos testes, a partir do número 17, esses tempos se mostram maiores para o algoritmo recursivo, se tornando significativamente maiores nos últimos números de teste.
Já nos algoritmos recursivo e iterativo de fatorial, os tempos de sistema e de relógio foram de 0 segundos para todos os testes.

Observação 3:

O tempo de usuário se mostrou maior que o tempo de sistema e de relógio na grande maioria dos testes, com a única exceção dos testes de Fibonacci recursivo, em que os dois mostraram ter o mesmo valor.

Em seguida, usamos o comando `gprof bin/fatop gmon.out > analise.txt` para rodar o gprof e gerar um relatório de execução.
Entretanto, os tempos fornecidos pelo gprof foram 0 para todas as operações executadas.

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
 0.00      0.00     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

```
index % time    self  children    called     name
                0.00    0.00       1/1           _GLOBAL__sub_I_main [10]
[8]      0.0    0.00    0.00       1         __static_initialization_and_destruction_0(int, int) [8]
-----------------------------------------------
```

Não cheguei à conclusão de se isso aconteceu por um problema de ajuste numérico do computador ou se por o gprof não estar funcionando da maneira adequada. Logo, não foi possível fazer uma comparação dos tempos como pedido no item 5 e nem se mostrou útil fazer a chamada recursiva de uma função que consuma recursos computacionais, já que não seria possível comparar os tempos de execução.

Em seguida, o próximo passo foi adequar o código para que o comando `make run` funcionasse como `make fat` e `make fib` ao mesmo tempo. Dessa maneira, o novo makefile fica como a imagem abaixo. Além disso, adaptamos o código para que ele executasse apenas o fatorial e o Fibonacci iterativos e recursivos de 5 e medisse seu valor.

```
CC = gcc
LIBS = -lm
SRC = src
OBJ = obj
INC = include
BIN = bin
OBJSFAT = $(OBJ)/fatop.o $(OBJ)/fat.o
HDRSFAT = $(INC)/fat.h
OBJSFIB = $(OBJ)/fibop.o $(OBJ)/fib.o
HDRSFIB = $(INC)/fib.h
CFLAGS = -Wall -c -pg -ftest-coverage -I$(INC)

EXE = $(BIN)/fatop $(BIN)/fibop

run: $(EXE)
    $(BIN)/fatop
    $(BIN)/fibop      You, now • Uncommitted changes

$(BIN)/fatop: $(OBJSFAT)
    $(CC) -pg -o $(BIN)/fatop $(OBJSFAT) $(LIBS) -pg

$(OBJ)/fatop.o: $(HDRSFAT) $(SRC)/fatop.c
    $(CC) $(CFLAGS) -o $(OBJ)/fatop.o $(SRC)/fatop.c

$(OBJ)/fat.o: $(HDRSFAT) $(SRC)/fat.c
    $(CC) $(CFLAGS) -o $(OBJ)/fat.o $(SRC)/fat.c


$(BIN)/fibop: $(OBJSFIB)
    $(CC) -pg -o $(BIN)/fibop $(OBJSFIB) $(LIBS) -pg

$(OBJ)/fibop.o: $(HDRSFIB) $(SRC)/fibop.c
    $(CC) $(CFLAGS) -o $(OBJ)/fibop.o $(SRC)/fibop.c

$(OBJ)/fib.o: $(HDRSFIB) $(SRC)/fib.c
    $(CC) $(CFLAGS) -o $(OBJ)/fib.o $(SRC)/fib.c


cleanfat:
    rm -f $(EXE) $(OBJSFAT) gmon.out

cleanfib:
    rm -f $(EXE) $(OBJSFIB) gmon.out
```

```
bin/fatop
Fatorial de 5: 120.
Fatorial iterativo de 5: executou em 7.000000s de usuario, em 0.000000s de sistema e 0.000000 de relogio.
Fatorial recursivo de 5: executou em 0.000000s de usuario, em 0.000000s de sistema e 0.000000 de relogio.
bin/fibop
Fibonacci de 5: 5.
Fibonacci iterativo de 5: executou em 5.000000s de usuario, em 0.000000s de sistema e 0.000000 de relogio.
Fibonacci recursivo de 5: executou em 0.000000s de usuario, em 0.000000s de sistema e 0.000000 de relogio.
```

Repare que, para números pequenos, os algoritmos recursivos são consideravelmente mais eficientes que os iterativos.