підготував старший викладач каф. ЕОМ Козак Назар Богданович

Домашнє завдання №9

Скласти програму (C/C++), яка дозволяє перевірити для заданої ***L(n)*** такі два твердження:

а)	$L(n) = \Theta(n^2)$
б)	$L(n) \neq \Theta(n^3)$

Вибір варіанту

(Nж + Nг + 1) % 3 + 1

де: Nж – порядковий номер студента в групі, а Nг – номер групи(1,2,3,4,5,6,7,8 або 9)

Варіанти завдань

Варіант	L(n)
1	$\frac{n^2}{4} - \frac{n}{8}$
2	$\frac{n^2}{8} - \frac{n}{16}$
3	$\frac{n^2}{16} - \frac{n}{32}$

Приклад коду

Наведений зразок коду демонструє виконання завдання для:

L(n)
$\frac{n^2}{2} - \frac{n}{4}$

Значення задані за допомогою наступного макросу:

```
#define COMPUTATIONAL_COMPLEXITY_POLYNOMIAL { { 0.5, 2 }, { -0.25, 1 } }
```

Лістинг

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"

#define COMPUTATIONAL_COMPLEXITY_POLYNOMIAL { { 0.5, 2 }, { -0.25, 1 } }
#define TERM_O_DEGREE__A 2
#define TERM_O_DEGREE__B 3

#define TERM_O__A { 1, TERM_O_DEGREE__A}
#define TERM_O__B { 1, TERM_O_DEGREE__B}

#define MAX_POLYNOMIAL 128

#define OP_SUCCESS 0
#define OP_FAILED -1

typedef struct Term_ {
  long double coefficient;
  long long int degree;
} Term;
typedef Term Polynomial[MAX_POLYNOMIAL];

char polynomialDivideByTerm(Polynomial polynomial, Term term) {
  if (!term.coefficient) {
    printf("Error: try to divide by zero-term\r\n");
    return OP_FAILED;
  }

  for (unsigned int index = 0; polynomial[index].coefficient && index <
MAX_POLYNOMIAL; ++index) {
    polynomial[index].degree -= term.degree;
    polynomial[index].coefficient /= term.coefficient;
  }

  return OP_SUCCESS;
}
```

```
char polynomialLimit(Polynomial polynomial, long double * limit) {
  if (!limit) {
    return OP_FAILED;
  }

  *limit = 0;
  for (unsigned int index = 0; polynomial[index].coefficient && index <
MAX_POLYNOMIAL; ++index) {
    if (polynomial[index].degree == 0) {
      *limit += polynomial[index].coefficient;
    }
    else if (polynomial[index].degree > 0) {
      *limit = INFINITY;
      return OP_SUCCESS;
    }
  }

  return OP_SUCCESS;
}

char verifyO(Polynomial temp, Polynomial computationalComplexityPolynomial, Term
bigO) {
  unsigned int index = 0;
  for (; computationalComplexityPolynomial[index].coefficient && index <
MAX_POLYNOMIAL; ++index) {
    temp[index].coefficient =
computationalComplexityPolynomial[index].coefficient;
    temp[index].degree = computationalComplexityPolynomial[index].degree;
  }
  temp[index].coefficient = 0;

  if (polynomialDivideByTerm(temp, bigO) != OP_SUCCESS) {
    return OP_FAILED;
  }

  long double limit;
  if(polynomialLimit(temp, &limit) != OP_SUCCESS){
    return OP_FAILED;
  }

  if (limit <= 0 || limit == INFINITY) {
    return OP_FAILED;
  }

  return OP_SUCCESS;
}

void printTerm(Term term) {
  if (term.coefficient != 1) {
    printf("%Lf*", term.coefficient);
  }
```

```
  if (term.degree == 1) {
    printf("n");
  }
  else if (term.degree < 0) {
    printf("1/(n");
    for (long long int index = term.degree + 1; index++;) {
      printf("*n");
    }
    printf(")");
  }
  else {
    printf("n");
    for (long long int index = term.degree - 1; index--;) {
      printf("*n");
    }
  }

}

void printComputationalComplexityPolynomialAndBigO(char * promt, char*
relationPromt, Polynomial polynomial, Term bigO) {
  if(promt){
    printf("%s: ", promt);
  }

  if (bigO.coefficient != 1 || bigO.degree < 0) {
    if (promt) {
      printf("\r\n");
    }
    printf("Warning: Bad BigO notation!\r\n");
  }

  printf("O(");
  printTerm(bigO);

  if (relationPromt) {
    printf(") %s ", relationPromt);
  }
  else{
    printf(") # ");
  }

  for (unsigned int index = 0; polynomial[index].coefficient && index <
MAX_POLYNOMIAL; ++index) {
    printTerm(polynomial[index]);
  }
  printf("\r\n");
}

int main() {
```

```
  Polynomial temp;

  {
    Polynomial computationalComplexityPolynomial =
COMPUTATIONAL_COMPLEXITY_POLYNOMIAL;
    Term bigO = TERM_O__A;

    if (verifyO(temp, computationalComplexityPolynomial, bigO) == OP_SUCCESS)
{
      printComputationalComplexityPolynomialAndBigO(NULL, "==",
computationalComplexityPolynomial, bigO);
    }
    else {
      printComputationalComplexityPolynomialAndBigO(NULL, "!=",
computationalComplexityPolynomial, bigO);
    }
  }

  {
    Polynomial computationalComplexityPolynomial =
COMPUTATIONAL_COMPLEXITY_POLYNOMIAL;
    Term bigO = TERM_O__B;

    if (verifyO(temp, computationalComplexityPolynomial, bigO) == OP_SUCCESS) {
      printComputationalComplexityPolynomialAndBigO(NULL, "==",
computationalComplexityPolynomial, bigO);
    }
    else {
      printComputationalComplexityPolynomialAndBigO(NULL, "!=",
computationalComplexityPolynomial, bigO);
    }
  }

#if defined(__linux__) || defined(__unix__) || defined(__APPLE__)
  printf("Press Enter to continue . . .");
  (void)getchar();
#elif defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)
  system("pause");
#else
#endif

  return 0;
}
```