Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Реферат

По дисциплине «Основы алгоритмизации и программирования»

На тему «Рекурсия»

Выполнила:

Студент(ка) 1 курса 7 группы

Подшиваленко Диана Игоревна

Проверил:

Белодед Николай Иванович

2023, Минск

**Содержание**

1. Предисловие … 3
2. Понятие рекурсии … 4
3. Применение рекурсии … 4
4. Отличительные особенности цикла и рекурсии… 5
5. Прямая и косвенная рекурсия … 5
6. Преимущества и недостатки … 6
7. Конкретные примеры реализации рекурсии для решения задач … 7

А. Реализация вычисления n-го числа Фибоначчи … 7

Б. Реализация вычисления факториала числа … 7

В. Реализация возведения числа в целую степень … 8

Г. Реализация перевода числа из десятичной системы счисления в двоичную … 8

Д. Реализация поиска в глубину на графах … 9

Е. Реализация алгоритма быстрой сортировки … 10

Ж. Пример реализации вычисления факториала с косвенной рекурсией, используя указатели на функцию … 11

1. Выводы … 11

**Предисловие**

В данном реферате будет рассмотрен такой метод решения некоторых задач как рекурсия, ее виды, применения, а также будут приведены примеры алгоритмов решения задач с использованием рекурсии.

**1.Понятие рекурсии**

**Рекурсия** — это поведение функции, при котором она вызывает саму себя. Такие функции называются рекурсивными. В отличие от цикла, они не просто повторяются несколько раз, а работают «внутри» друг друга.

Рекурсия считается одним из основных понятий в программировании. Это метод решения задач, похожий на математическую индукцию: чтобы функция выполнилась, нужно сначала получить ее результат при вызове с другим значением.

Чтобы рекурсивный вызов закончился, нужно прописать в функции условие выхода из рекурсии: например, если получено какое-то значение, нужно не запускать функцию заново, а вернуть некоторый результат. Если не прописать условие выхода, рекурсия будет бесконечной. А пока условие выхода не достигнуто, все вызванные экземпляры функции будут работать одновременно — один вкладывает другой в себя.

**2.Применение рекурсии**

Обычно рекурсию применяют при расчетах, которые подразумевают использование результата одного шага для подсчитывания другого. Зачастую задачи можно решить и без рекурсии, но ее использование иногда делает код проще, короче и быстрее, чем альтернативные варианты. Правда, рекурсия может слишком нагружать компьютер, поэтому такие решения применяют не всегда.

Рекурсию можно встретить в физике, биологии, лингвистике и даже в архитектуре. Самый наглядный пример рекурсии — два поставленных друг напротив друга зеркала.

Рекурсивное мышление может быть полезным при решении сложных математических или логических задач. Оно позволяет разбить задачу на более простые подзадачи и применять рекурсивные шаги для их решения. Также оно позволяет анализировать последствия и взаимосвязи между различными аспектами проблемы.

Рекурсия может использоваться для обработки некоторых структур данных. Например, для обхода дерева или графа, для обработки связанных списков. Также она применяется для генерации различных последовательностей, таких как числовые ряды, комбинаторные последовательности и другие. Рекурсия может быть применена для решения задач, основанных на принципе "разделяй и властвуй". Этот подход заключается в разбиении задачи на более маленькие подзадачи, решение каждой из которых осуществляется рекурсивно.

**3. Отличительные особенности цикла и рекурсии**

На интуитивном уровне рекурсивный вызов легко перепутать с циклом. И то, и другое понятие подразумевает, что функция выполняется несколько раз. Однако это принципиально разные понятия. Вот основные отличия:

1) Рекурсия основана на принципе вызова функции из самой себя: функция вызывает саму себя с новыми параметрами до достижения базового случая, или по-другому точкой остановки рекурсии. Циклы используют итерационную структуру, где блок кода выполняется многократно до выполнения заданного условия.

2) Рекурсия изменяет ход выполнения программы, потому что функция вызывает саму себя и должна ждать завершения рекурсивного вызова перед продолжением выполнения. Циклы предоставляют более прямой и линейный контроль выполнения, так как код в цикле выполняется последовательно без необходимости ожидания завершения других вызовов.

3) Рекурсия требует явного задания точки остановки или условия завершения, чтобы избежать бесконечной рекурсии. Если это не предусмотреть, рекурсия может вызвать переполнение стека вызовов. В циклах условие завершения проверяется на каждой итерации, и выполнение цикла прекращается, когда условие становится ложным.

4) Рекурсия может потреблять больше памяти и ресурсов, потому что каждый рекурсивный вызов сохраняет свое состояние. Циклы обычно требуют меньше памяти и ресурсов, так как они не создают дополнительных вызовов функций.

5) Рекурсия может быть удобной и красивой для решения определенного типа задач, особенно для работы с рекурсивными структурами данных, такими как деревья, графы. Однако она может быть более сложной для понимания и отладки. Циклы обычно более прямолинейны и просты в использовании.

**4.** **Прямая и косвенная рекурсия**

Рекурсию можно условно разделить на прямую и косвенную.

Прямая вызывает саму себя напрямую. То есть, например, функция A вызывает функцию A.

Косвенная действует через дополнительную функцию. Например, функция A вызывает функцию B, а та в снова вызывает функцию A. Это тоже рекурсия, просто не такая очевидная.

**5. Преимущества и недостатки**

Рекурсия является достаточно распространенным методом решения определенного типа задач. Этот вид решения, как и другие, имеет свои преимущества и недостатки. Рассмотрим основные из них.

**Преимущества:**

- прочитать рекурсивный код зачастую проще, чем программу, полную замысловатых алгоритмов для решения той же задачи без рекурсии. Это важный плюс, если речь идет о работе в команде, где код часто читают и другие люди.

- некоторые задачи, по типу чисел Фибоначчи или факториалов, — рекурсивные по самому своему определению. То же самое касается задач на ряды, фракталы и так далее. Решить такую задачу через рекурсию — один из наиболее наглядных и интуитивно понятных способов.

- рекурсивная функция обычно короче, чем реализация без рекурсии. Разработчику проще и удобнее написать несколько строк кода, чем создавать огромную программу, тратить время и силы и путаться в написанном.

- многие считают красивым и изящным этот алгоритм. Это и правда так, если, например, посмотреть на снежинки: в них есть определенная красота. Рекурсию используют даже в искусстве: от матрешек до сложных узоров, украшающих готические соборы.

**Недостатки:**

- несмотря на все плюсы рекурсии, нужно пользоваться ею очень осторожно. Она устроена так, что, пока не выполнится последний вызов функции, все предыдущие не завершатся, потому что они “ожидают” вычисления последующего шага. Поэтому такие программы отнимают много оперативной памяти. Так что разработчики смотрят на параметр производительности и решают, что выгоднее в конкретном случае — использовать рекурсию или нет.

- есть риск забыть написать базовый случай, и тогда рекурсия превратиться в бесконечную.

**6. Конкретные примеры реализации рекурсии для решения задач**

**6.1 Реализация вычисления n-го числа Фибоначчи**

#include <iostream>

#include<Windows.h>

#include <conio.h>

using namespace std;

int fib(int n) {

if (n == 0) {

return 0;

}

if (n == 1 || n == 2) {

return 1;

}

return fib(n - 1) + fib(n - 2);

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

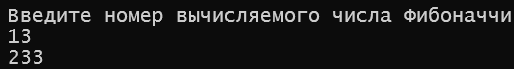
int n;

cout << "Введите номер вычисляемого числа Фибоначчи\n";

cin >> n;

cout << fib(n);

}

Результат выполнения:

**6.2** **Реализация вычисления факториала числа**

#include <iostream>

#include<Windows.h>

#include <conio.h>

using namespace std;

long long int factorial(long long int n) {

if (n == 0) {

return 1;

}

if (n == 1) {

return 1;

}

return factorial(n - 1) \* n;

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

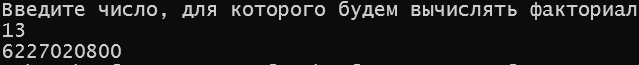
long long int n;

cout << "Введите число, для которого будем вычислять факториал\n";

cin >> n;

cout << factorial(n);

}

Результат выполнения:

**6.3 Реализация возведения числа в целую степень**

#include <iostream>

#include<Windows.h>

#include <conio.h>

using namespace std;

long long int pows(long long int n, long long int m) {

if (m == 0) {

return 1;

}

if (m == 1) {

return n;

}

return n \* pows(n, m - 1);

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

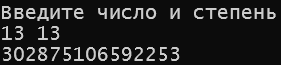
long long int n, m;

cout << "Введите число и степень\n";

cin >> n >> m;

cout << pows(n, m);

}

Результат выполнения:

**6.4 Реализация перевода числа из десятичной системы счисления в двоичную**

#include <iostream>

#include<Windows.h>

#include <conio.h>

#include <string>

using namespace std;

static string b = "";

void bin(int num) {

if (num % 2 == 0) {

b += "0";

}

else {

b += "1";

}

if (num / 2 != 0) bin(num / 2);

else reverse(b.begin(), b.end());

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int num;

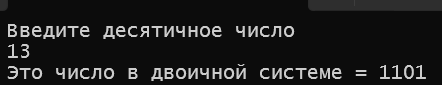
cout << "Введите десятичное число\n";

cin >> num;

bin(num);

cout << "Это число в двоичной системе = " << b;

}

Результат работы:

**6.5 Реализация поиска в глубину на графах**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <stdio.h>

#include <vector>

using namespace std;

void DFS(vector<vector<int>>& graph, int vershina, vector<int>& visited) {

cout << "Посещаем вершину " << vershina + 1 << endl;

visited[vershina] = 1;

for (int neighbour : graph[vershina]) {

if (!visited[neighbour]) {

DFS(graph, neighbour, visited);

}

}

cout << "У вершины с номером " << vershina + 1 << " посещены все соседи" << endl;

}

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n, m;

cout << "Введите количество вершин и ребер в графе " << endl;

cin >> n >> m;

vector<int> visited(n);

vector<vector<int>> graph(n);

int a, b;

cout << "Введите пары вершин, между которыми есть ребро" << endl;

for (int j = 0; j < m; j++) {

cin >> a >> b;

a--;

b--;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

for (int i = 0; i < n; i++) {

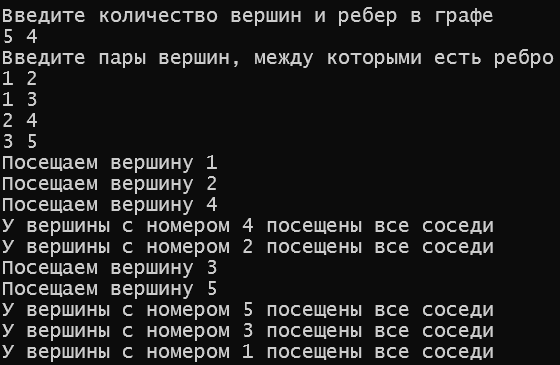
if (!visited[i]) {

DFS(graph, i, visited);

}

}

}

Результат работы:

**6.6 Реализация алгоритма быстрой сортировки**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <stdio.h>

#include <random>

using namespace std;

void quick\_sort(int\*& mas, int l, int r) {

int first = l;

int last = r;

int middle = mas[(first + last) / 2];

while (first < last) {

while (mas[first] < middle) first++;

while (mas[last] > middle) last--;

if (first <= last) {

swap(mas[first], mas[last]);

last--;

first++;

}

}

if (l < last) quick\_sort(mas, l, last);

if (first < r) quick\_sort(mas, first, r);

}

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

srand(time(NULL));

int n;

cout << "Введите количество элементов в массиве\n";

cin >> n;

int\* mas = new int[n];

cout << "Исходный массив\n";

for (int i = 0; i < n; i++) {

mas[i] = rand() % 1000 + 1;

cout << mas[i] << ' ';

}

cout << '\n';

quick\_sort(mas, 0, n - 1);

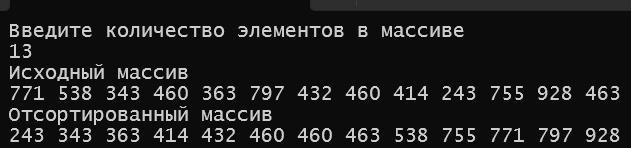
cout << "Отсортированный массив\n";

for (int i = 0; i < n; i++) {

cout << mas[i] << ' ';

}

}

Результат выполнения:

**6.6 Пример реализации вычисления факториала с косвенной рекурсией, используя указатели на функцию**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <stdio.h>

using namespace std;

long long int factorial1(long long int);

long long int factorial2(long long int);

long long int (\*f)(long long int) = factorial1;

long long int (\*g)(long long int) = factorial2;

long long int factorial1(long long int n) {

if (n == 0 || n == 1) {

return 1;

}

return n \* g(n - 1);

}

long long int factorial2(long long int n) {

if (n == 0 || n == 1) {

return 1;

}

return n \* f(n - 1);

}

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

long long int n;

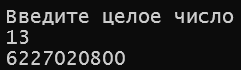
cout << "Введите целое число\n";

cin >> n;

cout << factorial1(n);

}

Результат выполнения:



**7. Выводы**

Рекурсия в программировании является мощным инструментом, который позволяет решать задачи, основанные на разбиении на подзадачи. Она часто применяется в случаях, когда задача может быть естественно разделена на более маленькие подзадачи, которые могут быть решены похожим образом.

Одним из важных правил использования рекурсии является определение базового случая, который описывает самый простой случай, когда рекурсия может быть остановлена и возвращен результат. Без базового случая рекурсия может вызвать переполнение стека и привести к ошибке.

Важно учитывать затраты памяти и производительности при использовании рекурсии, особенно для задач с большой глубиной рекурсии. В некоторых случаях рекурсивный код может быть переписан в итеративный, что может привести к более эффективной работе программы.

Наконец, при использовании рекурсии необходимо быть внимательным, чтобы избежать бесконечной рекурсии или зацикливания. Корректное определение базового случая и правильное обновление параметров на каждой итерации помогут избежать таких проблем.

Хотя рекурсия может быть очень полезной, она не всегда является наилучшим решением для всех задач. В некоторых случаях итеративные алгоритмы могут быть более простыми и эффективными. Поэтому перед использованием рекурсии важно внимательно проанализировать задачу и рассмотреть альтернативные подходы.