

###### Домашнее задание на тему

## «Анализ вычислительной сложности алгоритма»

## Вариант №17

По курсу: Системное программирование

Студенты: Подорин А.А. ИУ4-93

Зотьева Д.Е. ИУ4-93

Ковалев Д.Д. ИУ4-17М

Преподаватель: Чернов М.М.

Москва

2014

##### Москва

2009

Содержание

[Введение 3](#_Toc406196147)

[1 Условие задачи 5](#_Toc406196148)

[2 Анализ алгоритма решаемой задачи и его сложности 6](#_Toc406196149)

[2.1 Анализ общего случая 6](#_Toc406196150)

[2.2 Обработка выражения 6](#_Toc406196151)

[2.3 Пересечение множеств 7](#_Toc406196152)

[2.4 Объединение множеств 8](#_Toc406196153)

[2.5 Вычитание множеств 10](#_Toc406196154)

[3 Реализация алгоритма на языке C 12](#_Toc406196155)

[4 Экспериментальное исследование зависимости времени выполнения алгоритма от мощности множества 18](#_Toc406196156)

[Выводы 19](#_Toc406196157)

[Список использованных источников 20](#_Toc406196158)

Введение

Теория сложности вычислений зародилась ещё в конце XIX века, но бурное развитие получила в середине-конце XX века с ростом использования ЭВМ. *Оценка вычислительной сложности алгоритма* ведётся при помощи анализа объёма его работы при различных объёмах входных данных, при этом различают временную и пространственную сложности, оценку сложности в различных случаях (худший, лучший, вероятный) и другие нюансы, которые будут рассмотрены ниже.

Под *объёмом работы алгоритма* следует понимать затраченные машинные циклы, которые, однако, зависят не только от самого алгоритма, но и от языка программирования, компилятора, архитектуры процессора и др. *Объём входных данных* – это численный показатель, который характеризует массивы данных, обрабатываемых исследуемым алгоритмом. Допустим, при сортировке одним методом обработка тысячи чисел занимает 1 с, а обработка миллиона чисел – 10 с, при использовании другого алгоритма может потребоваться 2 с и 5 с соответственно. В таких условиях нельзя однозначно сказать, какой алгоритм лучше.

В ряде случаев алгоритмическое решение задачи может быть осуществлено преимущественно за счёт *ресурсов времени* (тактов процессора) или *ресурсов памяти* (запоминающие устройства). Два эти способа определяют два различных критерия оценки сложности вычисления: временной и пространственный.

Сложность алгоритма так же оценивают в лучшем, худшем и среднем (наиболее вероятном) случае. *Лучший* случай подразумевает под собой, что данные, поступившие на вход, находятся в наиболее удобном для алгоритма представлении (например, уже отсортированы), тогда сложность вычисления может значительно снижаться. *Худший* случай рассматривает противоположный, т.е. самый неблагоприятный вариант поступивших входных данных или начала работы алгоритма. Наиболее вероятный вариант начала работы алгоритма рассматривает *средний* случай. Стоит сказать, что по временной сложности алгоритмы обычно оценивают в среднем и худшем случае, а по пространственной сложности – только в худшем случае.

Каким же образом измеряется вычислительная сложность алгоритма? Алгоритм имеет *сложность O(f(n))*, если при увеличении размерности входных данных *N*, время выполнения алгоритма возрастает с той же скоростью, что и функция *f(N).* Такой способ оценки сложности алгоритма называется О-нотацией, она учитывает только наиболее быстрорастущие функции и позволяет однозначно оценить сложность работы алгоритма. В случае, когда алгоритм-это цикл, расположенный в другом цикле, их сложности перемножаются, а если в теле программы два эти алгоритма следуют друг за другом, их сложности складываются.

Чаще всего сложность алгоритма сводится к одной из перечисленных ниже функций:

1. *C* – константа

2. *log(log(N))*

3. *log(N)*

4. *NС, 0<C<1*

5. *N*

6. *N\*log(N)*

7. *NC, C>1*

8. *CN, C>1*

9. *N!*

Соответственно, список приведён по возрастанию сложности и скорости обработки информации алгоритмом. Приемлемыми в большинстве случаев считаются алгоритмы *до N\*log(N),* хотя однозначный ответ на вопрос об эффективности рассматриваемого алгоритма можно дать, зная частоту его использования, объем и характер обрабатываемого массива данных, вычислительные возможности ЭВМ и др.

Стоит отметить, что не смотря на то, что одни алгоритмы однозначно имеют большую сложность, чем другие, их всё-таки можно разумно использовать. Так, в случае небольшого объема данных, обрабатываемого алгоритмом большого уровня сложности, использование этого алгоритма вполне приемлемо. При взаимодействии алгоритмов, следует учитывать, какие подпрограммы включают в себя алгоритмы большой сложности, так, наиболее часто вызываемые программы со сложными алгоритмами будут отрицательно сказываться на скорости работы программы.

1 Условие задачи

Имеется файл, содержащий строки вида:

*<ИмяМножества><ИмяЭлемента>.*

Первым параметром в программу передается путь до данного файла. Вторым параметров передается строка, состоящая из имен множеств и символов –, +, \*, () которые означают операции вычитания, объединения, пересечения множеств. Операции применяются слева направо в равном приоритете.

Пример строки (из множества всех Существ вычесть Студентов и пересечь то, что получилось с множеством Раздолбаев):

*((Существо–Студент)\*Раздолбаи)*

2 Анализ алгоритма решаемой задачи и его сложности

2.1 Анализ общего случая

В данной задаче присутствует несколько параметров влияющих на сложность алгоритма. Их можно разделить на следующие группы:

– количество выполняемых операций (количество множеств);

– количество элементов в множествах.

В общем случае мы имеем суммарно *M* множеств и операций над этими множествами. На *M* наложено ограничение, связанное с присвоением имени множеству: множества именуются буквами латинского алфавита, следовательно, максимальное значение *M* не превышает 26.

Также в общем случае в каждом из множеств может быть произвольное количество элементов *Ni*, где *i = 1, 2 … m*, где *m* – количество множеств.

Таким образом, сложность алгоритма является функцией нескольких переменных и провести ее теоретический анализ для общего случая не представляется возможным. Ее значение будет колебаться в зависимости не только от количества элементов в множествах и количества операций, но также и в зависимости от типов операций, т.к. предлагаемые к рассмотрению операции над множествами генерируют множества непредсказуемого размера. Можно лишь качественно оценить, что для операций разности и пересечения мощность множества будет уменьшаться, а для объединения – возрастать.

В связи с этим рассмотрим фрагменты общего алгоритма, а также частные случаи для отдельно взятых операций.

2.2 Обработка выражения

В первую очередь происходит считывание файла, содержащего информацию о множествах и их элементах. Проверяется уникальность элементов в множествах, под каждое множество выделяется объем памяти, строго определенный мощностью этого множества. Имена исходных множеств уникальны и резервируются.

Рассмотрим считывание выражения на следующем примере:

*A+(B\*(C-D\*A))*

Результат обработки этого выражения представлен в таблице 2.1.

Таблица 2.1 – Результат считывания выражения

|  |
| --- |
| A+(B\*(C-D\*A)) A+(B\*(C-D\*A))-2 A+(B\*(C-D\*A))-4 A+(B\*(C-D\*A))-5 A+(B\*(C-D\*A))-7 A+(B\*(C-D\*A))-8 A+(B\*(E\*A))-2 A+(B\*(E\*A))-4 A+(B\*(E\*A))-5 A+(B\*(E\*A))-7 A+(B\*(E\*A))-8 A+(B\*(F))-2 A+(B\*(F))-4 A+(B\*(F))-5 A+(B\*(F))-7 A+(B\*F)-2 A+(B\*F)-4 A+(B\*F)-5 A+(G)-2 A+(G)-4 A+G-2 H |

Как видно из листинга, программа передвигается по выражению, осуществляя поиск самого вложенного выражения. Как только найдена операция с двумя операндами, она выполняется, после чего эти два множества сворачиваются в одно, которому присваивается имя согласно алфавиту, но не зарезервированное ранее (при чтении данных исходного файла). При отсутствии вложенности (реализованной с помощью скобок) осуществляется вычисление слева направо и последовательное сворачивание множеств. Как только длина строки составляет один символ, программа считается завершенной, а итоговое множество – результат вычисления выражения.

Поскольку количество возможных множеств и операций ограничено числом 26, то рассматривать вычислительную сложность в зависимости только от этого количества не является приоритетной задачей.

2.3 Пересечение множеств

Блок-схема алгоритма осуществляющего пересечение множеств представлена на рисунке 2.1.

Сущность алгоритма заключается в следующем: организуется вложенный цикл, в котором во внешнем цикле осуществляется перебор элементов первого множества, а во внутреннем перебор элементов второго. При нахождении совпадения элемент записывается в результирующее множество, иначе происходит переход к следующей итерации.

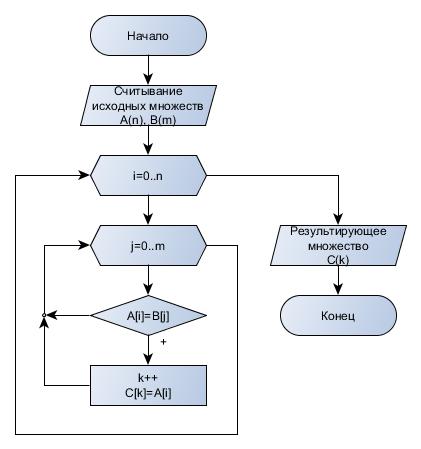


Рисунок 2.1 – Блок-схема алгоритма вычисления пересечения множеств

Лучшим случаем будет вариант, когда один из операндов является множеством из одного элемента (вариант пустого множества не рассматривается, т.к. в этом случае множество не существует). Таким образом вычислительная сложность составит *О(N)*, где *N* – мощность другого множества.

Худшим случаем является вариант полного несовпадения множеств, т.е. множества не имеют одинаковых элементов и их пересечение равно 0. В этом случае будут выполнены все возможные итерации алгоритма и вычислительная сложность составит *О(N\*M)*, или в случае одинаковой мощности *О(N2).*

2.4 Объединение множеств

Блок-схема алгоритма осуществляющего объединение множеств представлена на рисунке 2.2.

Сущность алгоритма заключается в следующем: определяется множество с большей мощностью. Элементы множества с большим множеством переписываются в результирующее множество. Сложность этой операции составляет *O(N)* или *O(M).*

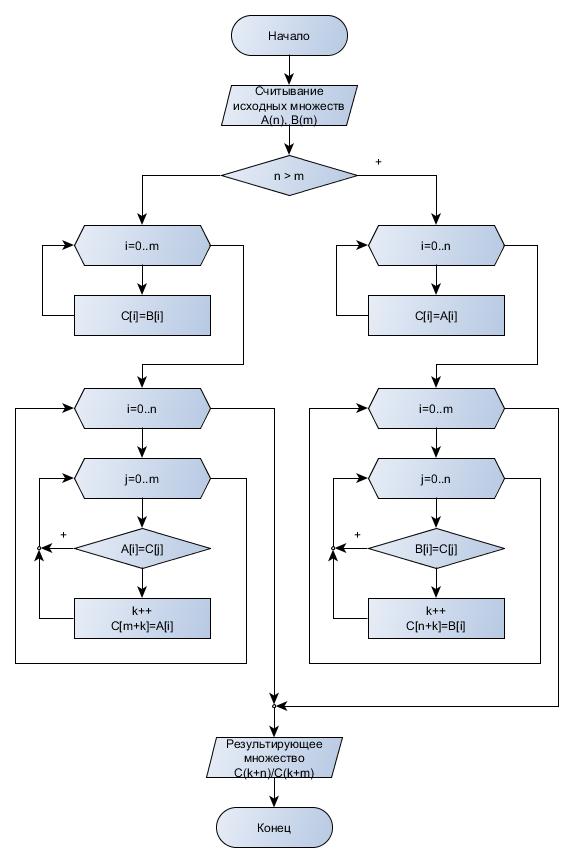


Рисунок 2.2 – Блок-схема алгоритма вычисления объединения множеств

После этого организуется вложенный цикл: во внешнем цикл осуществляется перебор элементов множества с меньшей мощностью, а во внутреннем перебор элементов уже сформированной части результирующего множества (по сути – исходного множества с большей мощностью). При обнаружении совпадающих элементов происходит переход к следующей итерации. Если элементу не было найдено соответствие, то он дописывается в конец результирующего множества.

Реализация вложенного цикла приводит к вычислительной сложности *O(N\*M)* или в случае множеств равной мощности *O(N2)*. Таким образом, результирующая вычислительная сложность алгоритма *O(N+N2)*.

Лучшим случаем является случай нулевого множества в качестве одного из операндов. Тогда вычислительная сложность снижается до *O(N).*

Худший случай достигается при полном несовпадении множеств: необходимо осуществить все итерации, т.е. при равномощных множествах сложность алгоритма *O(N+N2)*.

2.5 Вычитание множеств

Блок-схема алгоритма осуществляющего вычитание множеств представлена на рисунке 2.3.

Сущность алгоритма заключается в следующем: на первом шаге элементы первого множества переписываются в результирующее множество. Сложность этой операции составляет *O(N)*. На следующем шаге организуется вложенный цикл: во внешнем цикле осуществляется перебор элементов второго множества, во внутреннем – элементов результирующего множества. При нахождении совпадающих элементов, начиная с позиции вхождения осуществляется перезапись оставшейся части множества на один элемент в сторону уменьшения индекса (таким образом реализуется исключение элемента из множества). Перезапись представляет собой еще один цикл, количество итераций которого равно разности текущей мощности результирующего множества и позиции вхождения совпадающего элемента. Таким образом количество итераций вложенного цикла по j зависит от количества совпавших элементов.

Оценим вычислительную сложность алгоритма в различных случаях.

Лучшим случаем является вариант, когда второй операнд – нулевое множество. При этих условиях вложенный цикл второго шага алгоритма не реализуется и итоговая сложность алгоритма *O(N)*.

Определить какой случай является худшим достаточно трудоемкая математическая задача, поэтому приведем два крайних случая:

– множества абсолютно различны. В этом случае просто выполнятся все итерации циклов по *i* и *j*, таким образом общая вычислительная сложность составит *O(N+N\*M)* или для равномощных множеств *O(N+N2)*;

– множества абсолютно идентичны. В этом случае внутренний цикл по j и цикл по k в сумме с каждой итерацией по i будут выполнять на одну итерацию меньше. Определим вычислительную сложность для случая равномощных множеств.

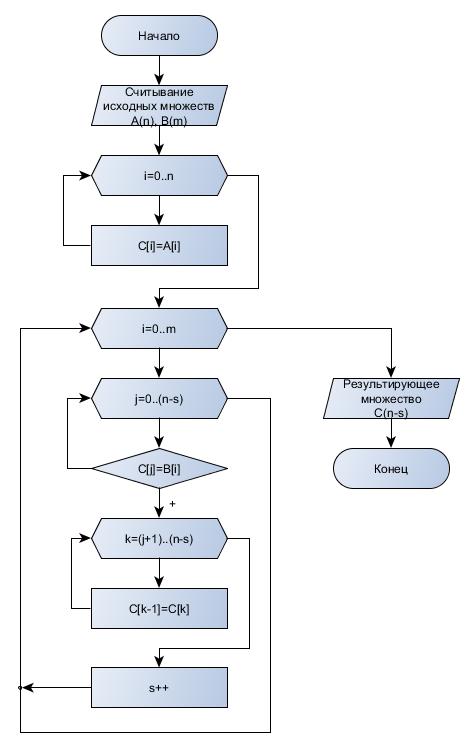


Рисунок 2.3 – Блок-схема алгоритма вычисления вычитания множеств

Общее число итераций можно записать формулой:

Таким образом, вычислительная сложность для данного случая с учетом копирования исходного первого множества в результирующее: *O(N2)*. На основании этого худшим случаем будем считать случай полностью различных множеств. Для равномощных множеств частично различных вычислительная сложность будет находиться между *O(N2)* и *O(N2+N).*

3 Реализация алгоритма на языке C

Листинг кода представлен в таблице 3.1.

Таблица 3.1 – Листинг кода реализации алгоритма решения поставленной задачи на языке C

|  |
| --- |
| /\*  \* File: main.c  \* Author: Александр  \*  \* Created on 10 Декабрь 2014 г., 19:44  \*/  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <stdbool.h>  #include <malloc.h>  /\*  \*  \*/  char\* mnoj;  int\*\* elems;  int\* num\_elems;  int getInd(char val);  int getElem(char\* val);  void init();  char generateNewMnoj();  void movePointer(char\* movep,int start,int movenumber);  bool isAlph(char ch);  bool isOper(char ch);  void peresech(char a, char b, char save);  void summa(char a, char b, char save);  void raznost(char a, char b, char save);  int main(int argc, char\*\* argv) {  init();  char\* data\_file=argv[1];  char\* sum\_mem=argv[2];  FILE\* f=fopen(data\_file,"r");  if(!f)  {  printf("Wrong file\n");  return 1;  }  char tmp[51];  char mnoj\_name;  int elem;  int num\_mnoj=0;    while(fgets(tmp,51,f)!=NULL)  {  mnoj\_name=tmp[0];  elem=getElem(tmp);  int ind=getInd(mnoj\_name);  if(ind!=-1)  {  bool is\_new=true;  int l;  for(l=0;l<num\_elems[ind];l++)  {  if(elems[ind][l]==elem)  {  is\_new=false;  }  }  if(is\_new)  {  num\_elems[ind]++;  elems[ind]=(int \*) realloc(elems[ind],num\_elems[ind]\*sizeof(int));  elems[ind][num\_elems[ind]-1]=elem;  }  }  else  {  num\_mnoj++;  mnoj=(char\*) realloc(mnoj,num\_mnoj\*sizeof(char));  mnoj[num\_mnoj-1]=mnoj\_name;    elems=(int\*\*) realloc(elems,num\_mnoj\*sizeof(int\*));  elems[num\_mnoj-1]=(int\*) calloc(1,sizeof(int));  elems[num\_mnoj-1][0]=elem;  num\_elems=(int\*) realloc(num\_elems,num\_mnoj\*sizeof(int));  num\_elems[num\_mnoj-1]=1;    }    }    puts("Data file is read");    int j=2;  puts(sum\_mem);    while(strlen(sum\_mem)!=1)  {  printf("%s-%d\n",sum\_mem,j);  if(sum\_mem[j-2]=='(' && sum\_mem[j]==')' && isAlph(sum\_mem[j-1]))  {  sum\_mem[j-2]=sum\_mem[j-1];  movePointer(sum\_mem,j+1,2);  j=2;  }  else if(isAlph(sum\_mem[j-2]) && isAlph(sum\_mem[j]) && isOper(sum\_mem[j-1]))  {  char newMn=generateNewMnoj();  num\_mnoj++;    mnoj=(char\*) realloc(mnoj,num\_mnoj\*sizeof(char));  mnoj[num\_mnoj-1]=newMn;    elems=(int\*\*) realloc(elems,num\_mnoj\*sizeof(int\*));  elems[num\_mnoj-1]=(int\*) calloc(0,sizeof(int));  num\_elems=(int\*) realloc(num\_elems,num\_mnoj\*sizeof(int));  num\_elems[num\_mnoj-1]=0;  if(sum\_mem[j-1]=='\*')  {  peresech(sum\_mem[j-2],sum\_mem[j],newMn);  sum\_mem[j-2]=newMn;  movePointer(sum\_mem,j+1,2);  }  else if(sum\_mem[j-1]=='+')  {  summa(sum\_mem[j-2],sum\_mem[j],newMn);  sum\_mem[j-2]=newMn;  movePointer(sum\_mem,j+1,2);  }  else if(sum\_mem[j-1]=='-')  {  raznost(sum\_mem[j-2],sum\_mem[j],newMn);  sum\_mem[j-2]=newMn;  movePointer(sum\_mem,j+1,2);  }    j=2;  }  else if(sum\_mem[j-2]=='(' && sum\_mem[j]!=')')  {  j++;  }  else if(sum\_mem[j]=='(')  {  j=j+2;  }  }    puts(sum\_mem);  int last\_ind=getInd(sum\_mem[0]);  int l;  for(l=0;l<num\_elems[last\_ind];l++)  {  printf("%d\n",elems[last\_ind][l]);  }    return (EXIT\_SUCCESS);  }  int getInd(char val)  {  int ret=-1;  int i;  for(i=0;i<sizeof(mnoj)/sizeof(char);i++)  {  if(mnoj[i]==val)  {  ret=i;  }  }    return ret;  }  int getElem(char\* val)  {  char new[49];  int ret;  int i;  for(i=2;i<50;i++)  {  new[i-2]=val[i];  }  ret=atoi(new);  return ret;  }  void init()  {  mnoj=malloc(0);  elems=malloc(0);  num\_elems=malloc(0);  }  char generateNewMnoj()  {  char ret='A';  while(isAlph(ret) && getInd(ret)!=-1)  {  ret++;  }  if(!isAlph(ret))  {  puts("No more letters! Sorry guys!");  exit (2);  }  return ret;  }  void movePointer(char\* movep,int start,int movenumber)  {  int k=start;  for(k=start;k<=strlen(movep);k++)  {  movep[k-movenumber]=movep[k];  }    }  bool isAlph(char ch)  {  if(((int) ch>='A' && (int) ch<='Z') || ((int) ch>='a' && (int) ch<='z'))  {  return true;  }  return false;  }  bool isOper(char ch)  {  if(ch=='\*' || ch=='+' || ch=='-')  return true;  return false;  }  void peresech(char a, char b, char save)  {  int ind\_a=getInd(a);  int ind\_b=getInd(b);  int ind\_c=getInd(save);  int i;  int j;  for(i=0;i<num\_elems[ind\_a];i++)  {    for(j=0;j<num\_elems[ind\_b];j++)  {  if(elems[ind\_a][i]==elems[ind\_b][j])  {  num\_elems[ind\_c]++;    elems[ind\_c]=(int\*)realloc(elems[ind\_c],num\_elems[ind\_c]\*sizeof(int));  elems[ind\_c][num\_elems[ind\_c]-1]=elems[ind\_a][i];  }  }  }  }  void summa(char a, char b, char save)  {  int ind\_a=getInd(a);  int ind\_b=getInd(b);  int ind\_c=getInd(save);  int i;  int j;  if(num\_elems[ind\_a]>num\_elems[ind\_b])  {  num\_elems[ind\_c]=num\_elems[ind\_a];  elems[ind\_c]=(int \*) calloc(num\_elems[ind\_c],sizeof(int));  for(i=0;i<num\_elems[ind\_a];i++)  {  elems[ind\_c][i]=elems[ind\_a][i];    }    for(i=0;i<num\_elems[ind\_b];i++)  {  bool flag=true;  for(j=0;j<num\_elems[ind\_c];j++)  {  if(elems[ind\_b][i]==elems[ind\_c][j])  {  flag=false;  break;  }  }  if(flag)  {  num\_elems[ind\_c]++;  elems[ind\_c]=(int \*) realloc(elems[ind\_c],num\_elems[ind\_c]\*sizeof(int));  elems[ind\_c][num\_elems[ind\_c]-1]=elems[ind\_b][i];  }  }  }  else  {  num\_elems[ind\_c]=num\_elems[ind\_b];  elems[ind\_c]=(int \*) calloc(num\_elems[ind\_c],sizeof(int));  for(i=0;i<num\_elems[ind\_b];i++)  {  elems[ind\_c][i]=elems[ind\_b][i];  }    for(i=0;i<num\_elems[ind\_a];i++)  {  bool flag=true;  for(j=0;j<num\_elems[ind\_c];j++)  {  if(elems[ind\_a][i]==elems[ind\_c][j])  {  flag=false;  break;  }    }  if(flag)  {  num\_elems[ind\_c]++;  elems[ind\_c]=(int \*) realloc(elems[ind\_c],num\_elems[ind\_c]\*sizeof(int));  elems[ind\_c][num\_elems[ind\_c]-1]=elems[ind\_a][i];    }  }  }  }  void raznost(char a, char b, char save)  {  int ind\_a=getInd(a);  int ind\_b=getInd(b);  int ind\_c=getInd(save);  int i;  int j;  elems[ind\_c]=(int\*) calloc(num\_elems[ind\_a],sizeof(int));  num\_elems[ind\_c]=num\_elems[ind\_a];  for(i=0;i<num\_elems[ind\_c];i++)  {  elems[ind\_c][i]=elems[ind\_a][i];  }  for(i=0;i<num\_elems[ind\_b];i++)  {  for(j=0;j<num\_elems[ind\_c];j++)  {  if(elems[ind\_c][j]==elems[ind\_b][i])  {  int k;  for(k=j+1;k<num\_elems[ind\_c];k++)  {  elems[ind\_c][k-1]=elems[ind\_c][k];  }  num\_elems[ind\_c]--;  elems[ind\_c]=(int\*) realloc(elems[ind\_c],num\_elems[ind\_c]\*sizeof(int));  break;  }  }  }  } |

В таблице 3.2 приведен листинг кода генерации тестовых значений.

Таблица 3.2 – Листинг кода генерации текстовых значений

|  |
| --- |
| <?php  ini\_set('memory\_limit', '1024M');  set\_time\_limit(0);  $num=$\_GET['num'];  $name=$\_GET['name'];  $file=fopen('gen.txt','w+');  $ret=array();  for($i=0;$i<$num;$i++)  {  $check=true;  while($check)  {  $rand=mt\_rand(0,$num\*100);  $flag=true;  for($j=0;$j<$i;$j++)  {  if($ret[$j]==$rand)  {  $flag=false;  break;  }  }  if($flag)  {  $ret[$i]=$rand;  $check=false;  }    }  fputs($file,$name." ".$ret[$i]."\n");  }  fclose($file); |

4 Экспериментальное исследование зависимости времени выполнения алгоритма от мощности множества

Экспериментальное исследование проводилось для случая одной операции над равномощными множествами сгенерированными случайным образом. Результаты исследования приведены в таблице 4.1.

Таблица 4.1 – Результаты экспериментальных исследований зависимости времени выполнения алгоритма от мощности множества

|  |  |  |  |
| --- | --- | --- | --- |
| Мощность множества | Время выполнения операции, с | | |
| Пересечение | Объединение | Разность |
| 10 | 0,001 | 0,001 | 0,001 |
| 100 | 0,001 | 0,001 | 0,001 |
| 1000 | 0,007 | 0,023 | 0,015 |
| 5000 | 0,128 | 0,241 | 0,159 |
| 10000 | 0,491 | 0,743 | 0,574 |
| 20000 | 1,989 | 2,767 | 2,091 |
| 40000 | 7,84 | 10,712 | 8,095 |
| 50000 | 12,265 | 16,209 | 12,605 |

На основе полученных результатов построим графики зависимостей (рисунок 4.1).

Рисунок 4.1 – Графики зависимостей времени выполнения алгоритма от мощности множества

Из графика видно, что каждая из зависимостей имеет квадратичный характер, что подтверждает сделанные в п.2 предположения.

Выводы

В ходе выполнения домашнего задания создана программа, реализующая выполнение поставленной задачи, а именно выполнение операций над множествами.

Была проанализирована вычислительная сложность алгоритмов пересечения, объединения и вычитания равномощных множеств для лучших и худших случаев:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Пересечение | Объединение | Вычитание |
| Лучший случай | *О(N)* | *О(N)* | *О(N)* |
| Худший случай | *О(N2)* | *О(N2+N)* | *О(N2+N)* |

Результаты теоретического анализа были подтверждены экспериментально. В частности, видно, что все зависимости имеют квадратичных характер, при этом зависимость для пересечения и вычитания более слабая, чем для объединения, а зависимости пересечения и вычитания схожи за счет уменьшения количества выполняемых итераций в алгоритме вычитания при нахождении идентичных элементов.

Стоит отметить, что при количестве операций *L>1*, сложность алгоритмов увеличится в *L* раз. Но необходимо понимать, что существенное влияние это окажет только при значениях мощности множеств сопоставимых с числом операций. При количестве операций *N>>L*, влиянием количества операций можно пренебречь.

Все рассмотренные случаи являются частными, поскольку в общем виде определить вычислительную сложность всего алгоритма не представляется возможным, что было обосновано в п.2.1.

Список использованных источников

1. Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. Алгоритмы, построение и анализ. Издание 3-е – М.: Вильямс, 2013. – 1324 с.

2. Брайан У. Керниган, Деннис М. Ритчи. Язык программирования С. Издание 3-е – М.: Вильямс, 2012. – 304 с.