ПРОЕКТ

по

Синтез и анализ на Алгоритми

на тема

„Алгоритми за генериране на прости числа. Решето на Ератостен.“

Изготвил: .................................

/Николай Чобанов/

Студент на ТУ-София, специалност: КСИ, факултетен № 121214189, 47 група

Проверил: .................................

/гл. ас. Константин Костадинов/

Дата:......................

Съдържание:

Въведение ......................................................................................... 3

Решетото на Ератостен ..................................................................... 6

Имплементация (Решетото на Ератостен) ...................................... 7

Програмен код (Решетото на Ератостен) ........................................ 7

Предимства и недостатъци (Решетото на Ератостен) .................... 8

Друг алгоритъм за генериране на прости числа ............................ 8

Програмен код (ІІ-ри алгоритъм) ..................................................... 8

Предимства и недостатъци (ІІ-ри алгоритъм) ................................. 9

Програмен код (Цяла програма) .................................................... 10

Използвана литература .................................................................. 12

1. **Въведение.**

* **Определение за алгоритъм** - Алгоритъм (от името на учения ал–Хорезми) е термин от математиката, информатиката, лингвистиката и други области, с който се означава крайна поредица от инструкции или изрично описание на постъпкова процедура за решаване на даден проблем, често свързан с изчисление или обработка на данни. По-строго казано, алгоритъмът е ефективен метод, който при даден списък от коректно дефинирани (описани) команди за изпълнение на задача и зададено едно начално състояние преминава през точно дефинирана поредица от последователни състояния и завършва в едно крайно състояние. Преходът между състоянията не е задължително да е детерминиран (еднозначно определен): някои алгоритми, известни като вероятностни алгоритми, съдържат елемент на случайност.
* **Типове алгоритми** 
  + **Според имплементацията**
    1. **Рекурсивни или итеративни** - Рекурсивният алгоритъм е алгоритъм, който прави поредица от обръщения към себе си дотогава, докато не се изпълни определено условие. Този метод на имплементация е характерен за функционалното програмиране. За да решат същите задачи, итеративните алгоритми използват повтарящи се конструкции (цикли), а понякога и допълнителни структури от данни като стекове. Някои проблеми са формулирани така, че е естествен изборът на едната или другата имплементация. Всяка рекурсивна версия на алгоритъм има еквивалентна (повече или по-малко сложна) итеративна версия, и обратно.
    2. **Логически** - Алгоритъмът може да се разглежда като контролирана логическа дедукция. Понятието алгоритъм е дефинирано през 1979 от Роберт Ковалски като

Алгоритъм = Логика + Управление

Логическият компонент изразява изходните аксиоми, а контролният компонент изразява правилата за извод, които се прилагат над аксиомите. Това лежи в основата на логическото програмиране. В чистите езици за логическо програмиране, управляващият компонент е фиксиран и алгоритмите се различават само по своите логически компоненти.

* + 1. **Серийни, паралелни или разпределени** - При дискутирането на алгоритмите обикновено се прави допускането, че компютрите изпълняват една команда на един такт. Проектираните по този начин алгоритми се наричат серийни, за сравнение с паралелните и разпределените алгоритми. Паралелни алгоритми се реализират тогава, когато компютърната архитектура се състои от няколко процесора, които могат едновременно да работят над изпълнението на алгоритъма, докато разпределените алгоритми използват ресурсите на множество машини, свързани в мрежа. Паралелните и разпределените алгоритми разделят изпълнението на задача на повече на брой подзадачи и след това събират резултатите от тях. При такива имплементации намаленото натоварване на отделните процесори се заплаща с комуникацията и синхронизацията между тях. Някои задачи могат да се реализират само със серийни алгоритми.
    2. **Детерминирани и недетерминирани** - Детерминираните алгоритми решават даден проблем, като винаги преминават през точно определена последователност от стъпки и при даден вход винаги извеждат един и същ резултат. Недетерминираните алгоритми използват различни техники, базирани на евристики и случайност.
    3. **Точни и приблизителни** - При все че много алгоритми достигат до точно решение на даден проблем, по различни причини в практиката се използват и приблизителни алгоритми, които (по горната класификация) могат да бъдат както недетерминирани, така и детерминирани.
  + **Според дизайна**
    1. **„Разделяй и владей“** - Алгоритмите, основани на парадигмата „разделяй и владей“, последователно (обикновено чрез рекурсия) раздробяват даден проблем на два и повече подпроблеми дотогава, докато те станат достатъчно малки, за да бъдат лесно решени. Пример за алгоритъм от този вид е алгоритъмът за сортиране чрез сливане. Сортирането се извършва, като целия масив от числа се раздели на сегменти и се сортират първо отделните сегменти, и след това (във фазата на „завладяването“) се слеят отделните сегменти.
    2. **Динамично програмиране** - Когато за дадена задача е известно, че оптималното решение може да бъде конструирано на база оптималните решения на поредица подзадачи, и то такива припокриващи се подзадачи, които също се свеждат до решаването на подзадачи, то се прилага подход, наречен динамично програмиране, при който се избягва необходимостта да се извършват повторно вече направени изчисления. Например, най-краткият път между начален и целеви връх в граф с тегла по дъгите може да се намери като се използва най-краткия път до целевия граф от всичките върхове, с които той е свързан.

Основната разлика между динамичното програмиране и „разделяй и владей“ е, че подзадачите са малко или много независими при „разделяй и владей“, докато подзадачите при динамичното програмиране се припокриват. Разлика между динамичното програмиране и обикновената рекурсия е в кеширането или мемоизацията на рекурсивните заявки. Когато подзадачите са независими, мемоизацията не е от полза, и следователно динамичното програмиране не е ефективно. С използването на мемоизация или с поддържането на таблица на вече решените подзадачи, динамичното програмиране свежда изчислението на много задачи, решавани с експоненциална сложност до задачи, решавани с полиномиална сложност.

* + 1. **Линейно програмиране** - Когато се решава дадена задача със средствата на линейното програмиране, проблемът се свежда до откриването на специфични неравенства, които определят дефиниционното множество на входните данни, и след това се прави опит да се намери максимума (или минимума) на някоя линейна функция над тях. Много задачи (като изчисляване на максимален трансфер по дъга на ориентиран граф) могат да се зададат в термините на линейното програмиране и да се решат със общ (стандартен) алгоритъм, какъвто е симплекс методът. Една по-сложна разновидност на линейното програмиране се нарича целочислено програмиране, при което пространството на решенията е ограничено от множеството на целите числа.
  + **Според приложението** - Всяка научна област си има своите специфични задачи за решаване и се нуждае от ефективни алгоритми. Свързани проблеми от една или повече области често се изследват заедно и се решават със сходни средства. Примери за цели класове алгоритми, с приложение в разнообразни области, са: алгоритми за търсене, алгоритми за сортиране, алгоритми за сливане, алгоритми за работа с числа, низове, графи, алгоритми за изчислителна геометрия, комбинаторни алгоритми, алгоритми за машинно обучение, криптографски алгоритми, алгоритми за компресиране на данни, алгоритми за синтактичен анализ (parsing) и други.
  + **Според сложността -** Алгоритмите могат да бъдат класифицирани по отношение на времето, което им е необходимо, за да решат проблем спрямо размера (дължината) на подадения вход. Някои алгоритми завършват за линейно време, други за полиномиално, трети за експоненциално време, а някои алгоритми никога не завършват работа. За някои задачи съществуват множество алгоритми с различна степен на сложност, докато за други задачи няма алгоритми за решаване или не са известни ефективни такива. Съществуват и изображения, задаващи съответствия между различни типове задачи. Благодарение на това е станало ясно, че е по-удачно класификацията да се прилага над типовете задачи, вместо над алгоритмите, като те се групират в класове на еквивалентност по отношение сложността на най-ефективните сред алгоритмите за решаването им.

1. **Решетото на Ератостен.**

* **Биографични данни** - Ератостен е древногръцки математик, географ и астроном. Считат го за създател на географията.

Живял е от 276 до 194 пр.н.е. Учил е в Атина, но около 245 г. пр.н.е. е повикан в Александрия, за да изпълнява функциите на възпитаник на престолонаследника Птоломей IV. Той е стоял начело на най-големия научен център на древността - Александрийската библиотека.

Ератостен е първият, който опитва да определи големината на Земята. Той знаел, че в деня на лятното слънцестоене Слънцето на пладне се оглежда в дъната на кладенците (т.е. то се намира точно отгоре в зенита си) в египетския град Сиене - съвременният Асуан. Той поставя стълб в двора си в Александрия и открива, че там слънчевите лъчи не падат вертикално, а под ъгъл (7°12´). Ератостен знаел, че двата града били разположени приблизително на един и същ меридиан. Знаейки също разстоянието между градовете (5000 египетски стадия) и разликата в ъглите на падане на слънчевите лъчи той успява да изчисли големината на земната окръжност по александрийския меридиан. Изчисленията му се оказват почти верни, разликата им с реалната дължина на земната окръжност е само 4 %. Това се дължи на неточните данни за разстоянието между двата града. Написал е първата книга озаглавена "География". Известен е и с алгоритъма си за намиране на простите числа, наречен "решето на Ератостен".

* **Решето на Ератостен** е алгоритъм за намиране на всички прости числа в интервала [1, n], където n е произволно естествено число. Идеята на алгоритъма е, че ако намерим просто число n, то всяко n-то след него няма да е просто.

Нека разгледаме списък на числата от 1 до n и започвайки от 2, изпълним следните стъпки:

Ако числото x е задраскано, преминаваме към следващото

Ако числото x не е задраскано, оставяме го така и задраскваме всяко x-то след него. Накрая всички незадраскани числа, които останат, са прости.

Решетото на Ератостен:

Записваме числата от 2 до n в редица:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ... , n

Намираме първото незачертано и немаркирано число – това е 2. Маркираме го, след което задраскваме всяко второ число след него:

(2), 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ... , n

По-нататък, отново намираме първото незачертано и немаркирано число: това е числото 3. Маркираме го и задраскваме всички числа в редицата, които са кратни на 3:

(2), (3), ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ... , n

След това, на ред е числото 5 – маркираме го и задраскваме всяко 5-то число:

(2), (3), ~~4~~, (5), ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ... , n

Така всички съставни числа се „отсяват“ и винаги сме сигурни, че минималното незадраскано или немаркирано число i е просто. Процесът продължава, докато не остане нито едно незадраскано число – тогава всички маркирани числа са прости, а всички задраскани съставни.

1. **Имплементация (Решето на Ератостен).**
2. Инициализираме масив checkArray [] с нули. По-късно когато задрасткваме някое число, на съответната позиция в масива ще записваме 1. Въвеждаме променлива-брояч i, която показва кое число разглеждаме в момента (т. е. ще сочи първото нездраскано или немаркирано число). Инициализираме i=2.
3. Увеличаваме i, докато checkArray [i] стане 0. Тогава числото i е просто и го отпечатваме.
4. Маркираме с 1 всички стойности checkArray [k], за k=i, 2i, 3i,… (n/i)i (т. е. всички кратни на i стойности).
5. Ако i≤n, то се връщаме на стъпка 2, иначе приключваме.
6. **Програмен код (Решето на Ератостен).**

int eratosthenesMethod()

{

int checkArray[MAX]= {0};

int n;

printf("Please input last number of the interval n=");

scanf("%d",&n);

printf("Primes in this interval:\n");

int j, i=2;

for(i=2; i<=n; i++)

{

if(checkArray[i]==0)

{

if(i==2)

printf("%d", i);

else

printf(", %d", i);

j=i\*i;

while(j<=n)

{

checkArray[j]=1;

j+=i;

}

}

}

return 0;

}

1. **Предимства и недостатъци (Решето на Ератостен).**

* Предимства:
  + 1. Проста имплементация.
    2. Сравнително добро бързодействие (В сравнение с алгоритъм, който проверява всяко число в интервала [2, n] дали е просто).
* Недостатъци:
  + 1. Необходимо е голямо количество последователно разположени клетки в паметта, за съхранение на масива с числа.

1. **Друг алгоритъм за генериране на прости числа.**

Започваме с празен масив, който ще попълваме последователно. Например, поставяйки в него първото просто число 2 можем да намерим всички прости числа в интервала [3, 22] – такова е 3 и го добавяме в масива. По-нататък, разполагайки вече с простите числа до 3, можем да намерим всички прости числа в интервала [4, 33] – това са 5 и 7, които също добавяме в масива. Продължаваме този процес докато в primes[] се добавят достътчно прости числа, за да покрият проверката дали число в интервала [2, n] е просто.

1. **Програмен код (ІІ-ри алгоритъм).**

void primesGenerator()

{

int n;

printf("Please input last number of the interval n=");

scanf("%d",&n);

n++;

printf("Primes in this interval:\n");

findPrimes(n);

}

int isPrime (int n)

{

int i=0;

while(i<pN && primes[i]\*primes[i]<=n)

{

if(n%primes[i]==0)

return 0;

i++;

}

return 1;

}

void findPrimes (int n)

{

int i=2;

while(i<n)

{

if(isPrime(i))

{

primes[pN]=i;

pN++;

if(i==2)

printf("%d", i);

else

printf(", %d", i);

}

i++;

}

}

1. **Предимства и недостатъци (ІІ-ри алгоритъм).**

* Предимства:
  + 1. Когато търсим прости числа в интревал [a, n] за достатъчно големи a (a>>1) алгоритъмът работи по-добре в сравнение с „Решетото на Ератостен“.Сравнително добро бързодействие (В сравнение с алгоритъм, който проверява всяко число в интервала [2, n] дали е просто).
    2. Няма нужда от голямо количество последователно разположени клетки в паметта.
* Недостатъци:
  + 1. По-сложна имплементация в сравнение с „Решетото на Ератостен“.

1. **Програмен код. (Цяла програма).**

//Алгоритми за генериране на прости числа. Решението на Ератостен.

#include <stdio.h>

#define MAX 46000

int eratosthenesMethod();

void primesGenerator();

void menu();

int primes[MAX];

int pN=0;

int main()

{

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*Welcome to Prime Number Generator!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

menu();

return 0;

}

void menu()

{

printf("\n====================Please choose a method====================");

short choice;

printf("\nPress \"1\" to generate primes using the Eratosthenes's method.\nPress \"2\" to generate primes using an other method.\nPress \"0\" to exit.\n");

for(;;)

{

printf("Your choice: ");

scanf("%d",&choice);

if(choice==1)

{

eratosthenesMethod();

break;

}

else if (choice==2)

{

primesGenerator();

break;

}

else if (choice==0)

{

printf("Goodbye!");

break;

}

else

printf("\nPlease make correct choice!\nPress \"1\" to generate primes using the Eratosthenes's method.\nPress \"2\" to generate primes using an other method.\nPress \"0\" to exit.\n");

}

}

int eratosthenesMethod()

{

int checkArray[MAX]= {0};

int n;

printf("Please input last number of the interval n=");

scanf("%d",&n);

printf("Primes in this interval:\n");

int j, i=2;

for(i=2; i<=n; i++)

{

if(checkArray[i]==0)

{

if(i==2)

printf("%d", i);

else

printf(", %d", i);

j=i\*i;

while(j<=n)

{

checkArray[j]=1;

j+=i;

}

}

}

return 0;

}

void primesGenerator()

{

int n;

printf("Please input last number of the interval n=");

scanf("%d",&n);

n++;

printf("Primes in this interval:\n");

findPrimes(n);

}

int isPrime (int n)

{

int i=0;

while(i<pN && primes[i]\*primes[i]<=n)

{

if(n%primes[i]==0)

return 0;

i++;

}

return 1;

}

void findPrimes (int n)

{

int i=2;

while(i<n)

{

if(isPrime(i))

{

primes[pN]=i;

pN++;

if(i==2)

printf("%d", i);

else

printf(", %d", i);

}

i++;

}

}

1. **Използвана литература.**

* <https://bg.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D1%8A%D0%BC#.D0.A1.D0.BF.D0.BE.D1.80.D0.B5.D0.B4_.D0.B8.D0.BC.D0.BF.D0.BB.D0.B5.D0.BC.D0.B5.D0.BD.D1.82.D0.B0.D1.86.D0.B8.D1.8F.D1.82.D0.B0>
* <https://bg.wikipedia.org/wiki/%D0%A0%D0%B5%D1%88%D0%B5%D1%82%D0%BE_%D0%BD%D0%B0_%D0%95%D1%80%D0%B0%D1%82%D0%BE%D1%81%D1%82%D0%B5%D0%BD>
* <https://bg.wikipedia.org/wiki/%D0%95%D1%80%D0%B0%D1%82%D0%BE%D1%81%D1%82%D0%B5%D0%BD>
* Книга „Програмиране=++Алгоритми“ с автори Преслав Наков и Панайот Добриков.