**Computer Algorithms**

**Assignment**

**submitted by,**

**Rushikesh Kawade**

**PRN: 2016BTEIT00025**

**Shell Sort**

• **Introduction:**

◦ In computer science, the**Sieve of Eratosthenes** is a simple, ancientalgorithm for finding allprime numbers up to few millions

• **Reason for this Algorithm:**

◦ Number Theory being an important part of computer science using Sieve of Eratosthenes can help us find prime numbers in an efficient way and also by backtracking we can get the value of factors of number

• **Description:**

To find all the prime numbers less than or equal to a given integer *n* by Eratosthenes' method:

Create a list of consecutive integers from 2 through *n*: (2, 3, 4, ..., *n*).

1. Initially, let *p* equal 2, the smallest prime number.
2. Enumerate the multiples of *p* by counting to *n* from 2*p* in increments of *p*, and mark them in the list (these will be 2*p*, 3*p*, 4*p*, ...; the *p* itself should not be marked).
3. Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this new number (which is the next prime), and repeat from step 3.
4. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below *n*.

Example:

To find all the prime numbers less than or equal to 30, proceed as follows.

First, generate a list of integers from 2 to 30:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

The first number in the list is 2; cross out every 2nd number in the list after 2 by counting up from 2 in increments of 2 (these will be all the multiples of 2 in the list):

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

The next number in the list after 2 is 3; cross out every 3rd number in the list after 3 by counting up from 3 in increments of 3 (these will be all the multiples of 3 in the list):

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

The next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after 5 by counting up from 5 in increments of 5 (i.e. all the multiples of 5):

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

The next number not yet crossed out in the list after 5 is 7; the next step would be to cross out every 7th number in the list after 7, but they are all already crossed out at this point, as these numbers (14, 21, 28) are also multiples of smaller primes because 7 × 7 is greater than 30. The numbers not crossed out at this point in the list are all the prime numbers below 30:

2 3 5 7 11 13 17 19 23 29

• **Implementation:**

/\* Program for Implementing Sieve of Eratosthenes \*/

// C++ program to print all primes smaller than or equal to

// n using Sieve of Eratosthenes

#include <bits/stdc++.h>

using namespace std;

void SieveOfEratosthenes(int n)

{

// Create a boolean array "prime[0..n]" and initialize

// all entries it as true. A value in prime[i] will

// finally be false if i is Not a prime, else true.

bool prime[n+1];

memset(prime, true, sizeof(prime));

for (int p=2; p\*p<=n; p++)

{

// If prime[p] is not changed, then it is a prime

if (prime[p] == true)

{

// Update all multiples of p

for (int i=p\*2; i<=n; i += p)

prime[i] = false;

}

}

// Print all prime numbers

for (int p=2; p<=n; p++)

if (prime[p])

cout << p << " ";

printf("\n");

}

// Driver Program to test above function

int main()

{

int n;

cout<<"Enter the number upto which you want to find prime number: ";

cin>>n;

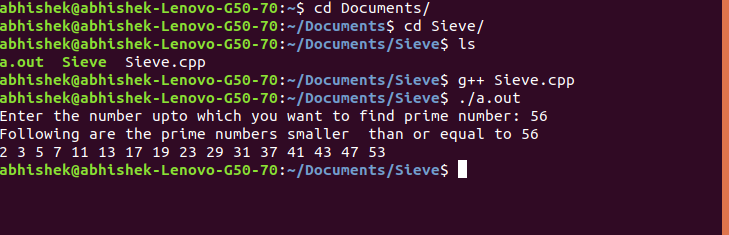
cout << "Following are the prime numbers smaller "<< " than or equal to " << n << endl;

SieveOfEratosthenes(n);

return 0;

}

• **Output:**

****

Time Complexity: O(log(log n))

• **Gprof Analysis:**

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

100.34 0.02 0.02 1 20.07 20.07 SieveOfEratosthenes(int)

0.00 0.02 0.00 1 0.00 0.00 \_GLOBAL\_\_sub\_I\_\_Z19SieveOfEratosthenesi

0.00 0.02 0.00 1 0.00 0.00 \_\_static\_initialization\_and\_destruction\_0(int, int)

% the percentage of the total running time of the

time program used by this function.

cumulative a running sum of the number of seconds accounted

seconds for by this function and those listed above it.

self the number of seconds accounted for by this

seconds function alone. This is the major sort for this

listing.

calls the number of times this function was invoked, if

this function is profiled, else blank.

self the average number of milliseconds spent in this

ms/call function per call, if this function is profiled,

else blank.

total the average number of milliseconds spent in this

ms/call function and its descendents per call, if this

function is profiled, else blank.

name the name of the function. This is the minor sort

for this listing. The index shows the location of

the function in the gprof listing. If the index is

in parenthesis it shows where it would appear in

the gprof listing if it were to be printed.

Copyright (C) 2012-2018 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,

are permitted in any medium without royalty provided the copyright

notice and this notice are preserved.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 49.83% of 0.02 seconds

index % time self children called name

<spontaneous>

[1] 100.0 0.00 0.02 main [1]

0.02 0.00 1/1 SieveOfEratosthenes(int) [2]

-----------------------------------------------

0.02 0.00 1/1 main [1]

[2] 100.0 0.02 0.00 1 SieveOfEratosthenes(int) [2]

-----------------------------------------------

0.00 0.00 1/1 \_\_libc\_csu\_init [16]

[9] 0.0 0.00 0.00 1 \_GLOBAL\_\_sub\_I\_\_Z19SieveOfEratosthenesi [9]

0.00 0.00 1/1 \_\_static\_initialization\_and\_destruction\_0(int, int) [10]

-----------------------------------------------

0.00 0.00 1/1 \_GLOBAL\_\_sub\_I\_\_Z19SieveOfEratosthenesi [9]

[10] 0.0 0.00 0.00 1 \_\_static\_initialization\_and\_destruction\_0(int, int) [10]

-----------------------------------------------

This table describes the call tree of the program, and was sorted by

the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the

index number at the left hand margin lists the current function.

The lines above it list the functions that called this function,

and the lines below it list the functions this one called.

This line lists:

index A unique number given to each element of the table.

Index numbers are sorted numerically.

The index number is printed next to every function name so

it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent

in this function and its children. Note that due to

different viewpoints, functions excluded by options, etc,

these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this

function by its children.

called This is the number of times the function was called.

If the function called itself recursively, the number

only includes non-recursive calls, and is followed by

a `+' and the number of recursive calls.

name The name of the current function. The index number is

printed after it. If the function is a member of a

cycle, the cycle number is printed between the

function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly

from the function into this parent.

children This is the amount of time that was propagated from

the function's children into this parent.

called This is the number of times this parent called the

function `/' the total number of times the function

was called. Recursive calls to the function are not

included in the number after the `/'.

name This is the name of the parent. The parent's index

number is printed after it. If the parent is a

member of a cycle, the cycle number is printed between

the name and the index number.

If the parents of the function cannot be determined, the word

`<spontaneous>' is printed in the `name' field, and all the other

fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly

from the child into the function.

children This is the amount of time that was propagated from the

child's children to the function.

called This is the number of times the function called

this child `/' the total number of times the child

was called. Recursive calls by the child are not

listed in the number after the `/'.

name This is the name of the child. The child's index

number is printed after it. If the child is a

member of a cycle, the cycle number is printed

between the name and the index number.

If there are any cycles (circles) in the call graph, there is an

entry for the cycle-as-a-whole. This entry shows who called the

cycle (as parents) and the members of the cycle (as children.)

The `+' recursive calls entry shows the number of function calls that

were internal to the cycle, and the calls entry for each member shows,

for that member, how many times it was called from other members of

the cycle.

Copyright (C) 2012-2018 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,

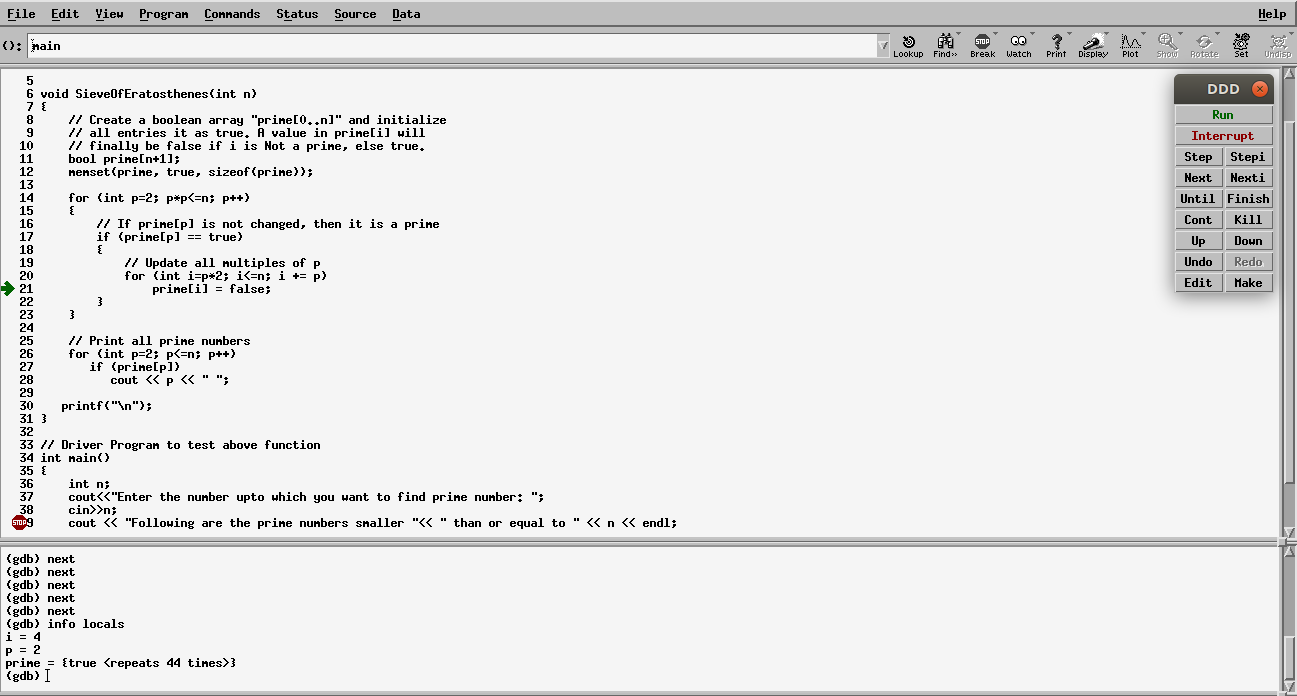
are permitted in any medium without royalty provided the copyright

notice and this notice are preserved.

Index by function name

[9] \_GLOBAL\_\_sub\_I\_\_Z19SieveOfEratosthenesi [2] SieveOfEratosthenes(int) [10] \_\_static\_initialization\_and\_destruction\_0(int, int)

**• DDD Analysis:**

****

• **Applications**

◦It can be used for fast and effective way of finding prime number

Using Backtracking we can use to find factors of number

Large Prime Number are used in the field of encryption

**Time Complexity**

O(n(log(logn))

