Прва крагујевачка гимназија

Семинарски рад из информатике

**Ератостеново сито линеарне сложености, динамичко програмирање и графови**

Ученик: Јанко Шуштершич

Професор: Зоран Васиљевић

Крагујевац, Јануар 2013

**Садржај**

1. **Ератостеново сито линеарне сложености** . . . . . . . . . . . . . . . . . 3
2. Опис алгоритма . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3. Реализација . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
4. Доказ исправности . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
5. Време и меморија . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
6. **Динамичко програмирање** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
7. Задатак ,,паркет'' . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
8. **Графови . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .**  9
9. Одређивање да ли је граф цикличан и проналажење циклуса . . . . 9
10. Реализација . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**Ератостеново сито линеарне сложености**

За задати природан број *n, n ≤ 107,* треба одредити све просте бројеве које припадају интервалу *[2, n].*

Класичан начин за решавање оваквог проблема је *Ератостеново сито*. Овај алгоритам је веома једноставан, али ради за време *О(n · log log n).*

Иако је тренутно познато много алгоритама линеарне сложености (тј. *О(n)* ) алгоритам описан у наставку је интересантан због своје једноставности – он уствари није нимало сложенији од класичног Ератостеновог сита.

Поред тога алгоритам је овде представљен као „споредан ефекат“, у ствари, он израчунава најмањег делиоца свих бројева из интервала *[2, n]*, што може бити применљиво у многим проблемима и задацима.

Мана алгоритма је вођена чињеницом да користи више меморије него класично Ератостеново сито: потребно је увести низ од *n* бројева, док је класичном Ератостеновом ситу довољно само *n-*битна меморија која може бити и до 32 пута мања.

Дакле, то је разлог зашто алгоритам има смисла применити на бројеве до *107*, не и на веће од њега.

Аутори овог алгоритма су Граис и Мисра (Gries, Misra, 1978. год). И, у ствари, назвати овај алгоритам „Ератостеново сито“ је погрешно, то су два веома различита алгоритма.

Опис алгоритма

Наш циљ је да за сваки број *i* из интервала *[2, n]* израчунамо најмањи делилац тог броја и доделимо га *i*-том члану низа *lp.*

Поред тога, потребан нам је и списак свих познатих простих бројева- низ *pr.*

У почетку, све вредности низа *lp* су попуњене нулама, што значи да предпостављамо да су сви бројеви прости. У току алгоритма низ ће се постепено попунити.

Пролазећи по свим бројевима од 2 до *n*, при чему је *i* тренутни број, можемо имати два случаја:

* *lp[i]=0,* што значи да је број *i* прост, тј. да нема делиоца мањих од *i.* Стога, неопходно је да се члану низа *lp[i]* додели *i,* а број *i* дода на списак *pr.*
* *lp[i]≠ 0,* што значи да је број *i* сложен, а његов најмањи прост делилац је *lp[i]*

У оба случаја након тога почиње процес ажурирања вредности низа *lp* помоћу мултипликатора *ј.* Међутим, наш циљ је да то урадимо тако да се сваки члан низа *lp* подешава највише једном.

То се може урадити помоћу формуле *,* где су прости бројеви мањи од *lp[i]* (зато нам је потребан списак свих простих бројева). Свим бројевима типа *lp[]* додељујемо нову вредност, која је очигледно једнака .

Зашто је овај алгоритам тачан и зашто ради у линеарном времену видећемо у наставку, а за сада ћемо представити реализацију.

Реализација

Ово решење је могуће мало убрзати, елиминишући вектор *pr* мењајући га са обичним низом, као и ослобађањем од друге *for*  петље. *N* можемо да прогласимо за константу.

|  |  |
| --- | --- |
| #include <vector>  #include <stdio.h>  using namespace std;  const int N = 10000000;  int lp[N+1];  vector<int> pr;  int main()  {  for (int i=2; i<=N; ++i)  {  if (lp[i] == 0)  {  printf ("%I64d",i);  lp[i] = i;  pr.push\_back (i);  }    for (int j=0;  j<(int)pr.size() &&  pr[j]<=lp[i] &&  i\*pr[j]<=N;++j)  lp[i \* pr[j]] = pr[j];  }  } | program esito;  const N=10000000;  var i,j,k:longint;  lp,pr:array[0..10000000] of longint;  begin  for i:=2 to N do  begin  if lp[i]=0 then  begin  lp[i]:=i;  inc(k);  pr[k]:=i;  write(i,' ');  end;  j:=0;  while (j<=k) and(pr[j]<=lp[i])  and (i\*pr[j]<=N) do  begin  lp[i\*pr[j]]:=pr[j];  inc(j);  end;  end;  end. |

Доказ исправности

Докажимо коректност алгоритма. Јасно је да ће сваком члану низа *lp* вредност бити додељена нијвише једном. То значи да алгоритам ради у линеарном времену, као и да остатак алгоритма очигледно ради у *О(n).*

Да бисте ово разумели имајте на уму да се сваки број *i* може представити овако:  *,* где је - као и раније, најмањи прост делилац броја *i*, а број број који нема делиоце мање од . Према томе

.

Сада, упоредимо ово са нашим алгоритмом. У њему се за свако набрајају сви прости бројеви који су мањи од и помноже се са како бисмо добили број наведеног облика.

Дакле, алгоритам ће сваком сложеном броју, тачно једном, доделити одговарајућу вредност.

То значи да је алгоритам исправан и да ради у линеарном времену.

Време и меморија

Иако је сложеност *О(n)* боља од сложености *О(n · log log n)* класичног Ератостеновог сита, разлика између њих није велика. У пракси ово значи двоструку разлику у брзини између оптимизованијих верзија Ератостеновог сита и алгоритма који је представљен овде.

Имајући у виду меморију коју овај алгоритам захтева, низ дужине *n* и низ простих бројева који је дужине приближнe , примећујемо да је овај алгоритам инфериоран у односу на класично сито.

Али, иде му у прилог то што низ који се попуњава и користи у самом алгоритму може послужити и за претраживање најмањег простог делиоца било ког броја из интервала *[2, n].*

Штавише, захваљујући трошку од само једног низа то може да се одреди не захтевајући било какво дељење.

Знати факторизацију свих бројева до *107* може бити корисна информација у неким задацима, а овај алгоритам је један од ретких који вам то омогућавају у линеарном времену.

**Динамичко програмирање**

Задатак „Паркет“

Задаци из динамичког програмирања који се често могу видети су:

* Пронађите број начина за поплочавање дате површине
* Пронађите поплочавање са најмњим/највећим бројем искоришћених плоча
* Пронађите најјефтиније поплочавање
* Пронађите поплочавање са минималним бројем плоча тако да је немогуће додати још једну плочу

Задатак: Дата је површина правоугаоног облика величине nxm. Пронаћи број начина да се ова површ поплоча плочицама 1х2, при чему несме бити празног простора, а плочице се не смеју лепити једна преко друге.

Потребно је направити матрицу *dp[i,mask],* где број врсте у површи и профил до последњег члана у врсти. Ако је ј-ти бит у 0 онда је он покривен, а ако је 1 онда није.

Овај алгоритам ће проћи по свим врстама и за сваку маску одредити број начина за покривање.

|  |  |
| --- | --- |
| #include <vector>  #include <iostream>  using namespace std;  int n, m;  vector <vector<long long>> d;  void calc (int x=0, int y=0, int mask=0, int next\_mask=0)  {  if (y >= m)  d[x+1][next\_mask]+= d[x][mask];  else  {  int my\_mask=1<< y;  if (mask & my\_mask)  calc (x,y+1,mask, next\_mask);    else  {  calc (x, y+1, mask, next\_mask | my\_mask);  if (y+1 < m && ! (mask & my\_mask) && !  (mask & (my\_mask << 1)))    calc (x, y+2, mask, next\_mask);  }  }  }  int main()  {  cin >> n >> m;  d.resize (n+1, vector<long long> (1<<m));  d[0][0] = 1;    for (int x=0;x<n;++x)  for (int mask=0; mask<(1<<m); ++mask)  calc (x, 0, mask, 0);    cout << d[n][0];  } | program p1;  var n,m,x,mask:integer;  d:array[0..100,0..100] of integer;  procedure calc(x,y,mask,next\_mask:integer);  var my\_mask:integer;  begin  if y>=m then inc(d[x+1,next\_mask],d[x,mask]) else  begin  my\_mask:=1<<y;  if (mask and my\_mask)>0 then calc(x,y+1,mask,next\_mask) else  begin  calc(x,y+1,mask,(next\_mask or my\_mask));  if (y+1<m) and ((mask and my\_mask)=0) and ((mask and (my\_mask<<1))=0) then  calc(x,y+2,mask,next\_mask);  end;  end;  end;  begin  read(n,m);  d[0,0]:=1;  for x:=0 to n-1 do  for mask:=0 to (1<<m) do  calc(x,0,mask,0);  writeln(d[n,0]);  end. |

**Графови**

Одређивање да ли је граф цикличан и

проналажење циклуса

Задатак: За усмерени или неусмерени граф одредити да ли је цикличан или није, и ако јесте, одредити циклус.

Алгоритам ради у сложености *О(m).*

Овај алгоритам прави серију претрага у дубину (или ширину) за сваки до сада не посећени чвор и уколико чвор наиђе на свог „оца“ то значи да је он цикличан, а уколико то не уради ни за један чвор онда је он ацикличан.

Циклус се може реконструисати пењањем уз „претке“.

Реализација

Овде је дато решење за један усмерени граф.

|  |  |
| --- | --- |
| int n;  vector < vector<int> > g;  vector<char> cl;  vector<int> p;  int cycle\_st, cycle\_end;  bool dfs (int v)  {  cl[v] = 1;  for (size\_t i=0; i<g[v].size(); ++i) {  int to = g[v][i];  if (cl[to] == 0)  {  p[to] = v;  if (dfs(to))return true;  }  else if (cl[to] == 1)  {  cycle\_end = v;  cycle\_st = to;  return true;  }  }  cl[v] = 2;  return false;  }  int main()  { ... ucitavanje grafa ...  p.assign (n, -1);  cl.assign (n, 0);  cycle\_st = -1;  for (int i=0; i<n; ++i)  if (dfs (i))  break;  if (cycle\_st == -1)  puts ("Acyclic");  else  {  puts ("Cyclic");  vector<int> cycle;  cycle.push\_back (cycle\_st);  for (int v=cycle\_end;  v!=cycle\_st; v=p[v])  cycle.push\_back (v);  cycle.push\_back (cycle\_st);  reverse (cycle.begin(),  cycle.end());  for (size\_t i=0;  i<cycle.size(); ++i)  printf ("%d ", cycle[i]+1);  }  } | program ciklgraf1;  var g:array [1..100,1..100] of boolean;  cl,p:array [1..100] of integer;  i,j,n,v,t,k,rez:integer;  procedure BFS(v:integer);  var i,b,poc:integer;  begin  cl[v]:=-1; p[1]:=v;  t:=1; k:=1;  while (t<=k) do  begin  poc:=p[t];  for i:=1 to n do  if (g[poc,i]) and (cl[i]<>0) then  begin  b:=poc;  while (b>0) and (rez>0) do  if cl[b]=i then rez:=0  else b:=cl[b];  end      else if (g[poc,i]) and (cl[i]=0) then  begin  k:=k+1;  p[k]:=i;  cl[i]:=poc;  end;  t:=t+1;  end;  end;  begin    rez:=1;  fillchar(cl,sizeof(cl),0);  for i:=1 to n do if cl[i]=0 then  BFS(i);  if rez=1 then  writeln('graf nije ciklican')  else writeln('graf je ciklican');  end. |