Алгоритми и нивна сложеност

Што се алгоритмите? Зошто воопшто се учат алгоритмите? Што е улогата на алгоритмите во споредба со другите технологии користени во компјутерите? Во продолжение, ќе ги одговориме овие прашања.

1. Што е алгоритам?

Зборот алгоритам (Algorithmi) е земен од латинскиот јазик и е даден по името на узбекистанскиот математичар од IX век Мухамед Ал Хорезми(Abu Jafar Mohammed Ibn Musa Al Khowarizmi), кој прв ги формирал правилата за извршување 4 основни операции со арапски цифри.

Неформално, алгоритам е било која добро-дефинирана сметачка процедура, што зема некоја вредност, или множество од вредности како влез и резултира некоја вредност, или множество од вредности како излез.

Решавајќи задачи од областа на математиката, физиката, статистиката и други ние користиме познати правила и методи. Како на пример, правило за множење на два броја, делење на броеви, методи за решавање на систем линеарни равенки и друго.

Исто така ќе го гледаме алгоритмот како алатка за решавање на добро специфициран сметачки проблем.

Алгоритмите се процедури за решавање на одредени проблеми. Многу едноставен пример е проблемот на множење на два броја. Множењето на мали броеви како што се 7 и 9 е тривијално, затоа што можеме да го мемориграме одговорот однапред. Но ако множиме големи броеви како што се 1234 и 789 ни треба чекор по чекор процедура или алгоритам. Сите ние во училиште сме учеле некои алгоритми кои биле составени од други помали алгоритми но не сме обрнувале внимание како тие процедури стручно се викаат и од што се составени, туку сме ги памтеле онакви какви што биле. Алгоритмите во сметањето имаат долга историја, можеби исто толку долга колку што постои цивилизацијата воопшто.

И во секојдневието извршуваме работи по некои правила. Ќе наведеме едноставен пример. Проблемот што треба да се реши е да се направи чоколадна торта по рецепт.

Рецепт	
Чоколадна торта	
400 гр чоколада	3 јајца
1 маргарин	1 пакетче ванила
2 чаши шеќер	1 чаша брашно

Стопи ги чоколадата и маргаринот.Истури го шеќерот во стопеното чоколадо и измешај.Потоа истури ги јајцата и ванилата и измешај.Истури го и брашното и измешај.Измешаното истури о во плех.Се пече на 250 степени околу 40 минути или додека вилушката кога ќе ја забодеме во тестото и извлечеме ќе биде скоро чиста.Се остава да се излади и потоа се јаде.

Програмскиот код за овој проблем е:

променливи

```
cokolada,margarin,seker,jajca,vanilla,brasno,izmesaj

izmesaj=stopi((400*cokolada)+margarin)
izmesaj=isturi(izmesaj+(2*seker))
izmesaj=isturi(izmesaj+(3*jajca)+vanila)
izmesaj= izmesaj+brasno
isturi(izmesaj)
sedodekanecista(viluska)
peci(izmesaj,250)
```

Значи и правењето торта може да се опише со алгоритам.

Пропишаните правила што ги користиме во врска со решавањето на одредената задача се вика алгоритам. Со други зборови алгоритмот се дефинира како конечно множество правила (инструкции) со кое се дефинира низа од операции со точно зададен редослед чиешто извршување е потребно за решавање на даден проблем.

Од дефиницијата за алгоритам можеме да заклучиме:

- алгоритамот завршува по конечен број на операции
- редоследот на операциите е точно зададен
  со што се добива решение на проблемот, што е и наша цел.

Секое поединечно дејство од множеството правила (инструкции) дефинирани во алгоритмот се нарекува алгоритамски чекор. Врз основа на ова можеме да кажеме дека алгоритмот се состои од низа алгоритамски чекори кои се извршуваат по однапред зададен редослед.

Пример1
Да се состави алгоритам за пресметување на плоштина на правоаголник со страни a и b.

Според дефиницијата ние треба да одредиме низа од дејства (алгоритамски чекори) така што со нивната примена врз податоците ќе го добиеме точниот резултат.

Најпрво тргнуваме од правилото за пресметување плоштина на правоаголник со страни a и b.
Формулата гласи: P=a·b
Сега ги дефинираме алгоритамските чекори:

Чекор 1. читање на вредностите на променливите a и b
Чекор 2. пресметување на вредноста на P по формулата P=a·b
Чекор 3. прикажување на вредноста на променливата P
Чекор 4. крај на алгоритмот

Да ги појасниме чекорите

Чекор1 е алгоритамски чекор за влез со кој на познатите променливи се доделуваат почетни вредности. Во алгоритмот на a и на b им се доделуваат почетни вредности.

Чекор 2 е алгоритамски чекор за пресметување на вредноста на променливата што е зададена со формула зависна од веќе познатите променливи. Во примерот се пресметува вредноста на P.

Чекор 3 е алгоритамски чекор за излез со кој се печатат или прикажуваат на екран вредностите на одделни променливи. Во примерот се прикажува вредноста на променливата P.

Мора да напоменеме дека не може да се наведе алгоритамски чекор за пресметување на вредноста на променливата ако не се познати вредностите врз основа на кои таа се пресметува. Исто така ни алгоритамски чекор за прикажување на вредност на променлива, на која не и е доделена или пресметана вредноста.

За еден алгоритам велиме дека е точен, ако за секоја внесена инстанца завршува со точен излез. Велиме дека точен алгоритам го решава дадениот проблем.

Алгоритмот мора да се направи така да биде испланиран да се извршува од човек или од машина. Во општ случај, алгоритмот мора да се формулира во чекори, доволно едноставни за да се извршат од човек или машина (компјутер).

Алгоритмите имаат многу заеднички работи со програмите, но постојат и значајни разлики меѓу нив.

Прво, алгоритмите се поопшти од програмите. Еден алгоритам може да биде решен од човек или машина, или од двете. Програмата мора да биде извршена од компјутер.

Второ, алгоритмите се поапстрактни од програмите. Еден алгоритам може да биде изразен во било кој конвенционален јазик или нотација, а програматамора да биде изразена во некој програмски јазик.

Ако алгоритмот имаме намера да го извршиме на компјутер, прво треба да го кодираме во одреден програмски јазик и може да бираме во кој програмски јазик сакаме.

Сортирањето е без сомнение единствениот сметачки проблем за кој алгоритмот бил развиван.

Секој алгоритам треба да ги има следниве карактерични делови:

- влез
- излез
- дефиниција
- ефективност
- крај

2. Претставување на алгоритмот

Алгоритмот може да се прикаже на два начини:

- текстуално
- графички

Преку пример ќе ги разгледаме двата начина на преставување на алгоритмот.

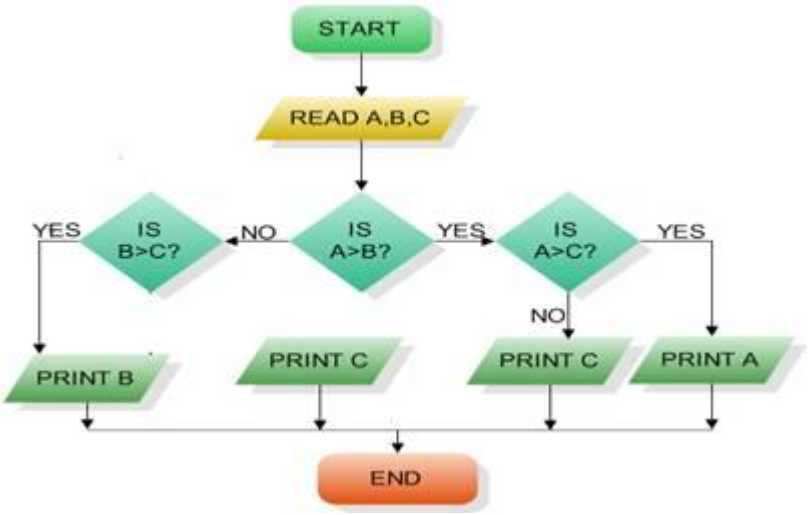
Пример2
Да се напише алгоритам за наоѓање на најголем број од три внесени броја.

Текстуално:

```
алгоритам
Најголем;
почеток
          читај a,b,c;
           ако a>b
                     тогаш
                                    p←a;
                     инаку
                                    p←b;
          крај_ако(a>b)
           ако p>c
                      тогаш
                                     n←p
                    инаку
                                     n←c
          крај_ако(p>c)
           печати n;
крај(Најголем)
```

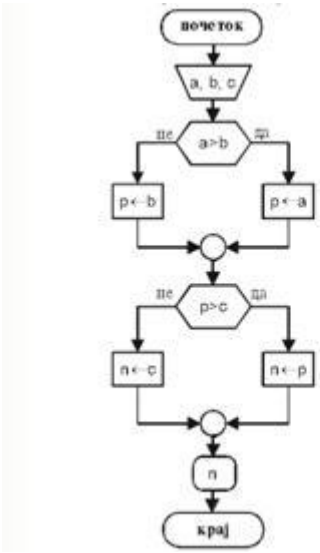
Во табелата горе е претставен текстуално алгоритмот за дадениот пример. Можеме да воочиме дека претставувањето се врши во чекори како што е кажано предходно. Се користи т.н. псевдојазик за опишување на алгоритмите чии зборови во примерот се со задебелени букви.

Графички:



Слика 1

На слика 1 е дадено графичко преставување на алгоритмот со помош на блок-дијаграм. Во блок-дијаграмот се користат посебни графички симболи за одредени дејства (операции).



Графичкото преставување има свои предности и недостатоци. Предноста во однос на текстуалното преставување е во поголемата прегледност на текот на дејствата во алгоритмот, бидејќи човекот подобро перцепира (осознава) слика од текст.

Бидејќи често се прави измена на првичниот алгоритам, полесно е таа да се направи кога алгоритмот е преставен графички, а и подоцна истата промена полесно може да ја увиди секој кој би го погледал алгоритмот.

Но од друга страна, графичкото преставување е непогодно за поголем алгоритам кој може да зафати повеќе страници, во кои тешко би се снаоѓале. Со графичкото преставување поголем проблем може да се сведе на помали елементарни проблеми кои би се поделиле на повеќе луѓе за да ги решат и потоа да се имплементираат во еден алгоритам кој го решава целиот проблем.

3. Својства на алгоритмите

При решавањето на даден проблем, треба да се води сметка за тоа на кој начин е напишан алгоритмот т.е.по кој редослед се извршуваат операциите во него.

При негово пишување треба особено да се внимава на:

- алгоритмот да има само еден почеток
- алгоритмот да има само еден крај
- во него да нема бесконечни циклуси
- да нема делови (чекори) кои никогаш не се извршуваат

Пример 3

```
алгоритам X;
почеток
          читај a,b;
          додека a<b извршувај
                    c←a*b;
                    печати a,b,c;
          крај_додека{a<b}
          печати a,b;
крај{X}
```

Пример 4

```
алгоритам Y;
почеток
      читај m,n;
       ако m≥n
              тогаш
                   скок на oznaka
               инаку
                     ако m<n
                         тогаш
                               печати m,'<',n;
                               скок на oznaka
                          инаку
                               печати m,'=',n;
                      крај_ако{m<n}
        крај_ако{m>n}

  oznaka: печати 'Каде е грешката?';
крај{Y}
```

Дадени се алгоритми во пример 3 и 4. Кој од нив не е правилен алгоритам? Зошто?

(Одговор:Алгоритмот од пример 3 не е правилен, бидејќи ако бројот a е помал од бројот b, тогаш циклусот ќе се извршува бесконечно и никогаш нема да се добие резултат (решение).

Исто така и алгоритмот во пример 4 не е правилен, бидејќи во него постои дел кој никогаш не се извршува. Делот печати m,'=',n; во внатрешната ако-тогаш-инаку контролна структура никогаш нема да се изврши.)

За даден проблем може да се напишат различни алгоритми и сите тие да бидат правилни. Ако се правилни секој од нив ќе даде точно решение. Но тоа не значи дека сите тие алгоритми се подеднакво ефикасни. Некој можеби ќе биде побрз, друг да дава поточни резултати, трет да има пократок запис или да биде појасен итн.

Но секој од тие алгоритми треба да ги има следниве особини:

- секоја операција е точно определена
- во алгоритмот е прецизно утврден редоследот на извршувањето на операции
- алгоритмот може да се расчлени на елементарни чекори
- дава резултати по конечен број чекори
- со алгоритмот се опфатени сите можни решенија за различни вредности на влезните податоци
- не зависи од тоа дали се извршува од човек или на сметач,и тоа независно од типот на сметачот
- да е разбирлив за секој,независно кој го напишал

Споменавме дека може да се напишат повеќе правилни алгоритми за ист проблем. Тогаш се поставува прашањето кој алгоритам да се користи? Кој алгоритам е подобар?
Лесно е да кажеме овој алгоритам е подобар од другиот, но како сме дошле до тоа сознание, зошто едниот е подобар од другиот алгоритам?
Овие прашања имаат одговор кој води до анализа на алгоритмот и одредување на неговата сложеност.

4. Анализа на алгоритмот

Претпостави дека компјутерите се бесконечно брзи и компјутерската меморија слободна.Дали би имале било каква причина да учите алгоритми? Одговорот е да, ако не за друго да докажите дека вашето решение завршува и го дава точниот одговор.

Ако компјутерите се бесконечно брзи, секој точен метод за решавање на одреден проблем ќе работи. Вие сигурно сакате вашата имплементација да биде добро дизајнирана и документирана но многу често го користите методот што е најлесен за имплементација.

Секако, компјутерите можеби се брзи, но не се бесконечно брзи. И меморијата можеби е ефтина но не е неограничена. Времето за пресметка е исто така ограничувачки ресурс, како што е просторот во меморијата. Овие ресурси треба да се користат мудро, и алгоритмите ќе бидат поефикасни во однос на времето и меморијата.

Анализата на алгоритмот овозможува да се даде квантитативна оценка за предноста на еден алгоритам во однос на друг алгоритам за ист проблем. Оценката може да преставува како алгоритмот се однесува во:

- најдобар случај
- најлош случај
- просечен случај

Значи анализата на алгоритмот се прави за да може полесно да се спореди со друг алгоритам за истиот проблем. Со анализирањето на алгоритмите и определување на најдобриот алгоритам од понудените за тој проблем значително се заштедува на време и пари и се добива поголема ефикасност во решавањето на зададените задачи.

Ќе се запрашаме што точно се оценува кај еден алгоритам? На кој начин се споредуваат два алгоритми?
Еден начин за споредба на алгоритмите е да се споредат кој побрзо го решава дадениот проблем. Некои доаѓаат побрзо до решението од другите. Најчесто и ние го одбираме алгоритмот што најбргу доаѓа до решението.

Друг начин за споредба на алгоритмите е да се види колку мемориски простор е потребен при извршување на алгоритмот напишан во програмски јазик. Некои од нив зафаќаат повеќе мемориски простор, но бргу доаѓаат до решението, а други помалку мемориски простор, а до решението доаѓаат поспоро.

Значи сложеност е ниво на тежина при решавање на одреден проблем мерен во време, број на чекори или аритметички операции, меморија.

Ние ќе се задржиме на временската сложеност на алгоритмот.
Времето што е потребно за решавање на одреден проблем зависи од одреден број на фактори:

- колку е брз компјутерот
- капацитетот на RAM-от на компјутерот
- оперативниот систем
- квалитетот на кодот генериран од компајлерот итн.

Ако некој од факторите се промени тогаш и времето на извршување се менува.

Затоа оваа големина не е доволно добра како мерка за алгоритамските перформанси. Нас ни е потребна мерка со која ќе може да споредиме 2 алгоритми.

Анализата на алгоритмот се состои од 2 фази:

- претходна анализа
- тестирање

Во првата фаза, претходна анализа, се формира функција на сложеност зависна од релевантни фактори која го определува времето на извршување. Функцијата всушност преставува оценка на алгоритмот кога тој не зависи од тоа на каков компјутер се извршува.

На кој начин се прави тоа?

Секој алгоритам се состои од конечен број инструкции. Колку повеќе иструкции, толку подолго тој ќе се извршува. Еден начин е да се бројат инструкциите (операциите) од кој е составен алгоритмот за решавање одреден проблем. Потребно е да се знае колку пати се извршува инструкцијата и која е “нејзината цена” т.е. времето (честота) на едно извршување на инструкцијата.

Нека се дадени следниве програмски сегменти:

```
а) x = x + y

б) For i = 1 to n do
      x = x + y
   end
```

```
в) For i = 1 to n do
     For i =1 to n do
   x = x + y
     end
      end
```

а) честота на извршување е 1
б) честота на извршување е n
в) честота на извршување е n^2

Бројот на инструкции ќе варира зависно од бројот на внесените податоци. Програма за пресметка на плати, за влез од 100 лица ќе изврши повеќе инструкции одколку за 10 лица. Затоа може да го изразиме бројот на инструкции како функција на сложеност од бројот на внесени податоци.

Пример 5

```
Алгоритам Збир                1
 Почеток                      1
   читај n;                     1
    s←0;                         1
   за i← 1 зголемувај до n       n
        s←s+i;                   n
    крај_за(i)                    1
  печати s;                      1
крај(Збир)                       1
```

Функцијата што го преставува бројот на инструкции за овој алгоритам може да се запише како f(n)=2n+7. Оваа функција на сложеност е линеарна, што значи и сложеноста на овој алгоритам е линеарна. Но мора да напоменеме дека оваа функција е зависна од сметачот и програмскиот јазик во кој се извршува.

Нека f(n) е функција на сложеност што зависи од компјутерот на кој се извршува алгоритмот, а g(n) функција на сложеност што не зависи од тоа на каков компјутер се извршува алгоритмот. Тогаш важи f(n)=O(g(n)).

Кога ќе се каже ...Алгоритмот има O(g(n)) време на пресметување (извршување)... Тоа значи дека ако алгоритмот се извршува повеќе пати на иста машина, на ист тип податоци, но за различно (растечко) n, времето на извршување секогаш ќе биде помало од некоја константна вредност |g(n)|.

Значи функцијата на сложеност која не зависи од тоа на каков сметач се извршува алгоритмот во примерот 5 е g(n)=n, и велиме алгоритмот има сложеност O(n) (се чита “од ред n ”).

Но дали треба да ги земеме во предвид сите инструкции во алгоритмот?

Најчесто алгоритмите се состојат од циклус и инструкции што се извршуваат во него или надвор од него. Бројот на инструкциите надвор од циклусот во алгоритми што се однесуваат за ист проблем незначително се разликува.

Од нив не зависи дали алгоритмот ќе има помала или поголема сложеност. Ако бројот на влезни податоци расте тогаш бројот на инстукции што се извршуваат во циклусот го надминува бројот на оние што се надвор од него.

За да ја упростиме нашата пресметка нема да ги броиме инструкциите надвор од циклусот.

Дали треба да ги земеме во предвид сите инструкции во циклусот?

Бројот на инструкции во циклусот незначително се разликуваат во 2 алгоритми напишани за ист проблем. Но, тоа што се разликува е колку пати циклусот се извршува.

Пример 6

```
алгоритам NajdiElement
  почеток
     читај n,baran;
      за i=1 зголемувај до n
          читај a[i];
      крај_за(i)
      i=1;
      додека ((iѕn) и (baran ≠ a[i] ))
         i=i+1;
      крај_додека((iѕn) и (baran ≠ a[i] ))
     ако i=n
             тогаш
                 печати "Елементот не е во низата";
              инаку
                 печати "Елементот се наоѓа на ",i,"-тата
позиција";
 крај(NajdiElement)
```

Потребно е да го пресметаме бројот на извршувања на циклусот. Во алгоритам кој пребарува низ низа елементи, во циклусот се споредува тековниот елемент од низата со елементот што треба да се најде.

Во овој случај го бараме бројот на споредувања на елементите во дадениот алгоритам. Но бројот на споредби пак зависи од должината на низата. Нека низата има должина n, во алгоритам за линеарно пребарување. Бројот на споредувања ќе биде n кога елементот што се бара не е во низата. Тогаш велиме дека алгоритмот има сложеност O(n) или линеарна сложеност.

Ако ја зголемиме двојно низата, двојно се зголемува и бројот на споредби во алгоритмот. Но некогаш имаме среќа да го најдеме елементот блиску до крајот, а некогаш блиску до почетокот на низата. Во тие случаи ни треба да ја пребараме половина од низата за да го најдеме тој елемент. Па тогаш би имале O(n/2) споредувања.

Ова се нарекува просечна сложеност на алгоритмот. Ако пак елементот не се наоѓа во низата, се прават n споредувања и ова се нарекува најлош случај на сложеност на алгоритмот. Но да забележиме дека сепак сложеноста на алгоритмот е O(n) наспроти фактот дека во просек се прават n/2 споредби.

Ова е поради тоа што ако n расте многу големо, разликата помеѓу n и n/2 станува се помалку значајна. Кога се споредува алгоритам важно е да се знаат и двете сложености (просечна и најлошиот случај).

Ќе опишеме друг алгоритам за барање низ низа наречен бинарно пребарување (Binary Search) каде бројот на споредувања е значајно помал. Тој изнесува O(log n).

Пример 7

```
алгоритам БинарноБарање
  почеток
        печати 'Внеси број на елементи во низата';
        читај n;
        печати 'Внеси ги елементите';
         за i=1 зголемувај до n
                читај ai;
         крај_за(i)
         печати 'Внеси ја вредноста што се бара';
         читај v;
         levo ← 1;
         desno ← n;
         повторувај
                   sredina ← [(levo+desno)/2];
                   ако v>asredina
                         тогаш
                                levo ← sredina+1;
                         инаку
                                desno ← sredina-1;
                   крај_ако( v>asredina)
         до asredina=v или levo>desno;
         крај_повторувај
         ако asredina=v
                  тогаш
                        печати 'Елемент со таква вредност
e',asredina;
                  инаку
                        печати 'Елемент со таква вредност не
постои';
           крај_ако(asredina=v)
 крај(БинарноБарање)
```

Ако имаме низа со должина 1000 со бинарно пребарување би го нашле бараниот елемент во најлош случај со 10 споредби (обратно од линеарното пребарување). Но низа со должина 1 билион во најлош случај го наоѓа елементот по 30 споредувања. Очигледно е дека бинарно пребарување е со помала сложеност од линеарното пребарување.

Бинарното пребарување се состои од делење на низата на половина и воочување во која од двете добиени низи е бараниот елемент. Потоа таа низа повторно се дели на половина инт., се додека не се пронајде елементот или пак додека не се воочи дека елементот не е во низата.

Во најлош случај бинарното пребарување престанува кога нема да го најде елементот во низата. По i-тото делење на низата на пола, кандидати за елементот што се бара се најмногу n/2i. Кога ќе заврши процесот i е најмалиот цел број така што (n/2i)log n. Значи алгоритмот се изведува во време O(log n).

Бинарно пребарување има ограничено користење т.е. само кога низата е сортирана. Тоа е метод кој го користиме кога листаме телефонски именик. Отвараме на “средина” и во зависно од тоа каде сме отвориле знаеме дали бројот е во едната половина од именикот или во втората половина. Со ова елиминираме половина од именикот со едно споредување. Оваа постапка се повторува на делот што останува додека не се најде бројот или додека не се утврди дека не се наоѓа во низата, со што секоја натамошна споредба ја намалува низата за половина.

Но како што рековме за да го користиме овој метод низата треба да е сортирана. Алгоритмите за пребарување и сортирање се повеќе се користат. Па всушност скоро во секој алгоритам за поголем проблем има сортирање и пребарување.

Ќе покажеме алгоритам за сортирање наречен метод на меурче (Bubble Sort).

```
алгоритам СортирањеСоМетодНаМеурче
  почеток
        печати 'Внеси број на елементи во низата';
        читај n;
        печати 'Внеси ги елементите';
         за i=1 зголемувај до n
                читај ai;
         крај_за{i}
          brojac ←2;
         повторувај
                   за i=n намалувај до brojac
                     ако ai<ai-1
                        тогаш
                              pom ←ai;
                              ai ←ai-1;
                              ai-1 ←pom;
                     крај_ако{ai<ai-1}
                   крај_за{i}
              brojac=brojac+1;
         до brojac=n;
         крај_повторувај
         печати 'Сортираната низа е';
         за i=1 зголемувај до n
                печати ai;
         крај_за{i}
 крај{СортирањеСоМетодНаМеурче}
```

Има многу начини да се сортира низа. Подолу ќе ја објасниме метода на меурче(Bubble Sort) и ќе ја покажеме нејзината сложеност. Ќе ја сортираме низата во растечки редослед. Нека низата е: 10, 34, 2, 16, 23, 8, 12. Сакаме да добиеме 2, 8, 10, 12, 16, 23, 34. За да ја сортираме мора да ги знаеме вредностите на елементите во низата, така што елементите со помала вредност ќе ги заменат местата со оние со голема вредност, со што големите вредности ќе дојдат на крајот, а малите на почетокот на низата.

Сакаме да ги поместиме големите вредности на крајот од низата. Може да споредуваме само 2 вредности од низата. Во овој метод почнуваме од почетокот на низата. Ги споредуваме првите 2 елементи L[0] и L[1].Ако L[0]>L[1] тогаш смени им ги местата. Потоа ако L[1]>L[2] тогаш смени им ги местата итн. Ова го поместува најголемиот елемент на крајот на низата. Другиот дел од низата не е сортирана. Сме поминале низ низата еднаш.

Низата ја има формата 10, 2, 16, 23, 8, 12, 34. Потоа почнуваме пак од почеток со горенаведената постапка. И се така до моментот кога ќе ја прођеме низата по n-ти пат ако таа има должина n. По завршувањето на методот низата е следна: 2, 8, 10, 12, 16, 23, 34 и е сортирана. Оваа метода се нарекува така затоа што елементот се движи кон крајот на низата како меурче во чаша вода за да излезе на површина. Поголемото се движи побрзо кон површината.

Задача:

Со метод на меурче да се сортира низата 2,4,15,6,7,9,10. Колкава е сложеноста на алгоритмот во овој случај?

Сложеноста на метод на меурче за низа од n елементи е $O(n^2)$. Зошто?
Низа од n елементи, со овој метод ја поминуваме n пати, а во секое поминување на низата правиме n споредби на елементите. Од таму сложеноста на овој алгоритам е $O(n^2)$.

Досега видовме сложеност на 3 алгоритамски методи:

- Линеарно пребарување O(n)
- Бинарно пребарување O(log n)
- Метод на меурче O(n2)

Сложеноста на алгоритмите со низи е најмалку O(n) бидејќи скоро секогаш мора да поминеме барем еднаш низ низата. Бинарното пребарување е на некој начин исклучок. Предноста е во тоа што не мора да се поминува целата листа која е сортирана. Но што ако низата не е сортирана? Тогаш или ќе користиме линеарно пребарување што се користи и за несортирани и сортирани низи или прво ќе ја сортираме низата, а потоа ќе употребиме бинарно пребарување. Линераното пребарување ќе биде подобро од комбинацијата сортирање и бинарно пребарување. Алгоритам со сложеност O(n) е поефикасен алгоритам во споредба со метод на меурче со сложеност O(n²).

Според сложеноста на алгоритмите разликуваме:

- константни O(1)
- линеарни O(n)
- логаритамски O(log2 n)
- линеарно-логаритамски O(n·log2 n)
- квадратна O(n2)
- експоненцијални O(2n) (n>1)
- факториелни O(n!)

Колку помала сложеност толку поголема ефикасност има алгоритмот. Нормално ние ќе го избереме оној алгоритам со најмала сложеност. Од горе наведените алгоритми најефикасни се логаритамските, а видовме и зошто.
Постојат голем број на линеарни алгоритми кои се ефикасни,а квадратните алгоритми имаат добра ефикасност.

Експоненцијалните и факториелните алгоритми имаат многу мала ефикасност и се користат само кога мора. Постојат проблеми за кои има пронајдено само експоненцијални или факториелни алгоритми. Наоѓањето на алгоритам со помала сложеност за таков проблем е голем успех.

Втората фаза од анализата на алгоритмот е неговото тестирање на машина во некој програмски јазик. Многу од програмите кои се користат содржат багови (грешки). Секој (чесен) програмер ќе ги преброи глупавите, смешни и сериозни грешки што ги направил во програмата. Дури и ќе се обиде да ги поправи и во секое наредно пишување на алгоритам за било кој проблем ќе гледа да не ги повтори. 70% од конструирањето на нов комплексен софтвер се троши за поправање на грешките. Почетниците најчесто веруваат дека нивните алгоритми го прават токму она што тие сакаат да го направат и дека тие алгоритми се најефикасните.

При тестирањето не може да се задржиме на функцијата на сложеност од претходната анализа, туку треба да се води сметка и за константите што се јавуваат во таа функција.

Прашање:

Ако имаме два алгоритми кои истата задача ја извршуваат со n влеза, и првиот има време на пресметување O(n), а вториот O(n2), кој е побрз?

Со сигурност ќе тврдиме дека првиот е побрз. Лесно е да се забележи дека за доволно големи вредности на n, времето на извршување на вториот алгоритам ќе биде поголемо од времето на извршување на првиот.

Нека вистинско време на извршување на алгоритмот на сметач е 2n и n2 соодветно.Тогаш првиот алгоритам е побрз за сите n>2.

Проблем: Што ако 104n и n2 се соодветните вистински времиња на извршување?

Тогаш вториот алгоритам е побрз за сите n104, побрз е првиот алгоритам. Значи не може да одлучиме кој алгоритам е побрз, ако не знаеме ништо за константите поврзани со функцијата на сложеност.

Методите во оваа фаза може да се групираат во две категории: тестирање и докажување. Тестирањето се состои од извршување на програмата за разни влезни податоци и дали ќе даде точен резултат за тие податоци.Тестирањето се врши само над одбрано множество влезни податоци. Докажувањето на програмата значи дека програмата е исправна за сите дозволени влезни податоци (од различен тип), дури и за голем број на влезни податоци. Тестирањето повеќе се користи бидејќи е полесна техника за проверка на исправноста на алгоритмот. Докажувањето на алгоритмот се користи за резонирање на однесувањето на програмата. Со него може да се зголеми довербата во исправноста на програмата.

5. Ефикасност

Алгоритмите што се смислени да решат ист проблем често драматично се разликуваат во нивната егикацност. Овие разлики можат да бидат позначајни од разликите во хардверот и софтверот.
Како за пример, ќе видиме два алгоритми за сортирање. Првиот, или уште попознат како инсертион сорт, зема време n2 за да сортира n работи. Вториот, merge sort, истите работи ќе ги сортира за време log2n. Вреди да се спомене тоа што insertion sort е обично побрз од merge sort за мали вредности на n, но кога бројот на внесени податоци доволно ќе нарасне merge sort е убедлив во однос на insertion sort.
За конкретниот пример, нека имаме компјутер А којшто е побрз и ќе работи со insertion sort и компјкутер Б којшто е поспор и ќе работи со merge sort. Нивна задача е да сортираат низа од 1000000 броеви.

Да претпоставиме дека компјутерот А извршува 1 билион инструкции во секунда, а компјутерот Б извршува само 10 милиони инструкции во секунда. Значи компјутерот А е 100 пати побрз од компјутерот Б. За да ја направиме разликата уште повеќе драматична нека константите ц1 за компјутерот А биде 2, а ц2 за компјутерот Б биде 50. За да сортира еден милион броеви, на компјутерот А ќе му требаат

$$\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}} = 2000 \text{ seconds},$$

додека на компјутерот Б ќе му требаат:

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 100 \text{ seconds}.$$

Со користење на алгоритам чиешто вреем на извршување расте многу поспоро, дури и со послаб компајлер, компјутерот Б работи 20 пати побрзо отколку компјутерот А! Предноста на merge sort ќе биде уште поизразена кога би сортирале 10 милиони броеви: кога на insertion sort би му требале приближно 2,3 дена, додека на merge sort помаалку од 20 минути. Генерално, колку големината на проблемот се зголемува, толку се золемува предноста на merge sort.

Псевдокод на insertionSort во Pascal

```
insertionSort(array A)

begin

    for i := 1 to length[A]-1 do

    begin

        value := A[i];

        j := i-1;

        while j ≥ 0 and A[j] > value do

        begin

            A[j + 1] := A[j];

            j := j-1;

        end;

        A[j+1] := value;

    end;

end;
```

Псевдокод на insertionSort во Java

```
InsertionSort(A) **sort A[1..n] in place
    for j = 2 to n do
        key  = A[j] **insert A[j] into sorted sublist A[1..j - 1]
        i  = j - 1
        while (i > 0 and A[i] > key) do
            A[i+1] =  A[i]
            i  = i - 1
            A[i+1]  = key
```

Псевдокод на mergeSort во Pascal

```
MergeSort(List[], leftIndex, rightIndex)
Begin:
    if leftIndex < rightIndex then
        mid = (leftIndex + rightIndex) / 2
        MergeSort(List[], leftIndex, mid) // split left sublist
        MergeSort(List[], mid + 1, rightIndex) // split right sublist
        Merge(List[], leftIndex, mid, rightIndex) // merge sorted
sublists
    endif
End
```

Псевдокод на mergeSort во Java

```
function merge(left[], right[], output_tape[])

     do

        if left[current] ≤ right[current]

            append left[current] to output_tape

            read next record from left tape

        else

            append right[current] to output_tape

            read next record from right tape

    while left[current] < left[next] and right[current] < right[next]

    if left[current] < left[next]

        append current_left_record to output_tape

    if right[current] < right[next]

        append current_right_record to output_tape

    return
```

Задачи:

1.Мирко се пријавил на конкурс за работа на проект. На интервјуто на кое бил повикан, работодавецот му објаснил дека ќе го плаќа на следен начин:

- прв ден 1 евро
- втор ден 2 евра

  ……

Мирко очекувал од проектот да заработи најмалку 200 евра, во спротивно не би ја прифатил работата. Проектот трае n денови. Помогнете му на Мирко да пресмета дали му се исплати да работи на проектот.

Решение:

```
Алгоритам Заработка                    1
   почеток                               1
    читај n;                             1
   s← n*(n-1)/2;                          1
   ако s<200                             1
      тогаш
       печати "Не ја прифаќам работата"; 1
      инаку
       печати "Ја прифаќам работата";    1
    крај_ако(s<200)
  крај(Заработка)
```

Значи f(n)=7,бидејќи кога алгоритмот се извршува на сметач секоја инструкција има определено време на извршување. Сега треба да го одредиме најмалото g(n) за да важи f(n)=O(g(n)). Согледуваме дека најмалото g(n)=1 и запишуваме дека алгоритмот има сложеност O(1) (се чита “од ред 1” т.е. константна сложеност.)

2. За дадена низа a да се создаде низа b така што i-тиот елемент во низата b е аритметичка средина на елементите до i-тата позиција во низата a.

Решение:

```
Алгоритам НизаОдНиза          1
 почеток                      1
  читај n;                     1
  за i←1 зголемувај до n       n
   читај ai;                   n
  крај_за{i}
  b1 ←a1;                     1
  за i← 2 зголемувај до n      n-1
   bi ←(bi-1 * (i-1)+ai)/i;          1
  крај_за{i}
   за i←1 зголемувај до n       n
    печати bi;                  n
  крај_за{i}
крај {НизаОдНиза}
```

Значи f(n)=1+1+1+n+n+1+(n-1)+1+n+n)=5*n+4. Најмалото g(n)=n, па имаме сложеност O(n) т.е. линеарна сложеност.

3. Следните Java методи имплементираат собирање и множење на матрици. Секоја матрица е претставена со n x n дво-димензионални низи од float броеви

```
static void matrixAdd (int n, float[][] a, float[][] b, float[][] sum) {
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            sum[i][j] = a[i][j] + b[i][j];
        }
    }
}
static void matrixMult (int n, float[][] a, float[][] b, float[][] prod){
    for(int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            float s = 0.0;
            for (int k = 0; k < n; k++){
                s += a[i][k] * b[k][j];
            }
            prod[i][j] = s;
        }
    }
}
```

Анализирајте ги овие методи. Која е комплексноста на секој метод?

(Одговор: На собирањето е $O(n^2)$, а на множењето е $O(n^3)$.)