1.Основни понятия.Варианти за алгоритми.Влияние върху производителността.Въведение в анализа.

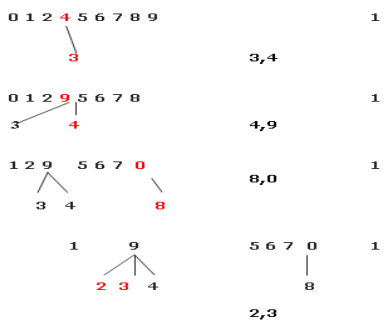
Алгоритъмът показва как се решава задачата, начинът за решаването й – чрез изпълнение на каква последователност от инструкции тя се решава.Алгоритъмът е последователност от инструкции (указания, команди, стъпки, оператори) за действие (или действия), която при изпълнението си реализира зададена функционална зависимост между данните и резултатите. Алгоритъмът може да се представи и като ориентиран граф: възлите съответстват на операциите, а дъгите – на предходите – те показват кои да са следващите операции в алгоритъма. Алгоритъмът може да се разглежда като преобразувател на данни – при изпълнението си алгоритъмът преобразува данните от едно представяне в друго, докато се получат резултатите.
Примери за алгоритми: Начини за извършване на аритметични действия, Числено решаване на системи линейни уравнения (по метод на Гаус, метод на проста итерация,итерационен метод на Зайдел),Числен метод за решаване на определен интеграл
Алг. за анализ: Анализ на честотен или преходен режим на една ел. верига, алг. за анализ и синтез на комбинационни логически схеми,алг.в каква последователност да се обадим по телефона,упътване как да се придвижим от т.А до т.Б.
При разработване на всеки алгоритъм трябва да се направи анализ на задачата която той решава, за да се определят свойствата, които има тази задача, с цел те да се използват в алгоритъма.
Запис на алгоритмите: Словестно, с блокова схема, с език за програмиране, с теоретични алгоритмични системи
Свойства на алгоритмите:Определеност – Алгоритъмът като цяло и всяка негова стъпка при едни и същи данни дават един и същ резултат при различни изпълнения. Крайност- Алгоритъмът и всяка негова стъпка се изпълняват за крайно време. Алгоритъмът съдържа краен брой стъпки и се изпълнява краен брой пъти. Дискретност – Изпълнението на алгоритъма във времето се извършва на интервали, на стъпки – процесът на изпълнение на алгоритъма е дискретен, не е непрекъснат. Всеки алгоритъм има множество от входни данни – това е дефиниционната област за която алгоритъмът работи коректно. Всеки алгоритъм при изпълнението си дава резултат – празно множество резултати не е допустимо. Коректността на алгоритмите се проверява по 2 начина: Експериментален - Използват се тестови набори от данни и известни резултати. При сегашните огромни размери на програмите 100% коректност няма. Аналитичен - на входа се задават входни условия, на изхода изходни и внякои вътрешни точки – междинни условия
Видове алгоритми: Според структурата: Неразклонени (линейни), Разклонени (нелинейни) – делят се на циклични и нециклични
Според броя на иструкциите: последователни, паралелни
Според спазването на условието за детерминираност – Детерминирани и недетерминирани.
Според оптималността на резултата-точни и евристични(приблизителни)Точният алгоритъм дава винаги оптимален резултат. Евристичните алгоритми за дадени входни данни дават резултат близък до оптималният или пък въобще не дава резултат.

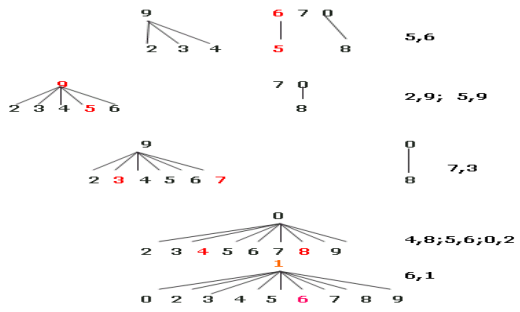
2.Примерна задача –свързаност на обекти.Дефиниране на абстрактни операции в задачата.Начален алгоритъм.Алгоритъм за бързо намиране, програмна реализация. Представяне в дърво.

Дадена е послед.от цели числа, всяко число представлява някакъв обект. Записът p – q означава p е свързано с q. Връзката е тран-зитивна,т.е ако p - q и q – r, то p - r. Иска се да премахнем ненужните двойки от множ., т.е.когато в прог.се въведе двойка p – q, тя трябва да я изведе само ако според досега въведените двойки p не е свърз. с q. Зад. е да се направи прог.която помни дост. инф. за постъпилите двойки до момента и може да прецени дали има нова връзка м/у обекти.В зад.се иска да знае дали опр.двойка е свързана,а не да покаже 1 или всички пътища, които я свързват.Намира важни прил.напр. връзки в компютърна,ел.мрежа и др. От теорията на графите => N обекта са свързани ⇔,когато има точно N – 1 изведени двойки в алг.на свързаност.
Св-во. Алг.с бързо намиране изпълнява поне MN инструкции,за да реши зад за свърз с N обекта,която вкл. M обединения
Намиране и Обединение: След като прочетем нова двойка p – q от входа,изпълн.опер.намиране за всеки член на двойката. Ако те са в едно множество, минаваме на следв.двойка. Ако не са, правим опер.обединение и извежд. двойката
Начален алг. Той запомня всички двойки и ги обхожда,за да проверим дали следва-щата дв.обекти е свърз(мн. ресурсоемен).
Алгоритъм за бързо намиране(БН)
Осн. на този алг е масив с цели числа с св- вото,че p и q са свърз ⇔,когато p и q ел. в масива са ≡.За да изпълним обедин p и q минаваме през масива като променяме вси-чки клетки с същата ст-ст като p да имат същата ст-ст като q.

P	q	0 1 2 3 4 5 6 7 8 9
3	4	0 1 2 4 4 5 6 7 8 9
4	9	0 1 2 9 9 5 6 7 8 9
8	0	0 1 2 9 9 5 6 7 0 9
2	3	0 1 9 9 9 5 6 7 0 9
5	6	0 1 9 9 9 6 6 7 0 9
2	9	0 1 9 9 9 6 6 7 0 9
5	9	0 1 9 9 9 9 9 7 0 9
7	3	0 1 9 9 9 9 9 9 0 9
4	8	0 1 0 0 0 0 0 0 0 0
0	2	0 1 0 0 0 0 0 0 0 0
5	6	0 1 0 0 0 0 0 0 0 0
6	1	1 1 1 1 1 1 1 1 1 1

Представ. в дърво.





Програмна реализация

```
#include <stdio.h>
#define N  10 000
main ()
{ int I, p, q , t, id[N];
  for( I = 0; I < N; I++) id[I] = I;
while(scanf("%d, %d \n", &p,&q) == край
{
if( id[p] == id[ q ] )continue;
for( t = id[p], I = 0; I<N; I++)
if(id[I] == t) id[I] = id[q];
printf("нова връзка")
 }}
```

3.Свързаност на обекти – алгоритъм с бързо обединение.Програмна реализация.Анализ и сравнение с алгоритъма за бързо намиране. Представяне в дърво.

Този алг. е базиран на структура данни: масив индекси по имена на обекти. Всеки обект сочи към друг обект от множеството в структура без цикли.За да определим дали 2 обекта са в едно и също множ. следваме указатели за всеки от тях, докато стигнем до обект,който сочи себе си.Ако не са в едно и също множ.,стигаме до разл.обекти, които сочат себе си. За да получим обеди-нение просто свърз.единия към другия.

Програмна реализ. за всяко p и q:

```
for ( i = p;  i != id[I];  i = id[ i ]);
for ( j = q; j != id[ j ];  j = id [ j ]);
if( i == j )  continue;  // двата указат. са //еднакви  => свързани числа
id[I ] = j;  // обединяваме
printf( " %d  %d \n", p, q);
```

При обработване двойката p - q ние:

1.следваме указатели от p докато id[I]= = I, 2.следваме указатели от q докато id[j]= = j. , 3.ако I!= j => id[I] = id[j] => обединението е просто прилепяне към дърво

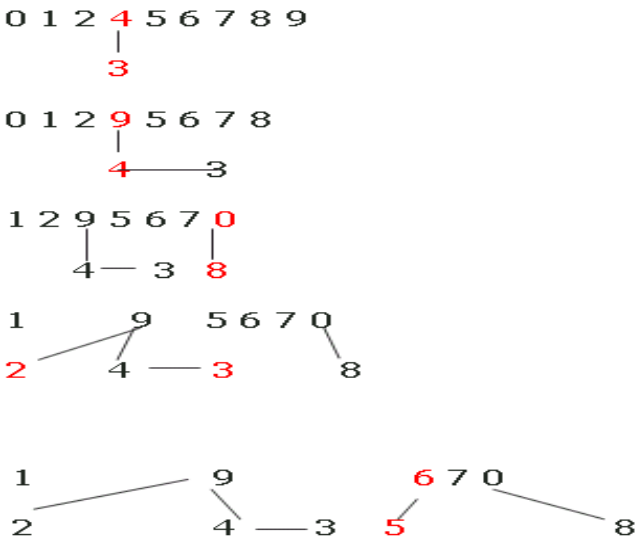
Анализ и сравнение

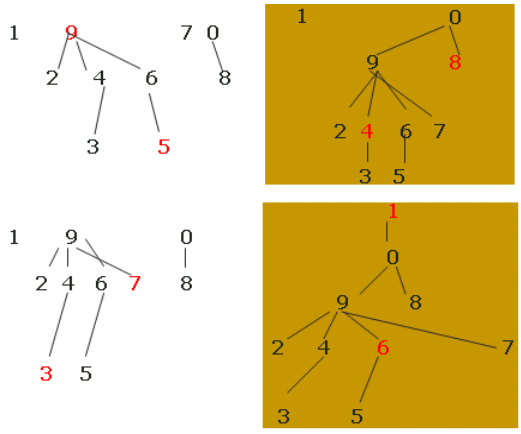
Алг.за БО е по бърз от този за БН,защото той не трябва да минава през целия масив за вся-ка входна двойка. Изпълн.на алг.за БО за-виси в гол.степен от х-ра на входа.Разлика-та м/у бързо обед. и намир. представлява подобрение, но недостатъка е че не може да се гарантира че подобрението в бързи-ната е значително защото вх. данни може да са такива че да направят намир. бавно.

Св-во. За M>N,алг.с БО може да отнеме повече от MN/2 инструкции, за да реши зад. за свързаност с M дв.от N обекта.

Док-во.Ако на вх. двойките идват подреде-ни така: 1-2, 2-3 и т.н След N-1 подобни дв. имаме N обекта в 1 и също множ,а дървото което се получило е права линия. Като N сочи към N -1, N – 1 сочи към N – 2 и т.н. За да изпълни опер.намиране за обекта N прог.трд мине през N-1 указателя. Ср. брой указатели е (N-1) /2. Ако останлите двойки свързват N с някакъв друг обект,опер.на-миране за всяка от тези дв. вкл. поне N – 1 указателя. Общата сума за M-те опер нами-ране за тази последов.от вх. дв е > MN/2

Представ в дърво.





4.Свързаност на обекти – алгоритъм с претеглено Бързо обединение.Представяне.

За избягване на лоши сл. при алг. за БО е създаден алг.за ПБО. Вместо случайно да се свързва II дърво към I при опер.обединение,се следят броя на върховете в всяко дърво и винаги свързваме по-малкото дърво към по-голямото. Промяната изисква малко повече код и още 1 масив, в който се държат броя на върховете, но ефективноста съществено се подобрява.

Св-во. Алг. проследява най-много 2 lgN указателя за да определи дали 2 измежду N обекта са свързани.

За големи числа -почти линеен алг. възможни подобрения:

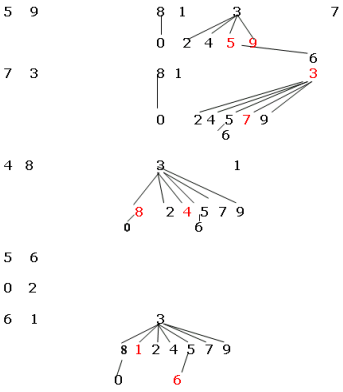
1.сплескване на дървото като всеки връх сочи директно корена(чрез смени на указатели)

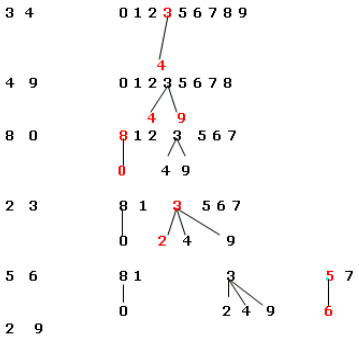
2. всяко ребро се разглежда 1 път - не е нужно да се помни => спестяване на памет.

Реализация:

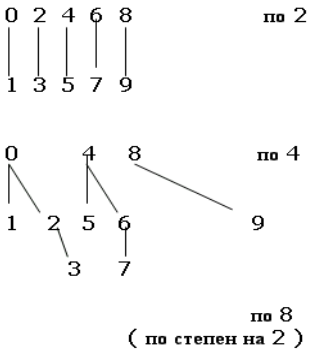
```
#include <stdio.h>
#define N  10 000
main()
{ int I, j, p, q, id[N], sz[N];
for(I=0;I<N; I++)
{id[I] = I; sz[ I] = 1;}
while(scanf("%d %d\n",&p,&q) == край)
 {for( I = p; I != id[ I ]; I = id[ I ]);
  for( j = q;  j != id[j ];  j = id[ j ]);
  if (I == j ) continue;
  if( sz [ I ]  < sz [ j ])
    { id[ I ] = j;  sz[ j ] += sz[ I ];}
  else  { id[ j ] = I;  sz [ I ] += sz[ j ];}
printf ( % d  %d \n", p, q);}}
```

част II на дървото.





най-тежък случай: еднакви дървета за обединение



5.Математически основи в анализа на алгоритми. Използвани формули с експоненти, логаритми, редиц и.

Основните причини,за да извършваме матем.анализ на алг. са:
-за да сравним различни алг.за една и съща зад.
-за да предвиждаме производителността в нова среда
-за да задаваме ст-сти на параметрите на алг.
Алг.обикновено имат време за работа, пропорц. на една от следните ф-ии*(N-главен параметър на нарастване):*
1-инструкцията се изпълнява само веднъж или най-много няколко пъти, казваме че времето за работа на алг. е *константа*
log N-ако времето за работа на алг. е *логаритмично*,програмата става по-бавна с нарастването на N.Среща се при прог, които решават големи задачи.
N-времето е *линейно*,когато всеки вх. елемент се обработва по малко.Както се променя N,така се променя и вр. за работа
N logN-такова време за работа се по-лучава,когато алг.решават зад.,като я разделят на по-малки подзадачи,решават ги независимо и после обединяват решението
N^2-когато времето е *квадратично*,алг.м д се използва практически само за малки зад. Появява се в алг,в които се обработват всички двойки от данни.
N^3-ако алг.обработва тройка от данни,той има *кубично* вр.за работа,за малки задачи
2^N-вр.за работа нараства *експоненциално*, тези алг.намират малко прилож в практиката.
Осн.формули,които се използ при работа с експоненти, логаритми и редици са:
1)експоненти-$X^AX^B=X^{A+B}$ $X^A/X^B=X^{A-B}$;
$X^N+X^N=2(X^N)$; $2^N+2^N=2^{N+1}$
2)логаритми- $X^A=B$, само ако $\log_x B=A$
--T1: $\log_a B=\log_c B/\log_c A$,при A,B,C>0,A!=1
--T2)logAB=logA+logB; за A,B>0
$x^*2^y=AB=2^z$ ->X+Y=Z.
3)редици-$\Sigma^N_{i=0}\,2^i=2^{N+1}-1$;
$\Sigma^N_{i=0}A^i=(A^{N+1}-1)/(A-1)$ при 0<A<1 то $\Sigma^N_{i=0}A^i<=1/(1-A)$; при N->безкр.,сумата клони към 1/(1-A)
-аритм.редици $\Sigma^N_{i=1}I=(N(N+1))/2\approx N^2/2$
и наистина 2+5+8+..(3к-1),което е 3(1+2+..+к)-(1+1+..+1),което е предст.и ка-то(к(3к+1))/2 => ((3к+1)к)/2=3(Х)-к =>
Х=(N(N+1))/2 ,което е $\approx N^2/2$.
$\Sigma^N_{i=1}I^2=(N(N+1)(2N+1))/6\approx N^3/3$
Матем.озн,които се срещат във формулите, описващи поведението на алг са:
$\lfloor x \rfloor$ - закръгляне отдолу
$\lceil x \rceil$ -закръгляне отгоре
lgN-двоичен логаритъм
lnN-натурален логаритъм
N!-функция факториел
F_N—числа на Фибоначи

0 1 1 2 3 5 8 13 21 34 55 ...
дефинирани по формулата
$F_N=F_{N-1}+F_{N-2}$ за N>=2,$F_0=0$,$F_1=1$
H_N—хармонични числа
$H_N=1+1/2+1/3+...+1/N$.

6.Въведение в рекурсията.Основни свойства на рекурсията.Типове рекурсии, аналлиз на производителността и доказателства.Формални техники за преобразуване на рекурсивни в нерекурсивни алгоритми;опашна рекурсия,множествена рекурсия.

Рекурсивен алг.е този,който при изпълн.си се обръща към себе си пряко или косвено поне 1 път.За да реализираме рекурс.алг. използ.*рекурс.ф-ии*.
Има 2 вида рекурсия:
- **проста** – в тялото си алг.се обръща поне 1 път към себе си пряко,но с др.аргументи, които постеп.намал.и се стига до кр.ст-сти, при които алг.не се обръща към себе си
- взаимна – алг.А1 се обръща на 1 или по-вече места към А2 и обратно. Мд има м/у повече от 2 алг.
Класич.пр.за рекурс.ф-я е *факториел.*
N!=N(N-1)! , N>=1 с 0!=1.

```
Int factorial(int N)
{ if (N==0) return 1;
  return N*factorial(N-1); }
```

Рекурс.ф-ии дават сигурност,компактна реализ.Тези ф-ии трябва да удовл.=>главните св-ва:
-да реши изрично осн.сл.(усл.за край)
-всяко рекурс.викане трябва да вкл.по-малка стойност на аргументите. Пример:

```
int F (int i)
{ if (i<1) return 0;
  if (i==1) return 1;
  return F(i-1) + F(i-2)
```

Рекурс.реализ.на тази зад.е изкл.неефективна.Алг.е с експоненц.време(изкл.мн.повт)
Типове рекурсия и оценки:
А) премахва 1 елемент от N входящи числа при циклично прохождане
$C_n=C_{n-1}+N$; $C_1=1$
Док: $C_n=C_{n+1}+N=C_{n-2}+(N-1)+N=$
C_1.. +2+..N=1+2+…+(N-2)+(N-1)+N=
=[N(N+1)]/2.=>$C_n\approx N^2/2$
Б)рекурсии с разполовяване на входния поток на всяка стъпка
$C_n=2C_{n/2}+1$; $C_1=1$;
Док:некаN=2^n(пр.битове за предст.на N)=>
$C_2{}^n=C_2{}^{n-1}+1=C_2{}^{n-2}+1+1=C_2{}^{n-3}+3=$
…=$C_2{}^0+n=n+1$ =>
оценката на C_N зависи от броя битове, необходими за представяне на N-> lgN
В)рекурсивни програми,разполовяващи входа,но преглеждащи всеки елемент
$C_n=C_{n/2}+N$; $C_1=0$;
Док: разписва се като N+N/2+N/4+N/8+… =>Оценката е строго пропорц.на вх.->2N
Г)рекурсии с лин.обхождане на входа преди,по време или след разполов.на входа
$C_n=2C_{n/2}+N$;$C_1=0$(алг.“разделяй и владей”)
Док: $C_2{}^n=2C_2{}^{n-1}+2^n$; делим на2

->: $C_2^n/2^n=[C_2^{n-1}/2^{n-1}]+1=[2C_2^{n-2}+2^{n-1}]/2^{n-1}+1 =C_2^{n-2}/2^{n-2}+2=...=n ->C_N=NlgN$

7.Оптимизации при взаимна рекурсия + малко от горния въпрос

Рекурсивен алгоритъм е този, който решава проблем чрез решаването на един или по-малки екземпляри на същия проблем.За да реализираме рекурсивни алг, използваме *рекурсивни функции*-рекурсивна функция е тази,която вика себе си.Рекурентни връзки са рекурсивно дефинирани функции. Една рекурентна връзка дефинира функция,чиято деф. област е неотрицателните цели числа или чрез някакви начални стойнос-ти,или (рекурентно) от гледна точка на нейни собствени стойности върху по-малки цели числа.Може би най-известната такава функция е функцията *факториел,*която се декларира чрез рекурентната връзка N!=N(N-1)! , за N>=1, 0!=1.

Ние използваме рекурсия,защото тя често ни помага да изразим сложни алг в компактна форма,без да жертваме ефикасност.Но както се оказва много лесно е да напишем проста рекурсивна фунция,която е изключително неефективна и е необх да упражняваме грижата да избягваме да бъдем натоварени с труднообработваеми реализации.Ето пример за една рекурсия:

```
 Int f(int x)
 { if (x==0) return 0;
     else return (2*f(x-1)+x*x);
}
```

т.е. $f(0)=0; f(x)=2f(x-1)+x^2$;

За да използваме рекурсия ние трябва да се съобразяваме с някои правила,напр. За предпочитане е реализация с 1 for(ако е възможно), и никога в 1 стъпка повече от 1 рекурсия,защото тогава възниква опасност от т.нар. *множествена рекурсия(като при числата на Фибоначи),* f(N)=f(N-1)f(N-2);

пр:

```
 voidDoubleCountDown (int N)
     { if(N<=0);
         {DoubleCountDown(N-1);
           DoubleCountDown(N-1)}
```

/*времето се удвоява ???

нека имаме означението T(N) и T(0)=1; времето на изпълнение се подчинява на формулата:1+2T(N-1)

N	0	1	2	3	4	10
T(N)	1	3	7	15	31	2047

виждаме $T(N) = 2^{N+1} – 1->O(2^N)$

Може също да се получи и индиректна рекурсия:пр. *Криви на Шерпшински* $O(4^N)$. При това се получава разход на памет:

```
  Int BadForMemory()
  { int *x;
    GetMemory(x,10 000);
     BadForMemory(); }
```

Съществуват техники на формално преобразуване на рекурс в нерекурс алг.

1)**Премахване на опашна рекурсия**

```
 procedure Recurse(A: Integer);
    begin
      //извършва нещо
       Recurse(B)
     end;
procedure NonRecurse(A: Integer)
    begin
      while(not done)  do
         begin
          //извършва нещо
          A:=B;
          end;
       end ;
```

2)Запаметяване на междинни резултати :

пр.: Fibonacci числа: за Fib(29)-> Fib(1) и Fib(0) се изчисляват 832 040 пъти??? Междинните резултати се съхраняват в таблица след първото изчисляване.

3)Подменяне посоката top-down към botton-up(пр. с Fibonacci числа)

При тази техника оценката за алг.на Fibonacci е O(N) за Fib(N) вместо O(Fib(N))

Напр. за Pentium 166 MHz и Fib(40)----155 sec с рекурсивен алгоритъм,докато с тази модификация Fib(1476) се смята за <sec.

4)Подмяна на алг.чрез преструктуриране на кода:оформяне процеса за съхраняване на информацията,стъпка и възстановяване:

пр.:

```
procedure Recurse(num:Integer);
      begin
         <block 1 of code>
          Recurse(<parameters>)
          <block 2 of code>
        end ;
```

преструктурираме:

```
procedure Recurse(num:Integer);
     begin
      <block 1 of code>
       Recurse(<parameters>)
       <block 2 of code>
      end ;
```

и

```
procedure Recurse(num:Integer);
   var    pc:Integer;
   begin
     pc:=1;
     repeat
     case pc of
     begin <code block 1>
     if(базов случай) then pc:=2 //прескачаме еркурсията
      else begin //съхраняваме пром. за следрекурсивното викане и
pc=2;устано-вяваме пром. нужни при рекурсия (напр. n:=n-1) и
преход към начало на рекурсията:
       pc:=1; end ;    end;
     begin
      <code of block 2>
      pc:=0;  end;
      //код след рекурсията
```

5)алг.промяна при сложни случаи на взаимна рекурсия:пр. криви на *Шерпински:*

при този случай се оформят отделни процедури с взаимна рекурсия:

SierpA,SierpB,SierpC and SierpD.

Всяка от тях вика останалите,които от своя страна я викат рекурсивно.Всяка служи за изчертаване на горна,лява,долна и дясна част от общата картина .

При рекурсивен вариант на всяка процедура оценката на алгоритмичната сложност е $O(4^N)$.

Нерекурсивният вариант на същата прог. е с много по-добра оценка-$O(N^4)$,допустима е голяма дълбочина на вложеност,но разбираемостта на кода е по-лоша.

8.Анализ на алгоритми. Математически обозначения,дефиниции и правила в анализа.

Анализът на алг.е ключът към пълноценно-то им разбиране,за да мд ги прилагаме ефективно към практически зад.Той играе роля във всяка стъпка от процеса на конструиране и писане на алг.Една от първите стъпки,за да разберем поведението на алг е да направим *емпиричен* анализ.Напр.ако имаме два алг за решаване на една и съща задача,изпълняваме ги и двата,за да видим кой работи по-бавно.Когато обаче емпирич изследвания почнат да отнемат повече време,се нуждаем от матем.анализ.Осн.причини,за да извърш. матем анализ на алг. са

-за да сравн разл.алг. за една и съща задача, -за да предвижд.произвстта в нова среда, -за да задаваме ст-сти на парам на алг

Това прави възможно да се предвиди точно колко дълго ще работи дадена програма, или дали дадена програма ще работи по-добре от друга при опр обстоятелства. Матем.означения,които се срещат във формулите,описващи поведението на алг са:

$\lfloor x \rfloor$ - закръгляне отдолу

$\lceil x \rceil$ -закръгляне отгоре

lgN-двоичен лагаритъм

lnN-натурален логаритъм

N!-функция факториел

F_N—числа на Фибоначи

0 1 1 2 3 5 8 13 21 34 55 ...

дефинирани по формулата

$F_N=F_{N-1}+F_{N-2}$ за $N>=2, F_0=0, F_1=1$

H_N—хармонични числа

$H_N=1+1/2+1/3+...+1/N$.

Дефиниции:

*T(N)=O(f(N)) ако съществуват c,n и за всяко $N>=N_0$ ->T(N)<c

цели:пренебрегване малки членове,облекчен анализ,оценка по горна граница. Пр.:$N(N-1)/2 \to N^2/2$ или N=O(N) или $2a_0NH_N+a_1N+a_2 \to 2a_0NH_N+O(N)$ (за големи N)

- T(N)= Ω (g(N)) ако съществуват const c и n_0, такива че T(N) >+ cg(N) за $N>n_0$

*T(N) = O (h(N)) ако и само ако T(N) = O(h(N)) и T(N) = Ω(h(N)) казваме :

"grouth rate" на T(N) = "grouth rate " h(N)

ефект от удвояване на голем.на зад в/у t:

1 – никакъв

2^N- квадратичен

lgN - лек

N^2 - с коефициент 4

N - удвоява

N^3 - с коеф. 8

NlgN - малко > от двойно

пр: $(N + O(1))\, N + O(\log N) + O(1)) = N + O(N) + O(N\log(N)) + O(\log N) + O(N) + O(1) = N^2 + O(N\log N) \approx N^2$ за големи N.

Базовите правила в анализа на алг. са:

▪правило 1: ако T1(N) = O(f(N)) и T2(N) = O(g(N)), то

a)T1(N) + T2(N) = max (O(f(N)), O(g(N)).

b)T1(N)*T2(N)= O(f(N) * g(N)).

▪правило 2: ако T(N) е полином със степен k, то $T(N) = O(N^k)$

напр: вместо $T(N) =O(2N^2) \to T(N)=O(N^2)$

правило 3 : $\log_k N = O(N)$ за всяко k (логаритмите растат бавно)

9.Задача за намиране на максимум на подниз.решения.Анализ.

1. преглед на всички възможности

```
int maxSubSum1(const vector<int>  &a)
{ int maxSum = 0;
for( int I = 0; I < a.size(); I++)
  for( int j = I; j < a.size(); j++)
 { int thisSum = 0; //O(1*N*N*N) = O(N^3)
  for( int k = I;k < j;k++) //N-1  N-1  N-1
       thisSum += a[k];              *1
  if( thisSum > maxSum) //i=0   j=0   k=i
       maxSum =  thisSum;          }
return maxSum;}
```

3. divide& conquer стратегия:

цепим проблема на 2 части и т. н. рекурсивно. Съединяваме двете решения.

Max сума може да е на 3 места:

I пол.от числа II пол.от числа

4 - 3 5 - 2 -1 2 -2

6 8

11

```
int maxSumRec
( const vector< int> &a, int left, int right)
{ if( left == right)        // базов случай
    if( a[left]) > 0)   return a[left];
    else             return 0;

int center = (left + right )  / 2;
int maxLeftSum =
maxSumRec( a, left, center);       // T(N/2)
int maxRightSum =
maxSumRec( a, center + 1, right);  // T(N/2)

int maxLeftBorderSum=0; leftBorderSum = 0;
for( int I = center; I > = left; I--)
  { leftBorderSum += a[I];
if(leftBorderSum > maxLeftBorderSum) maxLeftBorderSum =
leftBorderSum }

int maxRightBorderSum = 0;
rightBorderSum = 0;  // O(N)
for( int j = center + 1; j < right; j++)
  {  rightBorderSum  +=a[j];
   if(rightBorderSum > maxRightBorderSum)
```

2. премахваме третия for:

```
int maxSubSum2( const vector <int> & a)
 { int maxSum = 0;
  for (int I = 0; I < a.size(); I++)// натрупваме
{int thisSum = 0;// сумата като сме я нули-   //рали в началото на  всеки
външен for: т.е for( int j = I; j < a.size(); j++)        j      i-1
       { thisSum += a[j];
        if( thisSum > maxSum) //Ак=Aj +Ak                                  k=I
k=i
        maxSum = thisSum;}}
return maxSum; }       // O(N^2)

   maxRightBorderSum = rightBorderSum; }
return max3(maxLeftSum, maxRightSum,
MaxLeftBorderSum+maxRightBorderSum);}

int maxSubSum3 (const vector<int> &a)
{ return maxSumRec( a,0, a.size() – 1); }
```

анализ (както Фибоначи):

нека T(N) за N числа; за N = 1 -> T(1) = 1;

при N>1 имаме 2рекурсии.Всеки for->O(N)

=> T(N) = 2 T(N/2)+O(N) 2T(N/2) + N

ако N=2^к T(N)=N*(k+1) T(2) = 4 = 2*2

=NlogN=O(NlogN) T(4) = 12 = 4*3

T(8) = 32 = 8*4

T(16)=80=16*5

ако N != 2 ^к -анализът е по-тежък,но резултатът е същия.

4. линейно време

```
int maxSubSum4 ( const vector <int> &a)
 {int maxSum = 0; thisSum = 0;
    for( int j = 0; j < a.size(); j++)
    { thisSum += a[j];
     if( thisSum > maxSum)
              maxSum = thisSum;
     else  if( thisSum < 0)
     thisSum = 0; }
return maxSum; }
```

Отриц. ч-ло или сума едва ли е част от търсена подредица. Мд скочим към i+1 и дори към j+1. Запазили сме докъде е стигнало j.

а)всеки момент алгоритъмът дава реш. до което е стигнал (on-line алгоритми)

б)не изисква много памет.

10.Логаритми в анализа. Бинарно търсене, Евклидов алгоритъм за НОД, повдигане на степен. Анализ.

Един алг.е O(logN) ако изисква const време O(1) за разделяне задачата на половинки, както и за обработката на частите е необх const време – O(N).
Методът разделяй и владей се базира на идеята,че един обект се разделя на части и рекурсивно се намира реш.за всяка част и оттук за целия обект.
Друг пример за такъв алг.двоично търсене:
Търсим X в сорт.редица A0, A1, …An-1.
Сравн.X със средния елем.на сорт.редица и ако са равни,това е X,ако X<Acp,прилагаме метода за лява пол.,ако X>Acp – за дясната

```
int low = 0, high = a.size() – 1;
while( low <= high)
  {int mid = ( low + high) / 2;
   if( a[mid] < x)  low = mid + 1;
   else if(  x < a[ mid])  high = mid – 1;
        else  return mid;            //открито
} return  NOT_FOUND; }
```

O(1) вътре във всеки цикъл.
Циклите са:log(N – 1) + 2. => O(log(N))

Евклидов алг. за НОД(M, N) M >= N

Базиран е на наблюдението, че НОД на 2 цели числа X и У,X>У е същият,като на У и X **mod У(остатъкът при дел на X с У)**

```
long gcd(long m, long n)
{ while ( n != 0)
{long rem = m % n;
  m = n; n = rem; }
return m;}
```

Доказва се че след 2 итерации остатъкът е поне половината. => 2 logN = O(log(N))

Повдигане в степен

```
long pow(long x, int n)
{ if(  n == 0) return 1;
if( n == 1)  return x;// излишни
if (isEven(n))
   return pow ( x*x, n/2);// брой умнож при
                          // нечетно <= 2 logN
else return pow( x*x, n/2) *x;}
```

може: return pow (x,n-1)*x; Тогава O(logN)

11.Рекурсия и дървета. Рекурсивни алгоритми

Рекурсията е фунд.понятие за информ. Тя е тясно свърз.с дърветата. Използваме дървета за да разберем рекурс.прог. Рекурсивен алг.е този,който при изпълн.си се обръща към себе си пряко или косвено поне 1 път. За да реализ.рекурс.алг. се използ.*рекурс. ф-ии*. Има 2 вида рекурсия:
- **проста** – в тялото си алг.се обръща поне 1 път към себе си пряко,но с др.аргументи, които постеп.намал.и се стига до кр.ст-сти, при които алг.не се обръща към себе си
- **взаимна** – алг.A1 се обръща на 1 или по-вече места към A2 и обратно. Мд има м/у повече от 2 алг.

Класич.пр.за рекурс.ф-я е *факториел,* която се декларира чрез рекурентната връзка
N!=N(N-1)! , N>=1 с 0!=1.Тя може да се представи със следния код:

```
Int  factorial(int N)
 {   if (N==0) return 1;
     return N*factorial(N-1); }
```

Ф-ята връща коректна ст-ст,когато се вика с ст-сти на N,които са дост.малки,че N! да мд се представи като int. Тази ф-я е еквивалентно на прост цикъл for със същото предназначение **for (t=1; i=1; i<=N; i++) t* = I**; Винаги е възможно да транформ.рекурс. прог.в нерекурс.,изпълн.същите изчисл.
Използваме рекурсия,защото тя ни помага да изразим сложни алго-ритми в компактна форма. Например при използване на ф-ята факториел избягваме използ.на локални променливи. В др.случаи използ.на рекурс. води до създ.на неефективни алг.
Тези ф-ии трд удовл.=> главни свойства:
-да решават изрично основния случай
-всяко рекурсивно викане трд вкл. по-малка ст-ст на аргументите.
Дълбочина на рекурсията е степента на влагане на виканията по време на изчисл.В общия сл.дълбочината ще зависи от входа. Когато реализ.реална рекурс.прог.трд вземем предвид, че средата за прог.трд поддържа стек с размер дълбочината на рекурс. За огромните зад.,необх.за този стек прост-ранство мд ни откаже от използ.на рекурс.

12.Подходът: Разделяй и владей.Свойства, известни алгоритми, реализация и оценъчна формула.

Идеята на този подход е обектът(входните данни)да се раздели на части и рекурсивно се намира реш.на всяка част и оттук за це-лия обект.
Пр.намиране на макс.от N елем, съхранени в масив:a[0],a[1],…,a[N-1]
Нерекурс.реш:

```
for ( t = a[0]; i = 1; i < N; i++)
 if ( a[i] > t )  t = a[i]
```

Рекурс.реш. се изр.в следното:поредицата от елем. a[l],…,a[r] се разделя на 2 пореди-ци: a[l],…,a[m] и a[m+1],…,a[r],намират се max елем.в двете части(рекурсивно) и се връща по-гол.от тях като max от редицата

```
Item max (Item a[ ], int l, int r)
 { Item u,v;  int m = (l+r) / 2;
   if (l == r) return a[l];
   u = max (a, l, m);
   v = max (a, m+1, r);
   if (u > r) return u;
   else return v; }
```

Използ.на метода се дължи на факта,че той дава по-бързи реш.отколкото тези,получе-ни с прости итеративни алг.
Св-во. Рекурс.ф-я,разделяща зад.с размер N на 2 незав,непразни части,които тя решава рекурс.,се обръща към себе си<N пъти.
Тази прог.извърш.конст,количество работа за всяко викане=>общото време е линейно.
Друг клас.пример е зад.за Ханойските кули
Дадени са 3 пръчки и N диска с дупки в ср. които мд се надяват на пръчките,дисковете са разл.по размер и първоначално са подре-дени на най-лявата от пръчките в ред най-големия диск(N)долу,най-малкия(1) горе. Зад.е да се преместят дисковете на следв. отдясно пръчка по => правила:
1)само 1 диск се премества в даден момент
2)в/у по-малък диск не мд се пост.по-голям
//Ние премест.кулата от дискове надясно чрез рекурсивно премест.наляво на всички с изкл.на най-долния,после премест.най-долния надясно,след което рекурсивно ме-стим кулата обратно в/у най-долния диск.

```
void hanoi ( int N, int d)
            { if ( N == 0) return;
   hanoi ( N-1, -d);
   shift (N, d);
   hanoi ( N-1, -d); }  //
```

Тази прог.дава рекурс.реш.на зад.Тя опр. кой диск трдб преместен при всяка стъпка и в коя посока (+ значи премести 1 пръчка надясно,прескачайки циклично към най-лявата,тръгвайки от най-дясната, а – значи 1 пръчка наляво,прескачайки циклично към най-дясната,тръгвайки от най-лявата)
Св-во. Рекурс.алг.разделяй и владей за зад. с Ханойските кули дават реш.с $2^N - 1$ мест.
За реш.на Ханойските кули имаме реш.ли-нейно по време според размера на изхода. За Ханойските кули обикн.разгл.реш.като екпоненц.спрямо времето,защото разглежд размера на зад.от гл.т.на броя на дисковете.

13.Дървета.Основни понятия и класификации.Дефиниции и свойства.

Дърветата са математически абстракции, които играят важна роля при конструирането и анализа на алгоритми. От една страна използваме дърветата, за да изучим свойствата на алгоритмите, а от друга ги използваме като ясно формулирани структури от данни.
Def. Дърво е не празна съвкупност от върхове и ребра.
Връх (възел) е прост обект, които може да има име и да носи друга асоциативна информация.
Ребро връзка между два върха.
Път е списък от различни върхове, в които посл. върхове са свързани с ребра в дървото.
Дефиниращо свойство за дърво е, че има само един път свързващ всеки два възела.
Възлите директно под даден възел се наричат негови **деца.**
Възлите без деца се наричат **листа.**
Корен на дървото е възел, от които произлизат всички останали възли.
Видове дървета:
Бинарни дървета представлява наредено дърво, състоящо се от два типа възли външни, които нямат деца и вътрешни, които имат точно две деца.
Наредени дървета- дърво с корен, в които е определен реда на децата за всеки възел.
Дърво с корен е това при което един възел е определен за корен на дървото.
Бинарно дърво е или външен възел или вътрешен възел, свързан към двойка бинарни дървета, която се нарича ляво или дясно дърво на този възел.
Математически свойства на бинарните дървета
1св-во: Бинарно дърво с n на брой вътрешни възли има n+1 на брой външни възела.
2св-во: Бинарно дърво с n на брой вътрешни възли има 2 n на брой връзки. (n-1 на брой връзки към вътрешни възли и n+1 на брой връзки към вътрешни възли.
3св-во: Нивото на даден възел в дърво е по голямо с едно от нивото на неговия родител.
4св-во: Височината на бинарно дърво е максимума от нивата на неговите възли.
5св-во: Дължината на пътищата в бинарно дърво е сумата от нивата на неговите възли.

14.Математически свойства на двоичните дървета.

Св-во. Бинарно дърво с N вътр.възела има N+1 външни възела
Док.се по индукция.Бин.дърво без вътр.възли има 1 външ.възел=>изпълн.за N=0.
За N>0,всяко бин.дърво с N вътр.възли има К вътр.възли в лявото си поддърво и N-K-1
вътр.възли в дясното си поддърво,за К м/у 0 и N-1,тъй като коренът е вътр.възел.Спо-ред индукц.хипотеза,лявото поддърво има К+1 външ.възли,а дясното->N-K=>сум.N+1
Св-во. Бинарно дърво с N вътр.възела има 2N връзки;N-1 връзки към вътр.възли и N+1 връзки към външ.възли..
Във вс.дърво с корен вс.възел освен корена има 1родител,а вс.ребро свързва възел с неговия родител,така че има N-1 ребра,свърз.
вътр.възли.Подобно вс.от N+1 външни възела има по 1 ребро към своя родител.
Деф.Нивото на възел в дърво е > от нивото на неговия родител.
Височина на дърво е max от нивата на въз.
Дължина на вътр.пътища в бинар.дърво е сума от нивата на вътр.му възли
Дължина на външ.пътища в бинар.дърво е сума от нивата на външ.му възли
Тези деф.са пряко свърз.с анализа на алг.
Св-во. Дълж.на външ.пътища в вс.бинар. дърво с N вътр.възела е с 2N по-гол. от дълж.на външ.пътища-Док.се по индукция
Св-во. Височ.на бинар.дърво с N вътр.възела е >= от lgN и <= N-1
Док.Най-лошия сл.е дегенерирано дърво с 1листо,с N-1 връзки от корена до листото. Най-добрият сл.е балансирано дърво с 2^i
вътр.възли за всяко ниво i,освен за най-долното.Ако височ.е h,трд имаме:

$$2^{h-1} < N+1 <= 2^h$$

Тъй като имаме N+1 външ.възела. Височ. в най-добрия сл.ще е = lgN
Св-во. Дълж.на вътр.пътища в бинар. дър-во с N вътр.въз.е>=Nlg(N/4) и <=N(N-1)/2
Бинар.дърв.се появяват мн.често в комп. прилож.и произв-стта е best при баланс.дър

15.Обхождане на дърво и граф.

Ще разгл.алг.за най-осн.обработваща дърв. ф-я:обхождане.При бин.дърв.имаме 2 връз-ки=>3 осн.последов.,по които да посет.въз.
- преред – посещ.възела,после лявото и дясното поддърво
- поред – посещ.лявото дърво,после възела и накрая дясното поддърво
- постред – посещ.лявото и дясното поддъ-рво и после възела

Тези 3 метода са за обхожд.в дълбочина.Те мд се реализ.чрез рекурс.прог:

```
void  traverse (link h, void ( *visit)(link))
 { if( h==NULL) return;
   (*visit)(h);
   traverse (h->l, visit);
   traverse (h->r, visit); }
```

Тази рекурс.ф-я взема връзка към дърво като арг.и вика ф-ята visit с арг.всеки от въз-лите на дървото..Ако премест.викането на visit м/у рекурс.обръщ.,имаме поредно обхожд., а ако премест.след тях,имаме постред.обхож.
При използ.на нерекурс.,които използ.стек, правим опер.за вмъкване на дървото,които зав.от желаната последов.на обхожд:
- преред – вмък.в стека дясното поддърво после лявото и накрая възела
- поред – вмък.в стека дясното поддърво, после възела и накрая лявото поддърво
- постред – вмък.в стека възела,после дяс-ното поддърво и накрая лявото поддърво.

В стека не се вмък.нулевите връзки.
Друга стратегия за обхожд.е просто да посещаваме възлите отгоре надолу и отляво надясно – обхождане в широчина,защото вс.възли от дадено ниво се появяват заедно

```
void  traverse (link h, void ( *visit)(link))
 { QUEUEinit(max); QUEUEput(h);
   while ( !QUEUEempty())
    { (*visit)(h = QUEUEget());
      if (h->l != NULL) QUEUEput (h->l);
      if (h->r != NULL) QUEUEput (h->r);} }
```

При графите обхожд.се осъщ.рекурс. или се нар.още търсене в дълбочина.Това е прост рекурс.алг.Започв.от възел V,ние:
- посещаваме V;
- рекурс.посещ.вс.непосет.въз,достиж.от V

За да посетим вс.възли,свърз.към възел К в граф,ние го маркираме лкато посетен и то-гава рекурс.посещаваме вс.непосетени въз-ли от списъка на съседство на К.

```
void  traverse (int k, void ( *visit)(int))
 { link t;
   (*visit)(k); visited[k] = 1;
   for ( t = adj[k]; t != NULL; t = t -> next)
   if ( !visited[t->v])  traverse ( t ->v, visit); }
```

Разл.м/у търс.в дълбоч.и обобщ.обхожд.на дърво е,че е необх.да се предпазим изрично от посещ.на вече посетени възли.
Ако използ.опашка вместо стек,тогава имаме търс.в широч.,което е аналогично на обхождане в широч.на дърво.
Напр.за да посетим вс.възли,свърз.към възел К в 1 граф,вмък.в К в опашка FIFO,после влизаме в цикъл,където е следв.възел от опашката и ако не е посетен,го посещаваме и вмък.в опашката вс.непосетени възли от списъка му на съседство.

```
 void  traverse (int k, void ( *visit)(link))
 {link t;
  QUEUEinit(V); QUEUEput(k);
  while ( !QUEUEempty())
   if (visited[k=QUEUEget()] == 0)
   {(*visit)(k); visited[k] = 1;
     for ( t = adj[k]; t != NULL; t = t -> next)
     if (visited[t->v]==0) QUEUEput (t->v);} }
```

16.Рекурсивни алгоритми в двоични дървета.

Бин.дърв.мд се обхождат по няколко н-на:
- преред – посещ.възела,после лявото и дясното поддърво
- поред – посещ.лявото дърво,после възела и накрая дясното поддърво
- постред – посещ.лявото и дясното поддърво и после възела
-както и в широчина

Често е необх.да намерим стстите на разл. структурни парам.за 1 дърво,само по дадена връзка връзка към дървото.Следв.прог. съдържа рекурс.ф-ии за изчисл.броя на възлите и височината на дадено дърво.

```
int count (link h)
  { if ( h==NULL ) return 0;
    return count (h->l) + count (h->r) + 1;}
int height ( link h )
 { int u,v;
   if ( h == NULL ) return -1;
   u = height ( h->l );  v = height ( h->r);
   if ( u > v ) return u+1; else return v+1; }
```

Др.ф-я ни показва рекурс.алг.за обхождане на дърво е този за отпечатване на дърво. Ако отпеч.елемента преди рекурс.викане, получаваме прередно обхожд.Тази прог.допуска че елем.в възлите са символи:

```
void  printnode ( char c, int h )
 { int i;
   for ( i = 0; i < h; i++)  printf (" ");
   printf ("%c\n", c); }
void  show ( link x, int h )
 { if ( x==NULL) {printnode('*',h); return; }
   show (x -> r, h+1);
   printnode (x -> item, h);
   show (x ->l, h+1); }
```

17.Сортировки.Селективна сортировка.Сортиране чрез вмъкване.Примери.

В много сортиращи прил. е възм. метода. к-то тр. да се избер. да е прост. алгорит. Поради сл. причини: 1 – во. Често сорт. прог. се изп. само веднъж и ако елем. сорт. не е по – бавна от др. части на прогр. не е нужно да се търси по – бърз метод. Ако бр. на сорт ел. не е мн. гол.(до няк. 100) също може да се избере пост метод. 2 – ро. Елементар. сорт са по удобни за малки фаилове. Бав. методи са добри също за почти сортирани фаилове. или такива които съдърж.голям бр.повтар. се ключове.

Селективна сортировка. Този алгор. раб. по сл. начин 1-во намира най-малкият елемент от масива и го разменя с този от 1-ва позиц. После намир 2-я най-малък ел. и го разм. с този от 2 поз. Продълж. по този нач докато целия мас. се сортира. Методът се нар.селек.сорт.защ.той раб чрез повта-рящ се избор на мин.оставащ елемент.

Пример с числа :

27 32 29 30 18 28 24 30
18 32 29 30 27 28 24 30
18 24 29 40 27 28 32 30
18 24 27 30 29 28 32 30
18 24 27 28 29 30 32 30 Разменя се
18 24 27 28 29 30 32 32 сам с себ. си

18 24 27 28 29 30 30 32

```
void selection (Item a[], int l, int r)
{ int i, j;
for (i = l; i< r; i++)
{ int min = i;
for (j = i+l; j <= r; j++)
if (less(a[j], a[min])) min = j;
exch (a[i], a[min]);
}}
```

Вътр. цикъл представлява само сравнение, за да провери текущия елем. с намер. наи малък. Външният цик. разменя елем. Броят на размените е N -1. Времето на изпълн. се доминира от бр. на размен. Този брой е пропорц на N квадр.Недостатък на сорт. е че вр. за изпълн зависи съвсем малко от вече подредените елем. Методът е добър за сортир. на фаилове с огромни елем. и малки ключове.

Сортировка чрез вмъкване:

Този метод работи по сл. начин: елем. се разглеждат един по един вмъкв. всеки от тях на подходящ. място. сред вече сортир.(запазваики ги по този начин сортирани). Необходимо е да се направи място за вмъквания елем., като преместв. по – гол. елем с 1 поз надясно и после вмъкв. елем. във вакантната. поз.

Пример с числа

27 32 29 30 18 28 24 30
18 32 29 30 27 28 24 30
27 29 32 30 18 28 24 30
27 29 30 32 18 28 24 30 измесет.
18 27 29 30 32 28 23 30
18 27 28 29 30 32 24 30
18 24 27 28 29 30 32 30
18 24 27 28 29 30 30 32

Програмна реализация

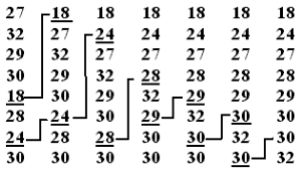
```
void insertion(Item a[], int l, int r)
{ int i;
for (i = r; i >l; i--) compexch (a[i-1], a[i]);
for (i = l+2; l<=r; i++)
{ int j = i; Item v = a[i];
while (less(v, a[j-1]))
{ a[j] = a[j-1]; j--; )
a[j]= v; }}
```

Програмата първо поставя най – малкият елем.на мас. на 1 –ва поз. , така че този елем. да служи като ограничител. Във бтр цикъл тя прави просто присвояване в място размяна. Прогр. завърщ. втр. цикъл., корато вмъкв. елем. е в същата позиция. За всяко i тя сортира елем. a[i].....a[i-1], които са по – гол. от a[i], след което поставя a[i] в подходяща позиция.

18.Сортиране по метода на мехурчето.Подобрение на алгоритъма.Примери.

Преминава се през файла като разменяме съседните елем. които не са подредени,. Деиствието продължава докато фаила не се сортира. Главн. предимство на тази сорт. е лесното и реализиране. Но тя е по – бавна от селективната сорт. и сорт. чрез вмъкване.
Сортировк се извърщва на N паса.

Пример с числа.



Програмна реализация

```
void bubble (Item a[], int l, int r)
{ int i, j;
for (i = l; i<r; i++)
   for (j = r; j>i; j--)
     comexch(a[j-1], a[j]);
}
```

За всяко i от l до r-1 вътр. цикъл намира мин елем. сред елем a[i],…, a[r] чрез преминаване през тях от дясно на ляво, сравнявайки и евентуално разменяики последоват елем. и така го поставя в поз. a[i]. Най – малкият елем. минава през всички – така тои изплува като мехурче в началото на масива.

Х-р на елементар сорт.

Селект. сорт, сорт чрез вмъкване и сорт по метода на мехурчето са алгоритми с квадратично вр. за изпълн. и за най – лошия , и за ср. сл. не изискват. допълн памет. В общия сл. времето за испълн. на алгор.за сорт е пропорц на бр. на сравн. които използва, на бр. пъти за които елем. се местят или разместват. или на двете.

Св. Селективната сорт. използва около $N^2/2$ сравнения и N размени. Времето за изпълнение на селективната сортировка при сред. сл. (O(NlogN) обновявания на min) е нечувствително спрямо входа.

Св. Сортировката чрез вмъкване използва около $N^2/4$ сравн и $N^2/4$ полуразмени (преминавания) в ср. случай и двойно повече в най – лошия.

Св. Сортиров. по Метода на мехурчето използва около $N^2/2$ сравнения и $N^2/2$ размени за средния и наи – лошия случ.

Метод. на мехурчето и сорт. чрез вмъкване работят добре за неслучайни фаилове, които често се появяват в практиката. Общоцелеви сорт. обикновено са неизползв за такива прил.

Разгл. действието на сортировк чрез вмъкване за вече сортиран фаил. Всеки елем. моментално се определя, че е на нужното място в файла и общото време за изпълнение е линейно. Същото е вярно и за сортировка по мет. на мехурчето.

Деф. Инверсия е дв. ключове, които не са в необходим ред в файла.

Св. Сорт. чрез вмъкване и по метода на мехурчето изискват линеен брои сравнения и размени за фаилове с най – мн. константен. брой инверсии, съответни на всеки елемент.

Сортировката чрез вмъкване е подходяща за почти сортиране файлове, където трябва да се добавят няколко елем. Докато метода на мехурчето и селект. сортировка не са.

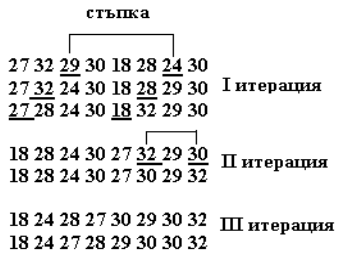
Св. Сортир. чрез вмъкване използва лин. бр. сравнения. и размени за файлове с най – многоконстант. брой. елем. имащи повече от констант. бр. съответстващи инверсии.

Св. Селективната сортировка работи за линейно време при файлове с големи елем и малки ключове.

19.Сортировка на Шел.Примери и свойства.

Сорт. на Шел е разшир на селект. сорт, к-то печели скорост, като позволява. размяна на елем, далеч един от др. Идеята е да пренаред. фаил, за да му дад св., че вземаики всеки h елем.(като започнем от някъде), пораждаме сортиран фаил. Такъв фаил се казва че е h сортиран. h сорт. фаил представлява h независ сорт фаилове. поставени последов.

Пример: В сл. стъпката е N/2 = 4; След 1 -вата. итерация пак. сорт. с стъпка 2. и после с стъпка 1.



Програмна реализация

```
void shellsort (Item a[], int l, int n)
{ int i, j, h;
for ( h =1; h<=(r-l)/9; h = 3*h+1);
for  (; h>0;h /=3)
for (i = l+h); i <=r; i++)
{ int j = i; Item v = a[i];
  while (j > = l+h) && less(v, a[j - h]))
{ a[j] = a[j-h]; j -=h;}
a[j] = v;
```

Използват се ограничители и после се заменя всяко срещане на l с h в сортиров. чрез вмъкване, прог. прави h сорт. на фаил.

Един от наи сложните въпроси при Шел сортировката е каква редица на нарастване да се използва. В п-ката обикнов.се използ. редици, които намал. геометрично

Св: Резултата от h – сортир на фаил, к-то е k - нареден, е фаил, к-то е и h и k нареден.

Св: Сортировката на Шел прави по – малк от N(h-1)(k-1) /q сравнения за g – сортиран фаил, к-то е h и k подреден, ако h и k са взаимно прости числа.

Св: Сортиров на Шел прави по – малко от $O(N^{3/2})$ сравн. за нарастването 1 4 13 40 121 364 и т.н.

Св. Сортиров. на Шел прави по – малко от O(N) на степен 4/3 срав. за нараст 1 8 23 77 281 1073 ...

Св: Сорт. на Шел прави по – малко от $O(N(\log N)^2)$
сравн. за нараст 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81.

Сортировката на Шел е много по – бърза от елементарните сортировки. Даже когато нарастванията са степени на 2., но някой редици на нарастване могат да я ускорят 5 и повече пъти.

Сортир. на Шел работи приемливо добре в/у разнообразие от типове файлове, а не само в/у файлове от случайни числа

Сорт. на Шел е избраният метод за много сортиращи приложения защото има прием-ливо време за изпълнение даже за умерено големи файлове и изисква малко кол. код, който е лесно да се пусне в действие.

20.Бързо сортиране.Стратегии за разделяне.Избор за разделящ елемент.Анализ

21.Сортиране чрез сливане.Анализ

В информатиката **сортирането чрез сливане** е алгоритъм за сортиране, базиран на сравняване, който има сложност Θ(nlogn). Алгоритъмът се гради на принципа „разделяй и владей“. Създаден е от Джон фон Нойман през 1945.

Принцип на действие.

- Несортирания списък по произволен начин се разделя на два подсписъка с приблизително обща дължина (за линейно време)
- Рекурсивно се разделят подсписъците, докато не се достигне до списъци с единична дължина
- Сливат се два подсписъка в нов сортиран списък (за линейно време)

Пример:

Процесът на съчетаване на две сортирани редици в една се нарича сливане.

Ако имаме сортираните списъци l1 и l2 и искаме да ги слеем в трети списък l, който също да бъде сортиран, можем да направим следното:

1.Докато има елементи в l1 и има елементи в l2:
2.Ако l1.pStart->data < l2.pStart->data
3.премахваме първия елемент на l1 и го поставяме в края на l
Иначе
4.премахваме първия елемент на l2 и го поставяме в края на l
5.Като излезем от горния цикъл или l1 или l2 е празен
6.Ако l1 е празен долепяме l2 към l1
7.Ако l2 е празен долепяме l1 към l2
8.Край.

За да сортираме един списък чрез сливане:
1.Ако списъкът има 0 или 1 елемент, то той е сортиран - Край
2.Иначе го разделяме на 2 части
3.Сортираме първата част - като използваме отново тази функция
4.Сортираме втората част - като използваме отново тази функция
5.Сливаме двете части в стария списък.
6.Край

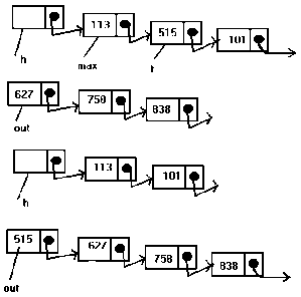
Недостатъкът на този алгоритъм е, че не сортира на място и ако сортирахме масив щеше да ни е нужна допълнителна памет за още един масив. Разпределението на паметта за новия масив е бавна операция. Това обаче не важи за свързаните списъци. Също така при тях намирането на оста нужна при бързото сортиране (quick sort) не е толкова лесно. Затова за сортиране на списъци ще използваме сортирането чрез сливане.

22.Сортиране на свързани списъци.Индексно и указателно сортиране.Примери.

Свързаните списъци са едни от осн.н-ни за структуриране на данни.След.прог.дава ин-терфейс за тип данни за свърз.списък(изпо-лзваме Item за тип данни на елем.Ф-ята init изгражда списъка,вкл.задел.на необх.па-мет.Ф-ята show отпеч.ключовете в списъ-ка.Прог.за сорт.използ.less,за да сравн.елем и манипул.указ.,за да пренарежд.елем.)

Мд се адаптира селект.сорт.или сорт.по ме-тода на мех.или вмък.за реализ.на свърз. списъци.Селект.сорт.е проста:поддържаме вход.списък(първонач.съд.данн)и изх.спи-сък(събира сорт.резулт.)и просто четем 2-я списък,за да намерим max елем.,махаме го от вход.списък и го добавяме към нач.на изх.списък.

```
typedef struct node = link;
struct node { Item item; link next; };
link NEW ( Item, link);
link init (int);
void show (link);
link sort (link);
```



1 стъпка от селект.сорт на свързан списък.

Програмна реализация

```
link listselection(link h)
{ link max, t, out = NULL;
while (h ->next !=NULL)
{ max = findmax(h);
  t = max -> next; max ->next = t->next;
  t-> next = out; out = t; }
h->next = out
return (h);
```

Поддържа се вх. списък(сочен от h->next) и изходен списък (сочен от Out).

Индексно и указателно сортиране.

Сортирането на низ от символи има голямо значение, защото те широко се използват като сортиращи ключове.Низът е указател към символ,така че сорт.ще обработи ма-сив от указ.към символи,пренареждайки ги така,че да сочат низовете в азбучен ред.Ка-то работим с низове в С,трд заделим памет за тях,статично.Пр.указателно сорт.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include “Item.h”
static char buf [100000];
static int cnt = 0;
int ITEM scan (char **x)
{ int t;
  *x = & buf [cnt];
  t = scanf (“%s”, *x); cnt+ = strlen (*x)
  return t; }
void ITEMshow (char *x)
{ printf (“%s”, x); }
```

1 прост подход за сорт.без мест.на елем.е да поддържаме индексен масив с ключове-те в елем.,достъпни само за сравн.Нека сорт.елем. са в масива data[0],…,data[N-1] и не искаме да ги местим.За да постигнем ефекта на сорт.,използ.втори масив а от индексите на елем.Започ.с иниц.на a[i]=i, за i=0,…,N-1.Целта на сорт.е да пренаредим индекс.масив а,така че a[0] да дава индекса на елем.данни с най-малък ключ,и т.н.Така манипулираме индексите вместо записите.

Др.възмож.е да използ.указатели,т.е.реализ указателна сорт.За сорт.на масив от елем. указ.сорт.е ⇔на индексната но с адресите на масива,добавени към вс.индекс.Указ. сорт обаче е по-обща,защото указ.мд сочат навсякъде,а сорт.елем.не е необх.да са фик-сирани по размер.Ако а е масив от указ. към ключове,тогава викането на сорт.ф-я ще доведе до това,че указ.ще се пренаредят така че ако се обръщаме към тях последов., ще достигнем до ключовете в сорт.ред. Ре-ализ.сравн.като следваме указ.,а размените релиз.чрез размяна на указ.

Глав.причина да използ.индекси или указ. е да избегнем вмешателство в сорт.данни. Мд сорт.файл даже ако всичко което имаме е достъп до него само за четене.Даже с ня-колко масива от индекси или указ.мд сорт. 1 файл по няколко ключа.

23.Пирамидална сортировка.Базирани на пирамида алгоритми.Конвертиране в пирамида.Сортиране в пирамида.

Един метод който усъвършенства метода на пряката селекция е метода на пирамидалното сортиране. За да говорим обаче за този метод е нужно да построим структура от данни , която да позволява избирането на най-малкия елемент със сложност O(logn) а не с O(n). Тогава общото време на бързодействието ще е n*O(n*logn) = O(n*logn).

Тази структура също така трябва да позволява бързо да се вкарват нови елементи (за да може да се построи бързо от изходния масив) и да се премахва максималния елемент(той ще се помества във вече сортираната част на масива – неговия десен край).

И така пирамида (Heap) се нарича бинарно дърво с височина k, в което:

- Всички възли имат дълбочина k или k-1
- При това ниво k-1 е напълно заето, а ниво k е запълнено частично от ляво надясно
- Изпълнено е свойството на пирамидата: *всеки елемент е по-малък или равен на родителя си*

Построяването на пирамидата може да започне от a[k]...a[n], k = [size/2]. Тази част на масива удовлетворява свойството на пирамидата, тъй като не съществуват индекси i,j: i = 2i+1 (или j = 2i+2). Просто заради това че тези индекси са извън границата на масива.

Следва да се отбележи че е неправилно да се казва че a[k]..a[n] се явява пирамида ако се разглежда самостоятелно. Това, говорейки конкретно, не е вярно: елементтите му могат да са произволни. Свойството на пирамидата се съхранява само в границите на изходния масив a[0]...a[n].

По нататък ще разширяваме част от масива, която има това полезно свойство, като добавяме един елемент на всяка стъпка. Този елемент ще е онзи който предхожда готовата част.

За да може при добавяне да се съхрани пирамидалната структура ще използваме следната процедура за разширяване на пирамидата a[i+1]..a[n] с елемент a[i] наляво:

1. Гледаме кои са синовете (в масива това са a[2i+1] и a[2i+2]) и избираме по-големия от тях
2. Ако избраният е по-голям от a[i] – сменяме го с a[i] и отиваме на стъпка 2 като имаме предвид новото положение на a[i]. В противен случай слагаме край на процедурата.

Казвме че новия елемент се пресява през пирамидата:

Строенето на пирамидата става със следния код :

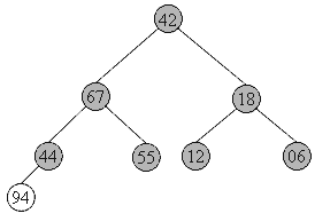
```
for(i=size/2-1; i >= 0; i--) downHeap(a, i, size-1);
```

като сме описали функцията `downHeap();`

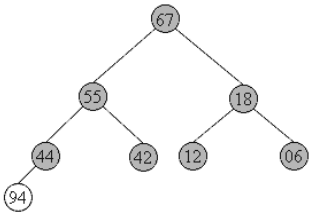
```
public void downHeap(int a[], int k, int n)
{
            int new_elem;
            int child;
            new_elem = a[k];
            while(k <= n/2)
            {
                child = 2*k;
                if( child < n && a[child] < a[child+1] ) child++;
                if( new_elem >= a[child] ) break;
                a[k] = a[child];
   k = child            }
          a[k] = new_elem;
}
```

Отчитайки височината на пирамидата h <= log n, **`downHeap()`** `изисква O(log n) време.`

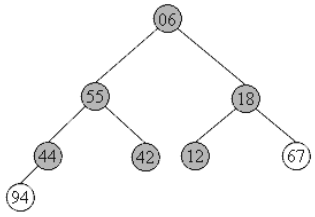
И така, задачата за построяване на пирамидата от масива е успешно решена. Както се вижда от свойството на пирамидата в корена и винаги се намира най-големия елемент. От тук се ражда и идеята за фаза 2 на алгоритъма.



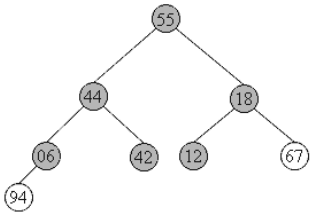
обменяли 94 и 42, забыли о 94



просеяли 42 сквозь 67, 55



обменяли 06 и 67, забыли о 67



просеяли 06 сквозь 55, 44…

1. Взимаме най-горния елемент на пиирамидата a[0]...a[n] (първият в масива) и го разменяме с последния. Сега забравяме за този елемент и по-нататък разглеждаме массива a[0]...a[n-1]. За превръщането му в пирамида е достатъчо да се пресее само новия елемент.
2. Повтаряме стъпка 1, докато обработваната част от масива не намалее до 1 елемент.

Очевидно в края на масива всеки път ще попада най-големия елемент от текущата пирамида, затова в дясната част постепенно възниква подредена последователност.

24.Списъци.Типове и реализация (през шаблон и обекти). Приложни аспекти.

Най–често срещаните и използвани са линейните (списъчни) структури. Те представляват абстракция на всякакви видове редици, последователности, поредици и други подобни от реалния свят. **Списък** е линейна структура от данни, която съдържа поредица от елементи. Списъкът има свойството дължина (брой елементи) и елементите му са наредени последователно.

Списъкът позволява добавяне на елементи на всяко едно място, премахването им и последователното им обхождането. Както споменахме по-горе, един АТД може да има няколко реализации.

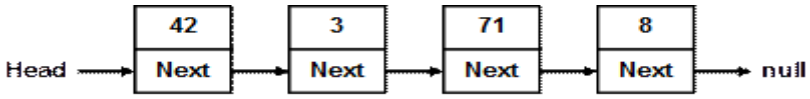
Статичен списък (реализация чрез масив)

Масивите изпълняват много от условията на АТД списък, но имат една съществена разлика – списъците позволяват добавяне на нови елементи, докато масивите имат фиксиран размер.

Въпреки това е възможна реализация на списък чрез масив, който автоматично увеличава размера си при нужда . Такъв списък се нарича **статичен**.

Свързан списък (динамична реализация)

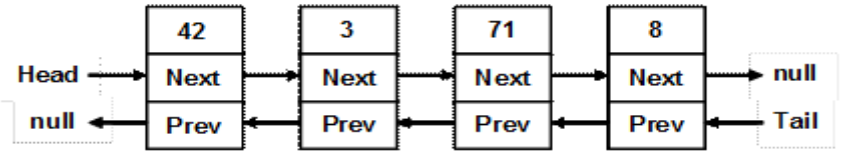
Както видяхме, статичният списък има един сериозен недостатък – операциите добавяне и премахване от вътрешността на списъка изискват пренареждане на елементите. При често добавяне и премахване (особено при голям брой елементи) това може да доведе до ниска производителност. В такива случаи се използват т. нар. свързани списъци. Разликата при тях е в структурата на елементите – докато при статичния списък елементите съдържат само конкретния обект, при динамичния списък елементите пазят информация за следващия елемент. Ето как изглежда един примерен свързан списък в паметта:



Изтриването по стойност на елемент работи като изтриването по индекс, но има 2 особености: търсеният елемент може и да не съществува и това налага допълнителна проверка; в списъка може да има елементи със стойност **null**, които трябва да предвидим и обработим по специален начин. За да работи коректно изтриването, е необходимо елементите в масива да са сравними.

Двойно свързани списъци

Съществува и т. нар. **двойно свързан списък** (двусвързан списък), при който всеки елемент съдържа стойността си и два указателя – към предходен и към следващ елемент (или **null**, ако няма такъв). Това ни позволява да обхождаме списъка, както напред така и назад. Това позволява някои операции да бъдат реализирани по ефективно. Ето как изглежда един примерен двусвързан списък в паметта:



Реализация чрез шаблони и обекти

Когато по-късно търсим даден елемент, ние го получаваме като **обект** и се налага да го превърнем в изходния тип. Не ни се гарантира, обаче, че всички елементи в списъка ще бъдат от един и същ тип. Освен това превръщането от един тип в друг отнема време, което забавя драстично изпълнението на програмата.За справяне с описаните проблеми на помощ идват шаблонните класове. Те са създадени да работят с един или няколко типа, като при създаването си ние указваме какъв точно тип обекти ще съхраняваме в тях.

25.Стек.Абстракция и реализация.Приложни аспекти: постфиксен запис и преобразувания от практиката.

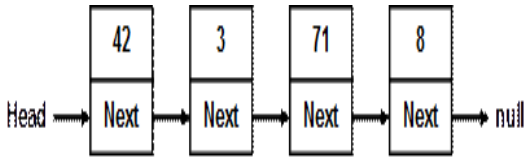
Можем да добавяме елементи най-отгоре и да извличаме последния добавен елемент, но не и предходните (които са затрупани под него). Стекът е често срещана и използвана структура от данни.

Абстрактна структура данни "стек"

Стекът представлява структура от данни с поведение "последният влязъл първи излиза".Елементите могат да се добавят и премахват само от върха на стека.

Статичен стек (реализация с масив)

Както и при статичния списък можем да използваме масив за пазене на елементите на стека. Ще пазим индекс или указател, който сочи към елемента, който се намира на върха. Обикновено при запълване на масива следва заделяне на двойно повече памет, както това се случва при статичния списък. Ето как можем да си представим един статичен стек:



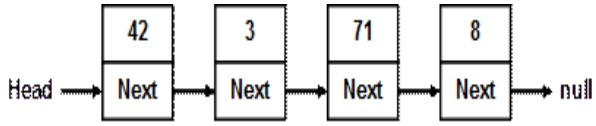
Четене: чете се елементът сочен от връх,след което върхът се намалява с 1.

```
Procedure chetene
(var stack:TypeStack;var vryh:integer);
Begin
        If vryh=0 then
          Writeln("stekyt e prazen")
        Else begin
        X:=stack[vryh];
        Vryh:=vryh-1;
        End;
End;
```

Както и при статичния масив се поддържа свободна буферна памет с цел по-бързо добавяне.

Свързан стек (динамична реализация)

За динамичната реализация ще използваме елементи, които пазят, освен обекта, и указател към елемента, който се намира "по-долу". Тази реализация решава ограниченията, които има статичната реализация както и необходимостта от разширяване на масива при нужда:



Четене: чете се елементът , сочен от връх, запомня се стойността връх в p, връх се пренасочва към следващия елеент и се изтрива елементът, сочен от p. Проверката за празен стек се прави преди викането на процедурата.

Procedure chetene(var x : Real);

```
Var p : strelka;
Begin
        X:=vryh^.inf;
        P:=vryh;
        Vryh:=vryh^.sledvasht;
        Dispose(p);
End;
```

Когато стекът е празен, върхът има стойност **null**. При добавяне на нов елемент, той се добавя на мястото, където сочи върхът, след което върхът се насочва към новия елемент. Премахването става по аналогичен начин.

26.Опашки.Абстракция и реализация.Използване на опашки.

Структурата "**опашка**" е създадена да моделира опашки, като например опашка от чакащи документи за принтиране, чакащи процеси за достъп до общ ресурс и други. Такива опашки много удобно и естествено се моделират чрез структурата "опашка". В опашките можем да добавяме елементи само най-отзад и да извличаме елементи само най-отпред.

Нека, например, искаме да си купим билет за концерт. Ако отидем по-рано ще си купим първи от билетите. Ако обаче се забавим ще трябва да се наредим на опашката и да изчакаме всички желаещи преди нас да си купят билети. Това поведение е аналогично за обектите в АТД опашка.

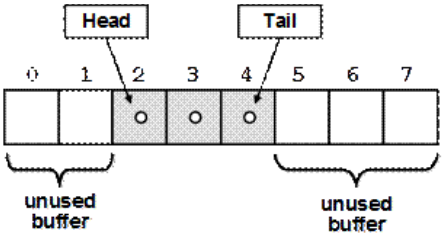
Абстрактна структура данни "опашка"

Абстрактната структура опашка изпълнява условието "първият влязъл първи излиза". Добавените елементи се нареждат в края на опашката, а при извличане поредният елемент се взима от началото (главата) ѝ.

Както и при списъка за структурата от данни опашка отново е възможна статична и динамична реализация.

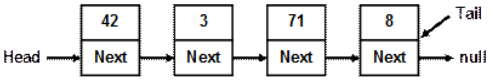
Статична опашка (реализация с масив)

В статичната опашка отново ще използваме масив за пазене на данните. При добавяне на елемент той се добавя на индекса, който следва края, след което края започва да сочи към ново добавения елемент. При премахване на елемент се взима елементът, към който сочи главата, след което главата започва да сочи към следващия елемент. По този начин опашката се придвижва към края на масива. Когато стигне до края, при добавяне на нов елемент той се добавя на първо място. Ето защо тази имплементация се нарича още **зациклена опашка**, тъй като мислено залепяме началото и края на масива и опашката обикаля в него:



Свързана опашка (динамична реализация)

Динамичната реализация на опашката много прилича на тази на свързания списък. Елементите отново съдържат две части – обекта и указател към предишния елемент:



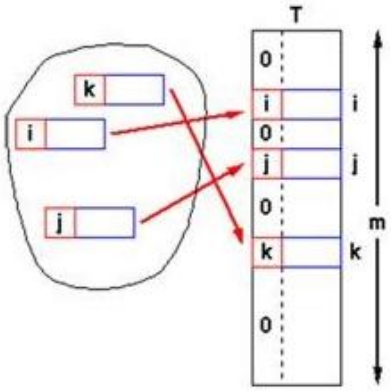
Тук обаче елементите се добавят в края на опашката, а се вземат от главата, като нямаме право да взимаме или добавяме елементи на друго място.

27.Хеш таблици.Идея, функции, избор на оптимална функция

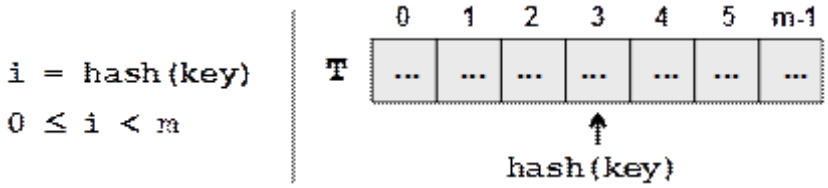
Реализацията с хеш-таблица има важното предимство, че времето за достъп до стойност от речника, при правилно използване, теоретично не зависи от броя на елементите в него.
За сравнение да вземем списък с елементи, които са подредени в случаен ред. Искаме да проверим дали даден елемент се намира в него. В най-лошия случай, трябва да проверим всеки един елемент от него, за да дадем категоричен отговор на въпроса "съдържа ли списъкът елемента или не". Очевидно е, че броят на тези сравнения зависи (линейно) от броя на елементите в списъка.
При хеш-таблиците, ако разполагаме с ключ, броят сравнения, които трябва да извършим, за да установим има ли стойност с такъв ключ, е константен и не зависи от броя на елементите в нея. Как точно се постига такава ефективност ще разгледаме в детайли по-долу.
Когато реализациите на някои структури от данни ни дават време за достъп до елементите й, независещ от броя на елементите в нея, се казва, че те притежават свойството **random access** (**свободен достъп**). Такова свойство обикновено се наблюдава при реализации на абстрактни структури от данни с хеш-таблици и масиви.

Какво е хеш-таблица?

Структурата от данни хеш-таблица обикновено се реализира с масив. Тя съдържа наредени двойки (ключ, стойност), които са разположени в масива на пръв поглед случайно и непоследователно. В позициите, в които нямаме наредена двойка, имаме празен елемент (**null**):



Размерът на таблицата (масива), наричаме **капацитет (capacity)** на хеш-таблицата. **Степен на запълненост (load factor)**, наричаме реално число между 0 и 1, което съответства на отношението между броя на запълнените елементи и текущия капацитет. На фигурата имаме хеш-таблица с 3 елемента и капацитет **m**. Следователно степента на запълване на тази хеш-таблица е **3/m**.
Добавянето и търсенето на елементи става, като върху ключа се приложи някаква функция **hash(key)**, която връща число, наречено **хеш-код**. Като вземем остатъка при деление на този хеш-код с капацитета **m** получаваме число между **0** и **m-1**:
На фигурата е показана хеш-таблица **T** с капацитет **m** и хеш-функция **hash(key)**:



Това число ни дава позицията, на която да търсим или добавяме наредената двойка. Ако хеш-функцията разпределя ключовете равномерно, в болшинството случаи на различен ключ ще съответства различна хеш-стойност и по този начин във всяка клетка от масива ще има най-много един ключ. В крайна сметка получаваме изключително бързо търсене и бързо добавяне. Разбира се, може да се случи различни ключове да имат един и същ хеш-код. **Използвайте реализация на речник чрез хеш-таблици, когато се нуждаете от максимално бързо намиране на стойностите по ключ.**
Капацитетът на таблицата се увеличава, когато броят на наредените двойки в хеш-таблицата стане равен или по-голям от дадена константа, наречена **максимална степен на запълване**. При разширяване на капацитета (най-често удвояване) всички елементи се преподреждат според своя хеш-код и стойността на новия капацитет. Степента на запълване след преподреждане значително намалява. Операцията е времеотнемаща, но се извършва достатъчно рядко, за да не влияе на цялостната производителност на операцията добавяне.

28.Философия на алгоритмизирането: Постъпателни алгоритми (greedy алгоритми). Проблемът – оптимална диспечеризация

работи поетапно. На всеки етап се взима изглеждащото за най-добро решение, независимо от последиците. Т.е.локален оптимум. Или стратегията е “take what you can get now”.В края на алг.приемаме, че локал.съвпада с глобал.оптимум. Пр.за това е стратегията на връщане на монети: връща на всеки етап монета с мак. възможна ст-ст

*$17.61 → 1*$10 +1*$5 +2*$1 + 2*0.25(quarters) +1*0.1 (dime) + 1*0.01(penny).*

Този алгоритъм не работи във всички монетарни системи. Друг пример: авт. трафик.

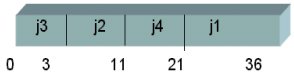
simple scheduling problem

имаме дадени задачи j1 …jN с времена на изпълнение t1…tN и 1 процесор. Искаме разпределяне с цел мин. средно време на завъшване на задача (т.е. на коя мин. средно завършва задача). (няма прекъсване на задача).

Задача	време
j1	15
j2	8
j3	3
j4	10

Първата завършва след 15, втората след 23, третата след 26, след 36 → общо 100 мин/4 = 25 средно ще завършва задача.

Друга подредба е показана по-долу:ср.време 17.75



Стратегията е най-късата задача – най-напред(тя участва най-много във формиране времената на следв.зад).

Това винаги дава оптимално решение. Нека първата пусната задача е **j i1…j iN** с времена на завършване: **t i1, t i1+t i2, t i1 +t i2 +t i3**.

Цената на разпределението става:

$$C = \sum_{k=1}^{N} (N-k+1)t_{ik}$$

$$C = (N+1)\sum_{k=1}^{N} t_{ik} - \sum_{k=1}^{N} k*t_{ik}$$

I член не зависи от подредбата, а само втория (“к”).

Ако предположим, че в подредба съществува x >y, такова че **t ix<t iy**, то ако разменим местата на **j ix и j iy**, втория член нараства и следоват. общата цена намалява. следоват. всяка подредба в която работите не са наредени в нарастващ ред е неоптимална. Затова се дава приоритет на най-късите задачи първо

.

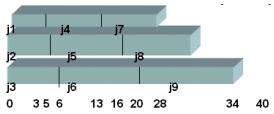
многопоцесорен вариант

Имаме j1……jN; ti….tN и P процесора.

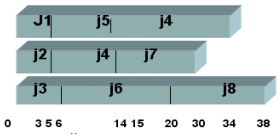
Подрежд. първо на късите зад. Нека P=3:

задача	време
j1	3
j2	5
j3	6
j4	10
j5	11
j6	14
j7	15
j8	18
j9	20

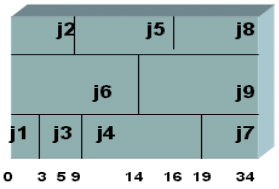
едно решение (започваме с къси задачи и циклим по процесори):



Общо време на завършване 165. Средно 165/9 = 18.33. Друга стратегия (когато P дели N точно) е: за всеки 0 <= I < N/P поставяме последоват. следващите P задачи на различен процесор.



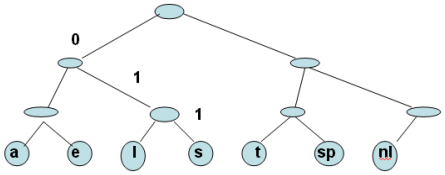
минимизация на крайното време на завършване на всички задачи по-горе това е 40 или 38. Ето подредба с t = 34:



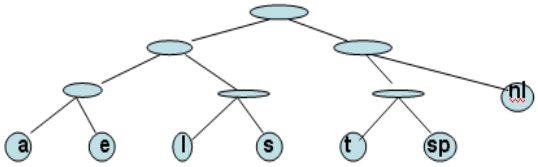
очевидно тази подредба не може да се подобри, защото всички са заети през цялото време. Това обаче е друга стратегия – най-късо общо завършване.

29.Постъпателни алгоритми – синтез на кодове Huffman

Нормално ASCII има 100 печатащи се символа ➔ log100 =7 бита + 1 бит контрол по четност.Т.е. за С символа са нужни logC бита за кодиране.
Нека във файл имаме само символи a,e,l,s,t , blank, NL. Нека след статистика знаем че във файла има
10 –a; 15 e; 12 l; 3 s; 4 t; 13 blanks; 1 ML. Нужни са 174 бита за съхраняване на поредицата
Ще представим стратегия постигаща за нормален файл 25%пестене и 60%за дълги файлове.Варираме с дължината на кода за разл символи. Най-честите → с най-къс код.При едн. честота – няма значение кой.Ето дърво, даващо н-н за кодировка на азбука от 7 символа:

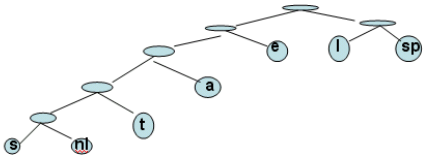


Данни има само в листата.Кодировката е еднозначна
Ако символ с i е в дълбочина d i и се среща f i пъти, то общ.цена на кодираната инф. е ∑d I * f i.Едно подобре-ние става при свиване дълбочината за възли с 1 листо:



Общата цена става 173, което изобщо не е подобрение.
Винаги работим с пълно дърво. При символи само в листата, всяка последователност от 0 и 1 е еднозначна и мд се декодира.Това е т.нар.префиксно кодиране.

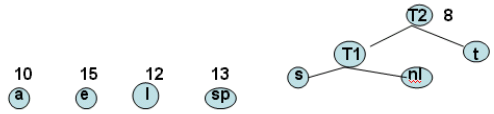
Проблемът се сведе до: намиране на full binary tree с мин. цена, в което символи има само по листата. Ето оптималното за примера (цена 146):



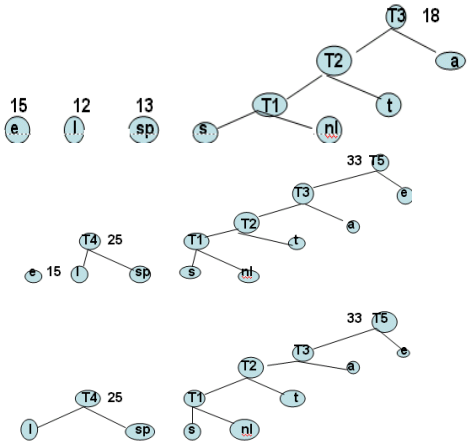
Алгоритъм на Huffman
Имаме С символа.Правим набор дървета.Тегло за дър-во се получава от сума на честотите в листата му. С-1 пъти избираме 2 дървета Т1 и Т2 с мин. тегла и ги обединяваме. В нач. имаме С дървета с по 1 възел. В края на алг. имаме 1 дърво и това е оптимал. кодово дърво.

Ето за нашия пример Начало:

първо обединение (няма значение кое е ляво, така че реализиращата процедура няма да прави връщания назад). Теглото на новополученото д-во е сума:



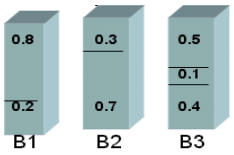
Реализацията в прог. е лесна: правим нов възел,устано-вяваме 2 указателя и изчисл. тегло.продължаваме:



краят (след обединението) беше вече показан
Накратко за доказателството, че Хофмановия алгоритъм води до оптимален код:
1. полученото д-во е пълно, така че не може да се подобрява с местене възли нагоре.
2. 2 букви “a, б” с мин. честота са най-долу. Доказваме с използване на обратно твърдение: ако това не е така , (поне едното от а и б не е в най-дълбоко листо), то има някакво г (дървото е пълно) което е там. Но щом а е по-рядко срещано от г, то най-лесно ще подобрим общата цена като разменим а и г .
3. Очевидно, 2 символа във възли с еднаква дълбочина могат да се разместват без променяне на оптималния избор. Отчитаме, че може да сливаме символ – дърво с слято вече. Тогава очевидно азбуката ни се е проме-нила (напр. е,Т4,Т3). Но крайните символи изгражда-щи напр Т3 са по-дълбоко,но и по-рядко срещани от е.
Алг.е greedy,защото на всеки етап правим сливане без оглед на глобал.оптимиз., а само в локален оптимум.
**Разбира се кодиращата инф.следва да се изпрати в началото на файла при трансмисия за да е възможно декодиране.За малки файлове това е лошо(много добавена инф.).
***Алгоритъмът е двупасов – първо се събират данни за честотата, после се кодира. За големи файлове това не е добре – има подходи за обединяване задачите.

30.Постъпателни алгоритми – проблемът „пакетиране” Online & FirstFit

Тези алгоритми са бързи, но не винаги дават оптимално решение. Имаме N пакета с размери s1….s N (0 <= s i <1). Искаме да ги пакетираме в min брой торби, като всяка торба има обем 1. Ето пример:



Съществуват 2 версии на решения:on-line всяка единица се поставя преди следваща (няма връщане назад). Off-line – първо изследваме всички, тогава започваме пакетирване.

On-line алгоритми

Не винаги дават оптималното решение. Разглеждаме поредица от I1 ел.от общо M в тегло ½ -б, следвани от останалите с тегло ½+б (б е нещо малко): очевидно, всички могат да се пакетират в M (1 малка + 1 голяма). Нека алгоритъм A прави това пакетиране. Тогава A ще постави малките I1 (половината) ел. всеки в отделен чувал (а не по 2) – общо напр. в M торби. Но как ще знае че следващите са по-големите, а не обратно. Значи A винаги ползва 2M торби вместо M.

Тъй като краят (общия брой)е неизвестен , алгоритмите от този вид дават само локална гаранция.

Теорема 1 :съществуват вх. поредици, които карат всеки on-line алгоритъм да използва поне 4/3 от оптималния брой торби.

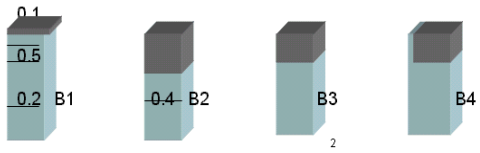
Док. Предполагаме обратното и нека M е четно. Алгоритъм A обработва входна поредица I1 от M малки ел., следвани от M големи. Вече сме обработили първите M ел. и сме използвали б торби. (оптималното е б=M/2). Значи (оптимистично)предполагаме че алгоритъмът постига : 2б/M < 4/3 → б/M <2/3

Нека всички елементи са обработени. Имаме б торби с първите б ел. (нали алгоритъмът работи оптимално и не може да оставя по 2 големи ел. за 1 торба). Тогава след края първите б торби ще имат по 2 ел. (малък и голям), а следващите - по 1 голям. За 2M ел. ще са нужни 2M – б торби. Оптимумът беще M. Следователно: (2M – б) / M < 4/3→ б/M >2/3

Имаме противоречие. Следователно няма on-line алгоритъм, даващ по-добро от 4/3 спрямо оптималното решение.

First Fit

Предишният алгор. създава нова торба не винаги когато това е нужно, защото разглежда само последната. Напр. за схемата горе, ел. 0.3 може да се постави в B1 или B2 а не в нова. First Fit стратегията сканира торбите подред за да установи дали може да постави новия ел.



Тъй като се сканират предходните торби, → O(N). Може да се сведе до O(N log N) – ако сканираме както при бинар. търсене.

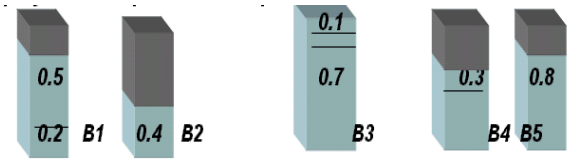
Видно е че във всеки момент най-много 1 торба може да е запълнена ½, тъй като ако са 2, то могат да се поставят в 1. Значи най-много 2M спрямо оптимумът от M торби.

31.Постъпателни алгоритми – проб лемът пакетиране. Методи BestFit & Next Fit

Next fit

проверяваме дали следващата в поредицата може да се постави в торбата, която съдържа последния ел. ако не – нова торба.Работи с линейно време.

Ето резултат над предходната поредица:



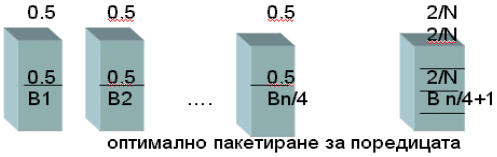
T2 Нека M е оптималния брой торби за пакетиране на I ел. Next fit никога не използва повече от 2M торби.

Док. Разглеждаме съседните B j и B j+1.

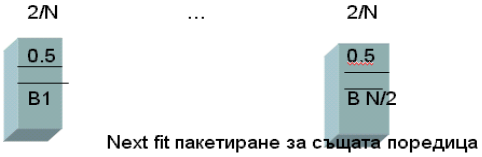
Сумата от обемите в тях е >1, иначе в 1 торба. Ако направим тези разсъждение за всички съседни, виждаме че най-много ½ пространство е похабено. Използвани са най-много двукратен брой торби.

Ето най-лоша последователност на входа:нечетни si имат размер 0.5, четни – размер 2/N. Нека N се дели на 4. Оптимално пакетиране е от N/4 +1.

Реалното в този алгоритъм заема N/2. Т.е почти двойно:



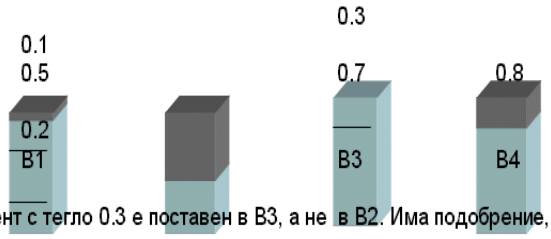
оптимално пакетиране за поредицата



Next fit пакетиране за същата поредица

Best Fit

Вместо да поставяме нов елемент в първа възможна торба, го поставяме там където е възможно, но и има най-малко свободно място.Елемент с тегло 0.3 е поставен в B3, а не в B2. Има подобрение, но не и за лошите поредици.

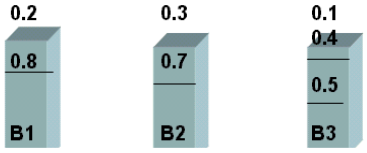


Елемент с тегло 0.3 е поставен в B3, а не в B2. Има подобрение, но не и за лошите поредици.
Затова ограниченията остават същите – 1.7 от оптимума

32.Off-Line алгоритми. Помощни теореми и оценки на подхода

Off-line алгоритми

Сега можем да анализираме цялата поредица, преди да подреждаме. Големият проблем на on-line алгоритмите е че не пакетират добре големи ел. когато те идват към края. Добре би било големите да ги сортираме отпред.
Това е алгоритъмът first fit decreasing:



В случая това е и оптимал.решение. Не винаги е така. Значи считаме, всички ел. са вече подредени намаляващо.Ще док, че ако оптимал.пакетиране изисква M торби, сега са нужни не-повече от (4M + 1)/3

Лема 1 Нека N ел. (сортирани намаляващо) с размери s1…sN могат да се подрдят оптимално в M торби.Тогава всички ел., които first fit decreasing поставя в допълнителни торби (над M) имат размер най-много 1/3.

Док. Нека елемент i е първия в торба M+1. Да докажем si <=1/3. Предполагаме si >1/3. Следва s1,…si-1 >1/3 (иначе лемата е доказана). Тогава в торби B1….BM има най-много ел (нали са >1/3). Тогава в първите торби има по 1 ел. а в оставащите до M по 2 ел.
Ако приемем обратното: в торба Bx има 2 ел. , а в торба By → 1 el. и 1<= x < y <= M. Нека x1 и x2 са елементите в Bx и y1 е елементът в By, x1>=y1, а също и x2 >=si то: x1 +x2 >= y1 +si
Значи si може да се пъхне в By, но това не беше възможно по предположение. Значи, ако si>1/3, в момента, когато го обработваме, в първите j торби има по 1 ел. а в оставащите M-j по 2 ел.

Сега да покажем че няма начин всички елементи да се сложат в торби (тогава не би оставал външен <1/3)
За първите j ел. няма начин 2 да се сложат в 1 торба (доказахме че в началото са по 1). Също така никой елемент до si не може да се пъхне в първите торби. Значи j торби са по 1 ел. и елементите с размери
sj+1,s j+2…. S i-1 са в M-j торби.Общият брой ел. в тях е 2(M – j)
Следователно, елемент si>1/3 няма начин да се вкара в първите M торби: той не може да се вкара в първите j торби, защото те съдържат по 1 ел. Не може и в оставащите M-j торби, защото в тях трябва да се появят 2(M-j) +1 елемента. Това означава в някоя торба следва да има по 3 ел (а всеки беше >1/3).Очевидно предвиж-дането в нач. беше грешно и s i <=1/3

Лема 2 Броят ел. поставени в допълнителните торби е най-много M-1

Док:Приемаме обратното: в допълнителните торби има най-малко M ел. Знаем si <= M Нека Bj е поела общо тегло Wj (1<= j <= M) Нека първите M ел. в допълнителни торби са с размер x1…xM.
Тъй като елементите в първите M торби + елементите в първите M допълнителни са част от всички то:

$$\sum_{i=1}^{N} si <= \sum_{j=1}^{M} Wj + \sum_{j=1}^{M} xj >= \sum_{j=1}^{M} (Wj + xj)$$

Wj +xj >1 защото иначе xj щеше да се вкара в Bj. Тогава:

$$\sum_{i=1}^{N} si > \sum_{j=1}^{M} 1 > M$$

Това е невъзможно, тъй като тези N ел. са пакетирани в M торби.
Следователно допълнителните торби са най-много M-1

Теорема Нека M е оптимал. брой торби за I елемента. First Fit decreasing никога не използва повече от (4M+1)/3 торби.
Имаме M-1 допъл.ел.с макс.размер 1/3.
Значи допълн. торби са най-много (M-1)/3.
Общият бр. торби използвани в метода е :
(4M –1) /3 <= (4M +1)/3
First Fit decreasing е добър и бърз подход

33.Стратегията Разделяй и владей – анализ на времето за изпълнение

Анализ на времето на изпълнение:

= време за изпълнение на частите + константно време на излъчване крайния резултат: T(N)= 2T(N/2) +O(N).

Теорема:Реш.на уравн.$T(N)=aT(N/b)+O(N^k)$

a>=1; b >1 (a – броят на зад.; в – частите) е:

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{за } a>b^k \\ O(N^k \log N) & \text{за } a = b^k \quad \text{//най-често} \\ O(N^k) & \text{за } a<b^k \end{cases}$$

Док:

Нека $N = b^m$ (най-често). Тогава $N/b = b^{m-1}$ и $N = (b^k)^m = b^{km}$.

Ако T(1) =1 и игнорираме константния фактор $\Theta(N^k)$, то имаме:

$T(b^m) = aT(b^{m-1}) + (b^k)^m$

делим на a^m:

$T((b^m)/a^m) = T(b^{m-1})/a^{m-1} + \{b^k/a\}^m$

ако продължим за останалите m:

$T(b^{m-1})/a^{m-1} = T(b^{m-2})/a^{m-2} + \{b^k/a\}^{m-1}$

$T(b^{m-2})/a^{m-2} = T(b^{m-3})/a^{m-3} + \{b^k/a\}^{m-2}$

….

$T(b^1)/a^1 = T(b^0)/a^0 + \{b^k/a\}^1$

сумираме отляво и отдясно:

$$T(b^m)/a^m = 1 + \sum_{i=1}^{m} \{b^k/a\}^i = \sum_{i=0}^{m} \{b^k/a\}^i$$

следователно: $T(N) = T(b^m) = a^m \{b^k/a\} \text{ ако } a>b^k$

$$T(N) = O(a^{\log_b N}) = O(N^{\log_b a}) = O(N^k \log_b N) = O(N^k \log_b N)$$

то сумата е геом. намаляваща серия $\sum_{i=0}^{N} A^i = (A^{N+1} - 1)/(A-1)$. Това води към конст.:

$T(N) = O(a^{\log_b N}) = O(N^{\log_b a})$ // увеличили сме

//основата, намалили сме степента, напр $2^k < 3^k$

ако a = в, всеки член =1.Членовете са 1 + $\log_b N$ (=m) на брой и $a = b^k$ →

$\log_b a = k$:

и ако $a< b^k$, то членовете са геом. серия в която всеки е >1 и като вземем предвид формулата от първата лекция за редици:

$$\sum_{i=0}^{N} A^i = (A^{N+1} - 1)/(A-1)$$

то получаваме за случая:

$$T(N) = a^m((b^k/a)^{m+1} - 1)/((b^k/a) - 1) = O(a^m (b^k/a)^m) = O((b^m)^k) = O(N^k)$$

Това са и 3 случая на рекурентно разбиване на задача на подзад. Напр. сортировка чрез разделяне и сливане : a=b=2; k=1. Имаме втория случай и отговор: O(NlogN)

Ако имаме 3 проблема и всеки е над половината данни и комбинирането за краен резултат изисква O(N) време, то a=3,b=2, k=1. $\log_2 3 \approx 1.59$

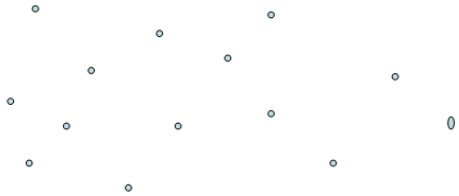
Имаме случай 1 и $O(N^{\log_2 3}) = O(N^1)$ Ако в горния пример времето за обединяване на резултата беше $O(N^2)$,то сме във случай 3 и: $O(N^2)$

34.Разделяй и владей -Проблемът“на-близкостоящи точки”

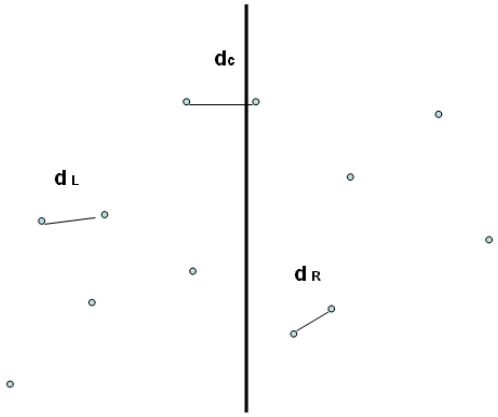
Имаме P точки в равнината, p1=(x1,y1)…..

Разстоянието м/ду тях :[(x1 –x2)² + (y1-y2)²] Търсим най-близкостоящите точки.

Имаме N(N-1)/2 разстояния. Пълен тест: O(N ²) Друг подход: Нека точките са сортирани по x координатата си. Ако не са, допълнително време O(NlogN):



Разделяме ги по вертикала на 2 равни части P L P R:



dl и dr могат да се изчислят рекурсивно (O(NlogN) . Остава да изчислим dc за O(N) време и да постигнем: O(NlogN) +O(N) Нека δ=min(dl,dr). Ще изчисляваме dc само ако подобрява б. Точките в лентата са малко: една по една (O(N)):

//points are all in the strip

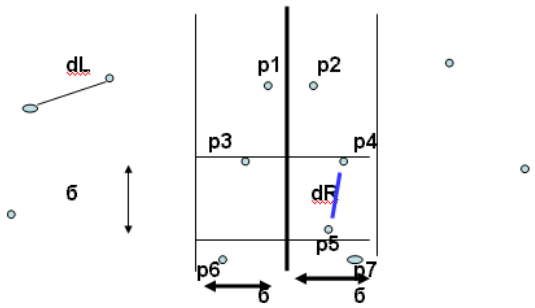
```
for(I=0;I< numPoinsInStrip; I++)
  for (j=I+1; j<numPoinsInStrip;j++)
   if(dist(pi,pj) <б)  б = dist(pi,pj);
```

ако точките в лентата са много (почти всички), горния подход е слаб. Нека т. са сортирани в лентата по y коорд. Тогава ако y коорд на pi и pj се различават с повече от б, можем да преминем към p i+1.

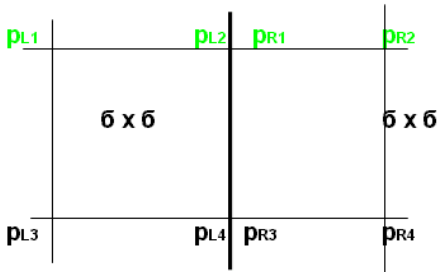
Ускорен вариант:

```
for(I=0; I<numPoinsInStrip; I++)
  for( j=I+1; j< numPointsInStrip; j++)
    if (pi and pj ’coordinates differ by more
       than б) break; // next pi.
    else
    if(dist(pi,pj)<б) б = dist(pi,pj);
```

само p4 и p5 се разглеждат, съобразно горната модификация:



най-много 7 т. могат да се разглеждат в правоъгълната област б *2б:



Времето за това < O(N).

Имаме O(NlogN) от рекурсивните повиквания отляво и отдясно + O(N). За сортирането по y е нужно още O(NlogN) за всяко рекурсивно повикване или общо O(Nlog ² N). (пълно претърсване: O(N ²).

Можем още да ускорим като предварително сортираме и по x и по y и изработим 2 списъка с точки, сортирани съответно. (O(NlogN) време в началото). После претърсваме списъка по x и махаме всички точки с коорд. > б. Автоматично остават само тези в линията и то сортирани по y. Това изисква O(N). общо време O(NlogN) +O(N)

35.Теоретични подобрения на аритметичните операции.Анализ

* Умножение на цели числа
- За малки ч-ла - линейно време.
- За големи, времето става квадратично.
Нека имаме N р-дни ч-ла X и Y. Знакът определяме лесно. По класически алгоритъм $O(N^2)$: всяка цифра на X се умножава по всяка на Y.

Нека: X= 61,438,521
Разбиваме X и Y:
Y=94,736,407 XY=5,820,464,730,934,047
X_l = 6,143 X_r = 8,521
Y_l = Y_r =

$$X = X_l * 10^4 + X_r; \quad Y = Y_l * 10^4 + Y_r$$

$$XY = X_l Y_l 10^8 + (X_l Y_r + X_r Y_l) * 10^4 + X_r Y_r$$

Имаме 4 умножения с N/2 цифри:
T(N)= 4T(N/2) + O(N)
според теоремата в началото това е:$O(N^2)$ - нямаме подобр.Трд намал броя умнож:
Xl Yr+Xr Yl=(Xl–Xr)(Yr–Yl)+Xl Yl+Xr Yr
Умножение на матрици
Ето стандартен алгоритъм с $O(N^3)$ време.

```
Matrix<int> operator*( const matrix<int> &a, const matrix<int> &b )
{int n = a.numrows();
matrix<int> c( n, n);
int I; for( I = 0; I < n; i++ )
for( int j = 0; j < n; j++ )
c[ I ][ j ] = 0;
for( I = 0; I < n; i++ )
for( int j = 0; j < n; j++ )
for( int k = 0; k < n; k++ )
c[ I ][ j ] += a[ I ] [ j ] * b[ I ] [ j ];
        return c;
}
```

Ако матр.умнож.се изпълн.в рекурсия:
$T(N) = 8T(N/2) + O(N^2) = O(N^3)$ =>няма подобрение.Трд се намал.подзад. <8.
Strassen дава divide &conquer алг. с из-ползване на 7рекурсивни повиквания :

Strassen подобрява това време. Осн. идея е разбиване на матриците на 4 квадранта:

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} * \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix} = \begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix}$$

Тогава:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

M1= (A12-A22)(B21+B22)
M2= (A11+A22)(B11+B22)
M3= (A11-A21)(B11+B12)
M4= (A11+A12)B22
M5= A11(B12-B22)
M6= A22(B21-B11)
M7= A21+A22)B11

Резултатът се получава така:
C11= M1+M2-M4+M6
C12= M4+M5
C21= M6+M7
C22= M2-M3+M5-M7
$T(N)=7T(N/2)+O(N)=O(N^{\log_2 7})=O(N^{2.81})$

36.Динамично програмиране – таблици вместо рекурсия

Рекурсията не е оптималната техника →в ред случаи рек. алгоритъм следва а се пренапише в нерекусивен вариан със запомняне межд. резултати в таблица.
Една техника за това е “динамичното програмиране”.
3.1 таблици вместо рекурсия. Ето неефективен вариант за изчисляване ч-лата на Фибоначи:

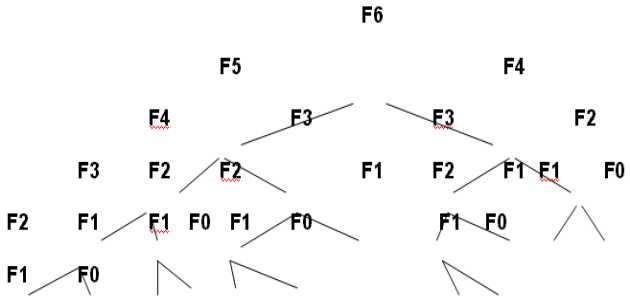
```
int fib( in n)
{ if( n <= 1 ) return 1;
  else return fib( n - 1 ) + fib( n - 2 ); }
```

T(N) >= T(N-1) +T(N-2)
сложноста е експоненциална. Мд съхраним изчислените вече F n-1; F n-2. :

```
int fibonacci( int n )
{ if( n <= 1 )
        return 1;
int last = 1;
```

int nextTolast = 1;
int answer = 1;
for(int I = 2; i <= n; i++)
{ answer = last + nextTo last;
nextTolast = last;
last = answer;
}return answer;}

Имаме O(N). Ако горната модификация не е направена, за изчисление на Fn се вика рекурсивно Fn-1 и Fn-2**…..**



Виждаме F3 се вика 3 пъти, F2 – 5 и т.н. Експлозивно нарастват сметките

37.Ускоряване на сортировката с паралелизми чрез стратегия разделяй и владей.

39.Динамично програмиране: Оптимално бинарно търсене в дърво.

Рекурсията не е оптималната техника →в ред случаи рек. алгоритъм следва а се пренапише в нерекусивен вариан със запомняне межд. резултати в таблица. Една техника за това е "динамич.програмиране".

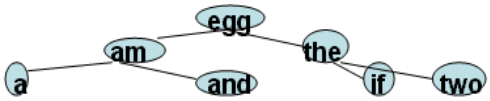
оптимално бинарно търсене в дърво

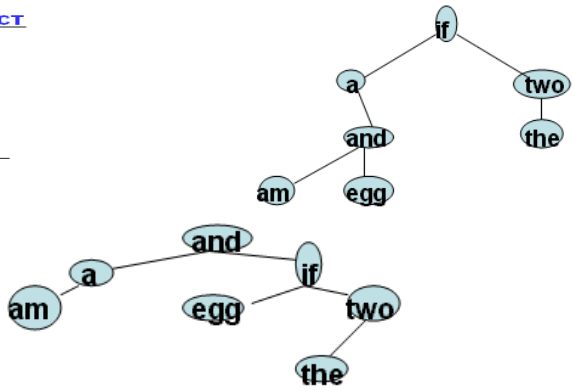
Имаме списък думи w1……wn с вероятности на появяване: p1…..pn

Искаме да подредим думите в бинарно дъво, така че общото време за намиране на дума да min. В бинарно д-во за да стигнем до ел. с дълбочина d, правим d+1 сравнения. Така че, ако wi е на дълбочина di, ние искаме да минимизираме

$$\sum_{l=1}^{N} p_i (1+d_i).$$

дума	вероятност
a	0.22
am	0.18
and	0.20
egg	0.05
if	0.25
the	0.02
two	0.08





първото използва greedy стратегия. Най-вероятната дума е най-горе. Второто е балансирано search tree (в дясно с по-ниско pi).Третото е оптималното – виж табл

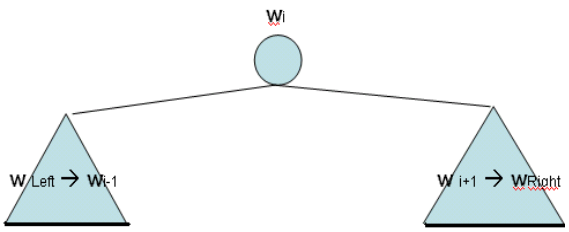
вход дума	вероятност	дърво 1 цена	достъп дълбоч	дърво2 цена	достъп дълб		дърво3 цена	достъп дълб
a	0.22	2	0.44	3	0.66	2	0.44	
am	0.18	4	0.72	2	0.36	3	0.54	
and	0.20	3	0.60	3	0.60	1	0.20	
egg	0.05	4	0.20	1	0.05	3	0.15	
if	0.25	1	0.25	3	0.75	2	0.50	
the	0.02	3	0.06	2	0.04	4	0.08	
two	0.08	2	0.16	3	0.24	3	0.24	
общо	1.00		2.43		2.70		2.15	

сравнение на 3 дървета с бинарно търсене

При optimal binary search tree данните не са само в листата (Hofman) и следва да удовлетворяваме критерий за binary search tree.

Поставяме сортирани според някакъв критерий думи wleft, wleft+1,….wright-1, wright в бинарно дърво. Нека сме постигнали оптималното бинарно дърво в което имаме корен wi и поддървета за които : Left <= i <- Right. Тогава лявото поддърво съдържа елементите wleft, …w i-1, а дясното – wi+1,……wright (критерий за binary search tree).

Поддърветата също правим оптимал. Тогава можем да напишем формулата за цената Cleft,right на оптимално binary search tree. Лявото поддърво има цена Cleft,I-1, дясното – Ci+1,right спрямо корена си.

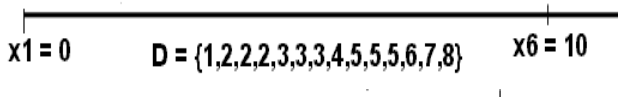


На основата на тази формула се гради алг. за цена на оптималното дърво.Търсим i, така че да се минимизира C Left,Right. Последователно се поставят за корени am, and, egg, if и се изчислява цена и оттам минималната цена. Например, когато and е корен, лявото поддърво am…am има цена 0,18 (вече определена), дясното – egg-if с цена 0.35 и pam+pand+pegg+pif = 0.68.

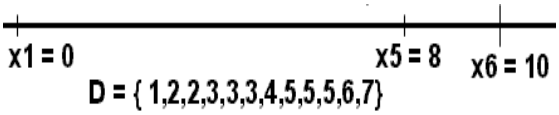
Тогава общата цена е 1,21.Времето е O(N ³) защото имаме тройно вложен цикъл

40.Алгоритми с backtracking: Проблемът – реконструиране

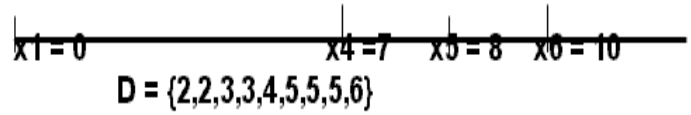
Достига до добри решения , но е непредвидимо бавен, поради връщанията. Пример: подреждане на мебели в къща. спира се на различни етапи и се кара до изчерпване. Проблемът – реконструиране (приложение Физика, молекулярна Биология..)
Нека имам N точки: p1,p2...pN подредени по оста x.
Дадени разстоянията N(N – 1) / 2. Нека x1 = 0
Ако имахме дадени координатите, лесно можем да изчислим разстоянията. O(N^2)
Нека дистанциите са дадени сортирано.
Задачата е: да се реконструират координатите на точките от дистанциите.
дадени: D ={1,2,2,2,3,3,3,4,5,5,5,6,7,8,10}
D е 15, → N=6 нека x1 = 0 очевидно x6 = 10



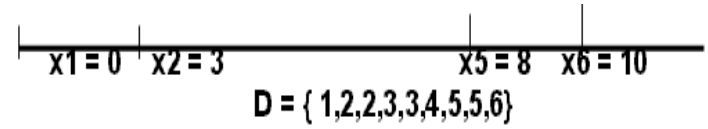
най-голямата оставаща дист. е 8, следоват или x2=2 или x5=8 откъдето и да тръгнем все ще стигнем (връщанията от неуспех са равновероятни).



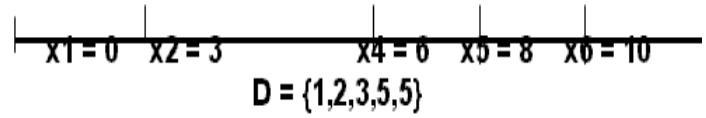
Нека x5=8. тогава x6-x5=2; x5-x1 = 8
7 е най-голямото разст. Или x4=7 или x2=3 (за да има разст. = 7 от x2 до x6).
Ако x4 = 7 – проверяваме разст. до x6 и x5 и виждаме че ги има във вектора с разст.
Ако решим x2=3, то 3-x1=3; x5-3=5 - също са във вектора. Значи и това става.
Избираме едното за да продълж. Нека x4=7



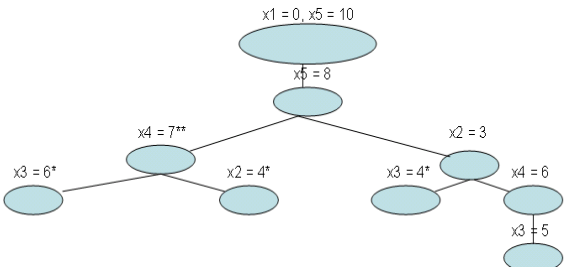
Сега макс. разст. е 6, така че или x3=6 или x2=4. Проверяваме x3=6 - не може. Проверяваме x2=4 – не може.Връщане назад.x4=7 отпадна. Сега опитваме x2=3



сега остава да изберем или x4=6 или x3=4. x3=4 е невъзможно. Значи остава x4=6:



остана x3=5, x1 = 0, x2 = 3, x3 = 5 x4 = 6 x5 = 8 x6 = 10 D = {}край!
ето граф на решенията:



*избраното решение води до разст. ,които липсват в D.
** върхът има само неуспяващи деца, следоват. този път да се изостави.
Поради backtracking анализът е вероятностен от... до... Ако няма връщане O($N^2 \log N$),ако има – достигаме до O (N^2),

41.Алгоритми от теория на игрите.Оптимизационни техники.

1.изчерпателен анализ на ходовете
2. стратегии: шах, крави /бикове крави/бикове е лесна за програмиране, защото броят е изчерпаем, известни са слабите ходове (могат да се вкарат в табл). Алг.никога не греши (никога не губи) и винаги ще спечели ако му се даде възмож.

Стратегия минимакс

разглеждаме "силата" на позициите: Ако води победа – оценката е +1, ако равенство – 0; ако е позиция с която компютърът губи –1. Такива позиции са терминални.

За останалите → рекурсивно се играе**.**

Значи стратегията е:противникът се стреми да минимизира стойността, играчът (комп) – да я максимизира. Пробват се всички възмож.в момента ходове.Избира се този с макс.ст-ст. За ход на противника – същото: прохождат се всички ходове и се избира този с минимална стойност.

```
Int TicTakToe::
findCompMove( int & bestMove)
{int i, responseValue;
int dc// don't care:ст-стта не се използ
int value;
if( fullBoard () )value = DRAW;
else
if(immediateCompWin(bestMove))//има  return COMP_WIN
//възм. за редица
else
{value = COMP_LOSS; bestMove = 1;
 for( i = 1; i < 9; i++ )
  { if( isEmpty( i ))
     { place( i, COMP );
       responseValue=findHumanMove(dc);
       unplace( i );
       if( responseValue > value )//търси^
        {      //update best move
            value = responseValue;
            bestMove = i;
         }
     }
  }
 } return value; }
```

играта на човека(ф-иите рекурсивно се викат,оценките за успеш.ход са противоположни

```
int TicTacToe::
findHumanMove( int & bestMove)
{int i, responseValue;
int value; int dc   //don't care
if( fullBoard()) value = DRAW;
else if( immediateHumanWin( bestMove) )
       return COMP_LOSS;
    else
    {value = COMP_WIN; bestMove = 1;
     for( i = 1; i <= 9; i++ ) //изпробва
                              // всяко квадратче
     {if(isEmpty( i ))
      {place(i,HUMAN);
       responseValue=findCompMove( dc );
       unplace( i );  //restore board
      if(responseValue<value)//търс.намал.
       {// update best move
        value = responseValue;
        bestMove = i;
       }}
     }} return value;}
```

Най- много изчисления са нужни когато комп.започва играта(много възм.за провер-ка) Тъй като в момента се счита равенство, избира се позиция 1 (горе,ляво,макар че това е условно).Има 97,162 възмож.позиции.

Ако комп.играе втори – поз.са намалели на 5185, ако е бил избран центърът; 9761 – при избор на ъгъл; 13,233 при друг избор.

Очевидно при шах тази стратегия за прохождане (до терминален) е непосилна (> 10 ^ 100 позиции). Спира се донякъде, вика се ф-ия, оценяваща стигнатата позиция. (напр. проходимост на фигури, качество на фигури и т.н.) .

Тази ф-ия е основната за шах- програма.

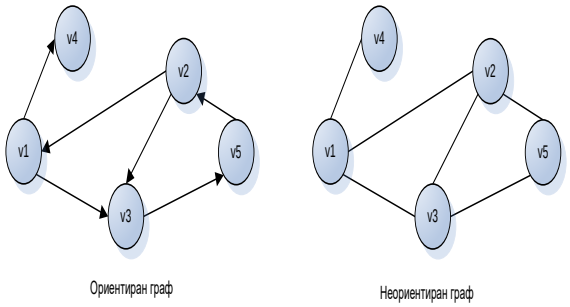
Много е важно колко хода напред могат да се прегледат (дълбочината на рекурсията). За да се увеличат:в таблица се пазят вече анализирани ходове. Когато се срещнат , все едно терминален.

β - Окастряне.

Стратегията е лесна за реализация и силна. Колкото по-нагоре в дървото я приложим , толкова ефектът е по-добър.Ограничава търсенето до O(√N) възли,където N е размера на цялото дърво, което позвол.удвояване броя проходени напред ходове.Играта крави/бикове не е подходящ пр.за ползата (има много идентични като оценка ходове). Въпреки това прилагането й в началния момент свежда от 97162 до 4493 възлите за разглеждане (само нетерминални възли се разглеждат при тази стратегия).

42.Теория на графите.Общи понятия.Представяне на граф.Топологично сортиране.

Краен ориентиран граф се нарича наредената двойка G=(V,E), където:
V={v1,v2,…,vn} е крайно множ.от върхове;
E={e1,e2,…,em} е крайно множ.от ориентирани ребра. Всеки елемент (k=1,2,…,m) е наредена двойка (vi,vj), , 1≤i,j≤n.
Ако ребрата на графа са неориентирани, графът се нарича неориентиран.
Ако в допълнение е дадена числова функция f:E→R, съпоставяща на всяко ребро ек тегло f(ek), графът се нарича претеглен. Най-често графът се представя графично с множество от точки, изобразяващи върховете му и свързващи стрелки, които са неговите ребра.

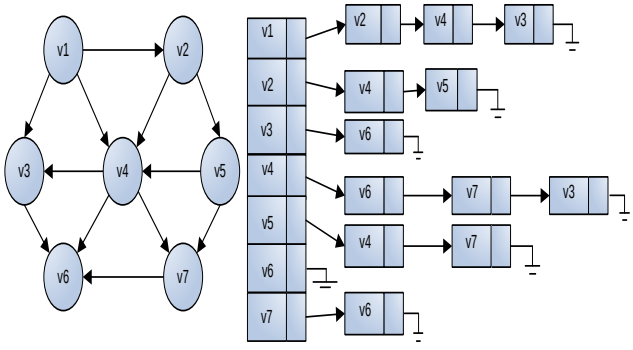


Върхът w е съседен на v тогава и само тогава, когато в неориентиран граф, ако w е съседен на v, то и v е съседен на w.
Път в граф ще наричаме последователността от върхове v1,v2,v3,…,vn, такава че , 1≤i≤n. Дължина на пътя се нарича броя на ребрата в него. Ще допускаме път от върха, до самия него, като ако няма ребра, дължината му е нула. Ако графът съдържа ребро (v,v), пътят v,v се нарича примка. Ще считаме, че графът по подразбиране няма примки.Прост път се нарича път, в който всички върхове са различни, с изключение на първия и последния. Цикъл в ориентиран граф е път с минимална дължина 1, за който v1=vn. Такъв цикъл ще наричаме прост, ако пътят е прост. За ориентирани графи ще изискваме ребрата да са различни. Причина за това е, че пътят u,v,w в неориентиран граф няма да се счита за цикъл, защото (u,v) и (v,u) са едно и също ребро. В ориентиран граф това са различни ребра и ще бъде цикъл. Един ориентиран граф е ацикличен, ако няма цикли.Един неориентиран граф е свързан, ако има път от всеки връх до всеки друг. Ориентиран граф с това свойство се нарича силно свързан. Ако съществува поне един, от двата възможни пътя, между всеки два върха, ориентираният граф е слабо свързан. Пълен граф е този, в който има ребро между всяка двойка върхове.

Предст. на граф може да се осъществи чрез:
Двумерна матрица на съседство: на всяка дъга (u,v) се съпоставя един елемент от матрицата A[u][v] със стойност 1 или теглото на графа, когато е претеглен. Липсата на дъги се означава с 0 или ∞. Ако съществува път (u,v), но не съществува (v,u) може да се запише (-1) за да се посочи ориентацията на дъгите в матрицата. При този начин на работа е необходима Θ(|V2|) памет за съхранение на матрицата. Това води до разхищение на памет, особено при граф с малко дъги. Нещата са по-добре при неориентиран граф, където матрицата е симетрична относно главния диагонал и може да се съхрани като триъгълна: само под главният диагонал. В този случай разхода на памет е

$$\Theta\left(\frac{|V|(|V|-1)}{2}\right)$$

По- ефективно представяне за разреден граф е списъка на съседство. При него за всеки връх се създава списък на съседите му. Необходимата памет тук е O(|E|+|V|). Проверката дали съществува път между два върха е по-сложна тук. Пример:

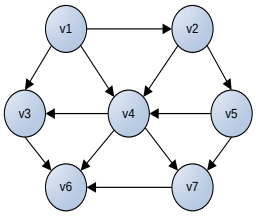


За неориентиран граф, списъкът на съседство включва две явявания на всяка дъга. При това паметта се удвоява.
Представянето облекчава задачата за намиране на съседни върхове. Тя се извършва с просто сканиране, т.е. времето е линейно. По същият начин стои и задачата за търсене на наследници. Проблем възниква при търсене на предшественици: няма връзка в списъка и трябва да се сканират всички списъци. В този случай е по-добре да се използват двусвързани списъци.Съхраняването може да бъде и в хеш-таблица, защото всеки връх на графа има уникално име и ще бъде ключа в хеша.Информацията за всеки връх ще съхраняваме в обект от тип Vertex. Тя задължително включва име на върха, което трябва да бъде уникално. Останалите параметри са опционни, в зависимост от алгоритъма, който решаваме. Всеки от представените по-нататък алгоритми е разработен с псевдокод, за да бъде представен по-ясно и да позволи реализация на различен език за програмиране. При това ще считаме, че графът е прочетен и зареден в паметта.

Топологично сортиране

В ацикличен ориентиран граф можем да номериране върховете така, че за всяко ребро (v,w), v да предшества w. Това се нарича топологично сортиране. За един граф може да имаме няколко топологични подредби. Пример за такава подредба може да бъде пътна карта. Топологично сортиране не съществува, ако в графа има цикли, защото за два върха v и w в един цикъл, v предшества w и w предшества v.
Намирането на топологична подредба може да се сведе до построяване на линеен списък Z от върховете на графа, такъв че за ,

ако , то v предхожда w в Z. Множеството от всички възможни Z е пълно топологично сортиране. Пример:



v1,v2,v5,v4,v3,v7,v6, както и v1,v2,v5,v4,v7,v3,v6 са две топологични подредби. Алгоритъмът първо открива връх без входящи ребра. После го премахва заедно със ребрата от него. Тази операция се повтаря докато не се изчерпи списъка с върховете без входящи ребра. Псевдокодът показва неговата реализация. Функция findNewVertexOfDegreeZero() сканира масива за връх с indegree=0, на който до момента не е присвоено топологично номериране. NOT_A_VERTEX се получава от функцията, ако няма такъв връх, т.е. налице е цикъл. Ф-ята заема O(|V|) време и след като се изпълява в цикъл за всички върхове, сложността ще бъде O(|V2|).
Подобрение на алгоритъма: при рядък граф са малко върховете, чиито indegree ще се променят след всяка итерация. Затова няма смисъл да обхождаме всички. Можем да запазим всички неизползвани в топологичната подредба до момента върхове с indegree=0 в отделен масив. Тогава ф-ята ще работи само с върхове от този масив. При това когато един връх стане с indegree=0 влиза в масива.
Най-добре вместо масив да се ползва стек или опашка. Всички върхове с indegree=0 се поставят в празната, за начало, опашка

```
Void Graph::topsort()
{ Vertex v,w;
For (int counter=0;
counter<NUM_VERTICES;counter++)
 { v=findNewVertexOfDegreeZero();
 if (v==NOT_A_VERTEX)
   throw CycleFound();
   v.topNum=counter;
for each w adjacent to v w.indegree--;
  } }
```

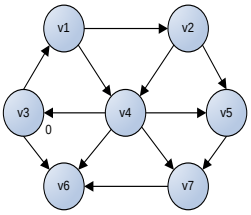
enqueue. Докато тя не се изпразни, връх се изнася от нея и за свързаните с него се намалява indegree.

```
 Void Graph::topsort()
 { Queue q(NUM_VERTICES);
  Int counter=0;
  Vertex v,w;
  q.makeEmpty();
  for each vertex v
    if (v.indegree==0)   q.enqueue(v);
  while (!q.isEmpty())
    {  v=q.dequeue();
       v.topNum= ++counter;
       for each w adjacent to v
       if (--w.indegree==0)   q.enqueue(w);
     }
   if (counter != NUM_VERTICES)
   throw CycleFound(); }
```

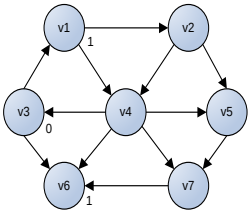
Времето за обработка е O(|E|+|V|) при използ. на списъци на съседство,т.е. лин

43.Намиране на най-къс път.Алгоритъм на Дейкастра

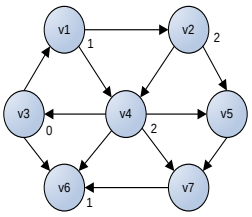
Ще разгледаме алгоритъм за намиране на най-къс път при безтегловен граф. Зададен е безтегл. граф G=(V,E) и стартов връх v, търсим най-късия път от v до всички оста-нали върхове в G. Нека разгледаме реш. за графа, показан. Приемаме за стар-тов връх v3. Тогава пътят до v3 е 0. Търсим съседните на v3.Пътят до тях е 1.Избираме един от тях и отново търсим съседи. Пътят вече е 2 и така до изчерпване на всички върхове. Това стратегия на обхождане се нарича обхождане в ширина на графа и се използва при търсене по нива.



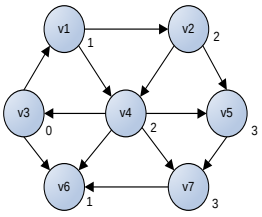
Стъпка 1



Стъпка 2



Стъпка 3

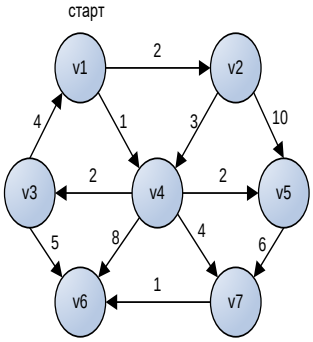


Стъпка 4

Алгоритъм на Дейкстра

Ще използваме същата идея и при граф с тегла. Отново всеки връх ще маркираме с Known=T, след като е обходен. Отново ще натрупваме разст. в dv и ще съхраняваме предшественика в pv. Тази идея за I път е реализирана от Дейкстра през 40-те год. Този алг.е обобщение на предходния и носи името алгоритъм на Дейкстра. Това е типичен пример за greedy algorithm Пр. за такава задача е какви монети трд се върнат до опр.ст-ст, за да бъде броят им мин. Проблем на такива алгоритми е, че не винаги работят. Пример за това са връщането на 15 цента в САЩ, което алгоритъма ще даде като 12+3*1, докато оптималното е 10+5.

Алг.на Дейкстра работи на етапи, като за всеки етап избира връх v с мин.дължина dv, измежду всички, неизползвани до момента и декларира най-късия път от s до v за известен.=> актуализация на ст-стите на dw според теглото на реброто (v,w), т.е. dw = dv + cv,w. Тази актуализация се прави само ако новото разст.е по-късо от същест-вуващото. В противен случай разст.не се променя. Ще разгледаме приложението на алг. за графа, показана на фиг. 6. Отново ще използваме табл.,но този път стартов връх ще бъде v1. Нека проследим отделни-те стъпки. Започваме от стартовия връх v1 и нанасяме разстояние 0 . Маркираме върха за обходен (Known=T). Определяме неговите съседи. Те са v2 и v4. Попълваме табл.за тях, след което маркираме v4 за обходен. Отново определяме съседите на v4: v3, v5, v6 и v7. Попълваме табл.за тях .Сега избираме v2. Неговите съседи са v4 и v5. От тях само v5 има Known=F.Тъй като дълж.на пътя 10+2 =12 е по-голяма от съществуващата (3), не се правят промени по таблицата.Следващият връх, който ще обработим е v5. Той има само един съсед, който не е обходен: v7. Отново няма да правим про-мени по таблицата, защото новото разсто-яние 3+6=9 е по-голямо от 5 (съществува-щото). Следва избор на v3 и корекция на разст.на v6, което става 8 . Сега идва ред на v7. Отново се променя разст.на v6,което става 6.Накрая се обхожда v6, който няма съседи и не води до промяна на разст.,и алг.спира

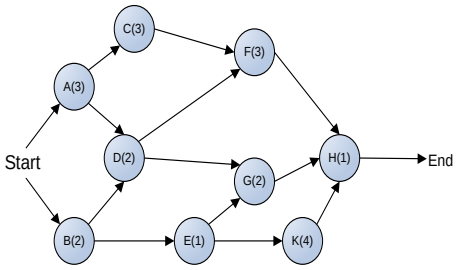


44.Алгоритъм на Дейкастра при ацикличен граф.

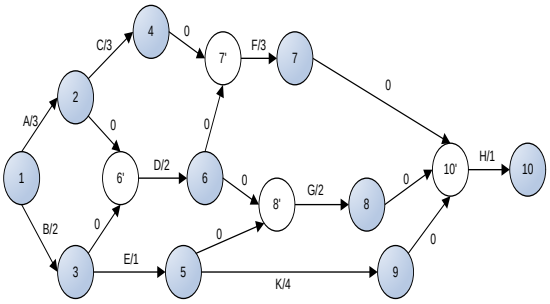
Ако знаем че графът е ацикличен можем да подобрим алг.на Дейкстра, като определим еднозначно последователността на избор на връх за обхождане. Това е последователността, съотв.на топологичната подредба. Алг.базиран на подреден топологично граф, се обработва за един пас. Това се дължи на факта, че пътят до избран според топологичната подредба връх не може вече да се намали, защото към него няма входящи ребра от необходени върхове. По този начин отпада необх.от приоритетна опашка и оценката на алг. е O(|E|+|V|).

Може да се посочат различни пр. от практиката: как да достигнем макс. бързо до дадена точка като се движим само в една посока. Друг възможен пример е моделиране на химич. реакция, в която се отделя енергия, т.е. движението е само в една посока от по-високо към по-ниско енергийно ниво. Най-важен пример за нас е оценката на дейности и анализ на критичния път.Посредством разглеждане на граф на дейностите необх. за завършване на софтуерен проект. Мд отговорим на следните въпроси:

- Кое е мин време за завърш на проекта;
- Колко мд закъсняват отделните дейности, без това да се отрази на крайния срок.



В графът на дейностите всеки връх пред-ставя дейност, която трд се изпълни като е посочено и времето, което заема тя.Ребрата показват реда на изпълн. на дейностите. Графът е ацикличен и освен това допуска работа в паралелизъм за независещи една от друга дейности.За да решим поставената зад.трд преобразуваме графа на дейностите в граф на събитията.При това всяко съби-тие отговаря на завършване на дейност и всички зависещи от нея дейности. За да се деф.еднозначно зависимостите м/у отдел-ните дейности, добавяме фиктивни(празни) възли, когато една дейност зависи от завършването на няколко други.

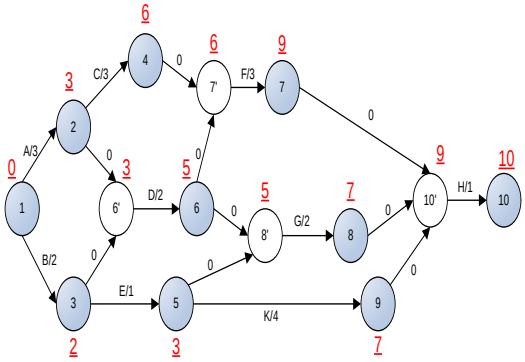


Търсим най-дългият път до края за да определим EC (earliest completion time) най-краткото време за завършване на проект. При това ще адаптираме алг. за намиране на най-къс път. Ако ECi е най-късото време за изпълнение на върха I, то изчисл.става посредством:

$$EC_1 = 0$$

$$EC_w = \max_{(v,w)\in E}(EC_v + c_{v,w})$$

след прилагане на формулите:

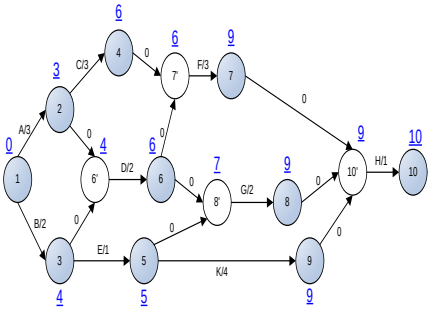


Мд определим и времето LCi (latest completion for node i), до което мд забавим даден етап, без да отлагаме крайния срок:

$$LC_n = EC_n$$

$$LC_v = \min_{(v,w)\in E}(LC_w - c_{v,w})$$

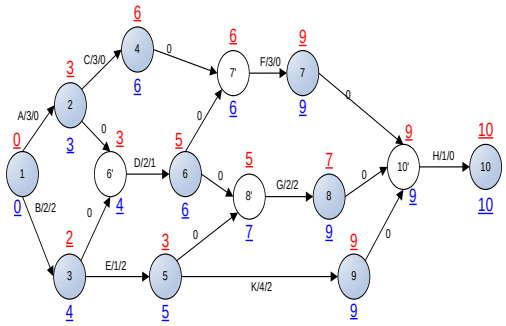
След резултатите от това изчисление:



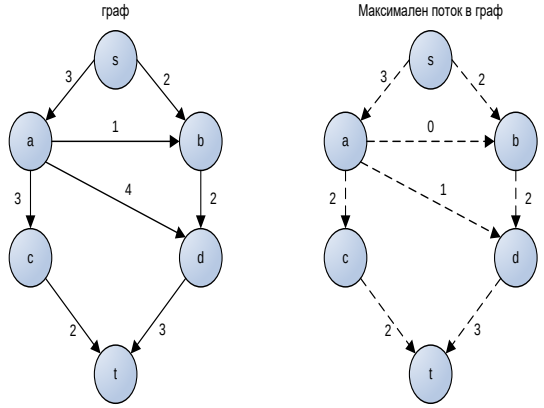
Тези ст-сти могат да се изчислят за лин. време за всеки връх като се използва топологично сорт.за ECi и обратно топологично сорт.за LCi. Мд изчислим и луфта (slack) във време, с който разполагаме. Той показва макс.закъснение за всеки връх, без да се просрочи крайния срок:

$$slack_{(v,w)} = LC_w - EC_w - c_{v,w}$$

Ето пример като горната почертана цифра показва ECw, а долната – LCw. Означени-ята по ребрата са: дейност/продължителност/луфт. Както се вижда, някои дейности нямат закъсн.Те са критични.Път,който съдържа такива дейности се нарича критичен.



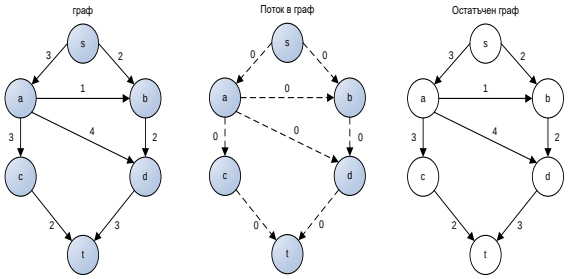
45.Пропускателна способност на мрежа



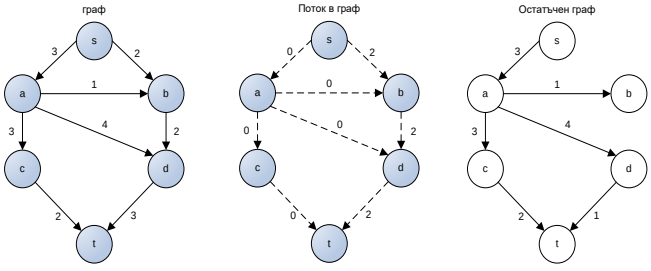
т.н. Дадени са два върха s и t. Търсим макс.поток,който мд мине от s до t.

Всяко ребро мд поеме поток равен на капа-цитета си. Означенията в макс поток в граф са, както следва: връх a има входен поток 3 ед-ци, които се разпределят към c и d. Връх d също има входен капацитет 3 единици, които получава от a и b и ги предава на t.

За построим макс. поток в граф, работим поетапно. На основа на графа G, строим граф на потоците(flow graph) Gf, съдържащ потоците определени до момента. В нач. всички ребра имат нулев поток.При прик-лючване на алг. Gf ще съдържа макс.поток. При това строим и граф с остатъч. потоци Gr (residual graph), съдържащ неизполз. капацитети по всяко ребро (разликата от капацитета и използ.количество).Търсим път от s към t в Gr. Мин.капацитет на реб-ро по този път е възможния за добавяне в поток. Добавяме го в Gf. Продължаваме по този начин, докато не се изчерпят всички пътища в Gr. Алг.е недетерминиран, защо-то няма стратегия за последователността на избор на възможните пътища.

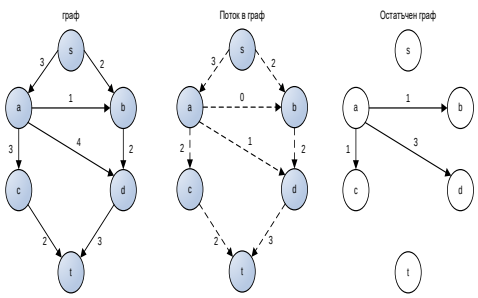


Даден е ориентиран граф с указан капаци-тет на всяко ребро $c_{v,w}$. Този капацитет мд показва пропускателна способност на тръ-ба,на улица и Избираме един от възможните пътища: s-b-d-t. Мин.поток тук е 2 единици.Когато реб-ровият капацитет се запълни, премахваме ребро от графа Gr.

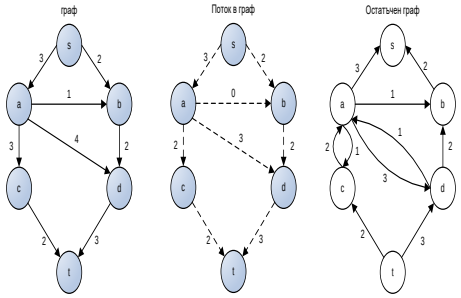


=>избор на нов път.Той е s-a-c-t с мин. поток 2 единици

=>пътя s-a-d-t с мин.поток 1. Тази единица се добавя към теглата в Gf. Алг.приключва, защото няма повече пътища в Gr.



Накрая сумираме входните капацитети в Gf за върха t и получаваме макс.поток. В случая той е 5 единици. Сега ще покажем недетерминираността на алг. Ако в нач. тръгнем по пътя s-a-d-t, ще допуснем поток от 3 единици. Това е добър резултат, но в Gr не остава път от s към t (фиг. 25). Това е пример как greedy алг.не винаги работят Разсъждавайки по този начин, налице е нов път s-b-d-a-c-t, който има поток 2 единици. Това означава, че щом връщаме от d към a поток, то по тази тръба могат да минат същите единици и в обратна посока, от d към a, стига да има наличен път

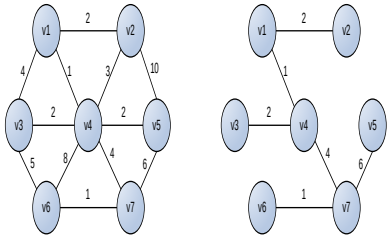


Отново макс. поток е 5 единици, но сме елиминирали възможността грешен нач. избор на път да повлияе върху решението. Доказано е, че така модифицирам алг., винаги открива максималния поток. Нещо повече, алг.работи и при цикличен граф.

Накрая ще направим оценка на алг. Ако капацитетите са цели числа, всяка итера-ция увеличава потока поне с една единица. Ако макс.поток е f,то f етапа са достатъчни за намирането му и времето е O(f*|E|). Това е така, защото път мдб намерен за O(|E|) време, при използване на алг. за намиране на най-къс път в безтегловен граф (unweighted shortest path). За да нама-лим времето, ще избираме винаги пътя с макс поток, при което ще приложим алг.на Дейкстра. Ако cap_{max} е макс.капацитет на дадено ребро, то O(|E|log cap_{max}) итера-ции са необх.за получаване на макс.поток. Знаем, че времето за една итерация е O(|E|log|V|) и => общото време ще е O(|E|^{2}log|V|log cap_{max}). Ако работим с много малки капацитети, едно добро прибл.се дава с времето O(|E|^{2}log|V|).

Друг начин за избор на текущ път е пътя с мин.брой ребра. В този случай етапите са O(|E|*|V|) и след като всеки етап изисква O(|E|)време(отново използ.алг.за най-къс път),получаваме O(|E|^2|V|) за общото време.За по-нататъшно подобрение на времето, трд се смени структурата данни. Възможно е подобрение на времето и при налагане на някои ограничения,напр. за граф, в който всеки връх има само едно входящо или едно изходящо ребро (без s и t, разбира се) с капацитет 1. В този случай времето е O(|E||V|$^{1/2}$).

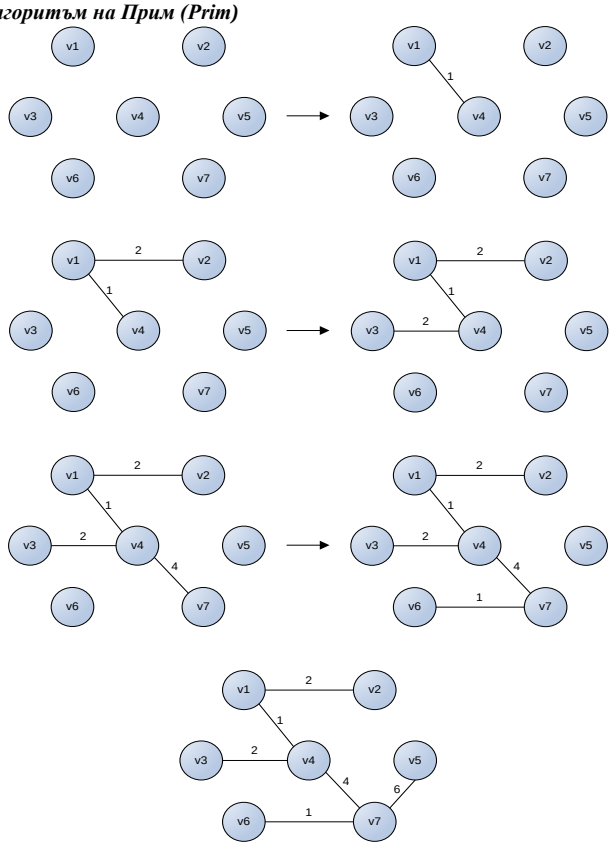
46.Минимално обхващащо дърво.Алгоритъм на Прим.Алгоритъм на Крускал.



Ще търсим мин.обхващащо дърво в неори-ентиран граф. Зад.има смисъл и за ориен-тиран граф, но там алг.е по-труден. За да-ден граф G, мин.обхващащо дърво е дърво-то, изградено от дъги на графа, които свър-зват върховете му на мин.цена. Мин.обхва-щащо дърво съществува само,ако графа е свързан. Ще работим при това ограниче-ние. Пр. окабеляване на жилища.

минималното обхващащо дърво има |V|-1 ребра. За всяко мин.обхващащо дърво T, ако ребро (v,w), което не съществува в T се добави, се създава цикъл. Изтриването на всяко ребро от цикъла, възстановява мин. обхващащо дърво. Цената на мин.обхва-щащо дърво е толкова по-ниска, колкото по по-ниска е цената на изграждащите го ребра, следователно при изграждането на мин.обхващащо дърво трябва да добавяме ребро с мин.стойност. Ще разгледаме два алг.,които се различават по избора на ребро с минимална стойност

Алгоритъм на Прим (Prim)



Работим на етапи. На всеки етап взимаме един връх v от дървото за корен, добавяме друг w от невключените до момента и асо-цираме ребро в дървото. Измежду всички ребра (v,w), за които v е от дървото, а w не е, избираме това, което има мин.цена. На фиг.е показано как работи алг.с начало v1. В нач.момент v1 е дървото и на всеки етап добавяме по един връх и едно ребро

Алг.е подобен на този на Дейкстра.Както и преди ще поддържаме d_v, p_v и Known. d_v е мин.тегло на реброто (v,w), където w е пре-дшественика на v с Known=T. p_v съдържа последния връх w,който е променил d_v.Ос-таналата част от алг.е същата. Единствено се променя актуализацията на d_v: след избор на връх v, за всички съседи w с Known=F на v, d_w=min(d_w,$c_{v,w}$).

Нач.на алг. е показано на фиг. 30.На фиг. 31 е избран върха v1, а за v2, v3 и v4 са актуализирани d_v и p_v. Следващият избран връх е v4. Всички останали върхове са му съседи. V1 се изключва от разглеждането, защото има Known=T, информацията за v2 не се променя, защото d_v =2, а стойността на реброто (v4, v2) е 3. Всички останали върхове се актуализират (фиг. 32).

Следващият избран връх е v2. Неговият избор не води до промяна на d_v. По-нататък избираме v3. При това се променя d_v на v6, както е показано на фиг. 33. Фиг. 34 показва резултата след избор на v7, при което се променя d_v за v6 и v5. След избор на v6 и v5, алг.приключва своята работа. Крайният резултат е показан на фиг. 35. Ребрата на мин. обхващащо дърво се получават от таблицата и са както следва: (v2, v1), (v3, v4), (v4, v1), (v5, v7), (v6, v7), (v7, v4). Общата цена е 16

V	Фиг.30 K d p_v	Фиг.31 K d p
V1	F 0 0	F 0 0

V2	F ∞ 0	F ∞ 0
V3	F ∞ 0	F ∞ 0

V4	F ∞ 0	F ∞ 0
V5	F ∞ 0	F ∞ 0

V	F ∞ 0	F ∞ 0
V6		
V7	F ∞ 0	F ∞ 0

V	Фиг.32			Фиг.33		
	K	d v	p_v	K	d v	p v
V1	T	0	0	T	0	0
V2	F	2	V 1	T	2	V 1
V3	F	2	V 4	T	2	V 4
V4	T	1	V 1	T	1	V 1
V5	F	7	V 4	F	7	V 4

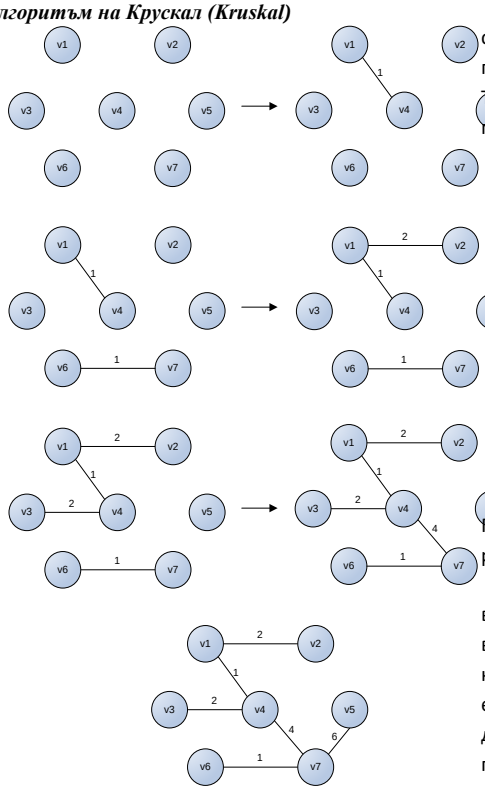
V	F 8 V	F 5 V
V6	4	3
V7	F 4 V 4	F 4 V 4

V	Фиг.34			Фиг.35		
	K	d v	p_v	K	d v	p v
V1	T	0	0	T	0	0
V2	T	2	V 1	T	2	V 1
V3	T	2	V 4	T	2	V 4
V4	T	1	V 1	T	1	V 1
V5	F	6	V 7	T	6	V 7

V	F 1 V	T 1 V
V6	7	7
V7	T 4 V 4	T 4 V 4

Реализацията на този алг.е идентична на Дейкстра. Аналогично е и положението с анализа. Трябва да се внимава при реализацията на алг.при неоринтирани графи. Тогава всяко ребро трябва да участва два пъти в списъка на съседите. Времето е $O(|V|^2)$ без използване на heap, което е оптимал.за плътни графи и $O(|E|\log|V|)$ при използване на двоичен heap – добро решение за разредени графи.

Алгоритъм на Крускал (Kruskal)



Стратегията тук е да се избере ребро с най-малко тегло и да се одобри, ако не формира цикъл. За графа от предходния пример действията са показани на фиг. 36. При това се попълва таблица, в която за всяко ребро се записва неговото тегло и флаг, който показва дали върха е отхвърлен или не. Табл. за графа от предходния пример е показана на фиг. 37.

Ребро	Тегло	Флаг
(v1,v4)	1	Включено
(v6,v7)	1	Включено
(v1,v2)	2	Включено
(v3,v4)	2	Включено
(v2,v4)	3	Отхвърлено
(v1,v3)	4	Отхвърлено
(v4,v7)	4	Включено
(v3,v6)	5	Отхвърлено
(v5,v7)	6	Включено

Погледнато формално алг. на Крускал работи с множество дървета. В нач. това са $|V|$ броя дървета, състоящи се от по един връх. Добавяйки ребро,сливаме 2 дървета в едно. Алг.приключва когато са включени необх.брой ребра. При това е налице само едно дърво и то е търсеното. При добавяне-то на ребро се използва следното правило: ако v и w са два върха в едно дърво, ребро-то, което ги свързва, не се избира за разгле-ждане,защото формира цикъл.В противен случай реброто се включва и двете дървета (включващи v и w) се сливат. Вижда се, че множествата са инвариантни (върховете в тях стоят постоянно) и към тях само мд се добавят нови чрез сливане на дървета. Реб-рата могат да се сортират за ускоряване на търсенето, но изграждането на heap е по-добра идея за ускоряване.

```
Void Graph::kruskal()
{ int edgeAccepted;
  DisjSet s(NUM_VERTICES);
  PriorityQueue h(NUM_EDGES);
  Vertex u,v;
  SetType uset, vset;
  Edge e;
  h = readGraphIntoHeapArray();
  h.buildHeap();
  edgeAccepted=0;
  while (edgeAccepted<NUM_VERTICES-1)
  { h.deleteMin(e);
    uset = s.find(u);
    vset = s.find(v);
    if (uset!=vset)
   { edgesAccepted++;
     s.unionSets(uset,vset);
    }
  }
}
```

Оценката на алг. е $O(|E|\log|E|)$ в най-лошия случай, което се определя от heap операциите. Отбележете, че тъй като $|E| = O(|V|2)$, то времето за изпълнение ще бъде $O(|E|\log|V$