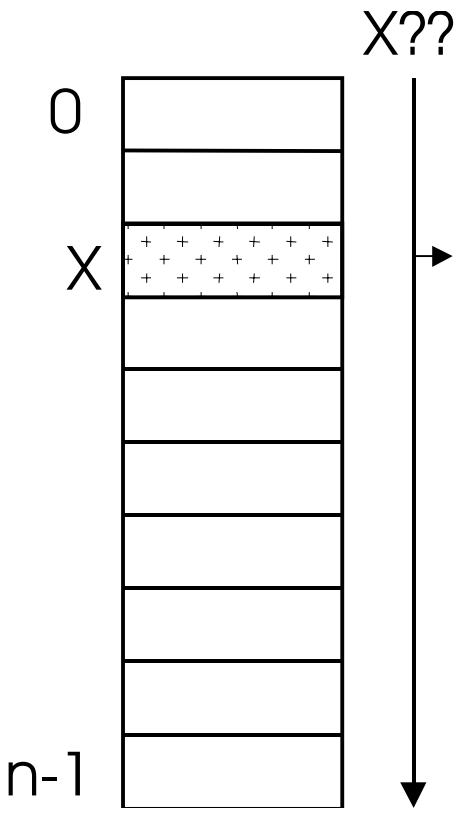
2 Haku ja lajittelu

Tässä luvussa käsitellään tiedon hakemista taulukosta sekä taulukon lajittelua. Useista erilaisista lajittelumenetelmistä on kirjaan otettu suurehko osa. Lajittelumenetelmien tehokkuutta kannattaa vertailla eri suuruisilla taulukoilla: huomattavia eroja lajitteluajassa alkaa syntyä, kun alkioiden määrä on kymmeniä tuhansia. Lajittelumenetelmiä oppii ymmärtämään paremmin, jos kynää ja paperia käyttäen simuloi lajittelua pienehköllä taulukolla. Luvun lopussa on kokeiltu **stl** (standard template library) -kirjaston sort()-funktiota, joka osoittautuukin erittäin tehokkaaksi.

2.1 Luvun hakeminen taulukosta

Jonkin tietyn luvun hakemiseen taulukosta on käytettävissä useitakin menetelmiä. Tässä esitellään kahta yleistä tapaa, nimittäin *peräkkäishaku* ja *binäärihaku*.

Peräkkäishaussa alkiot voivat olla epämääräisessä järjestyksessä. Menettely on yksinkertainen, mutta suurilla alkiomäärillä helposti tehoton. Peräkkäishaussa taulukko käydään läpi alusta alkaen järjestyksessä. Kuvamme esittää tilannetta, jossa taulukosta (tai listasta) haetaan arvoa x.



Seuraavassa lineaarihaku-ohjelmassa haetaan oliotaulukosta henkilöä, jonka henkilökoodi on annettu:

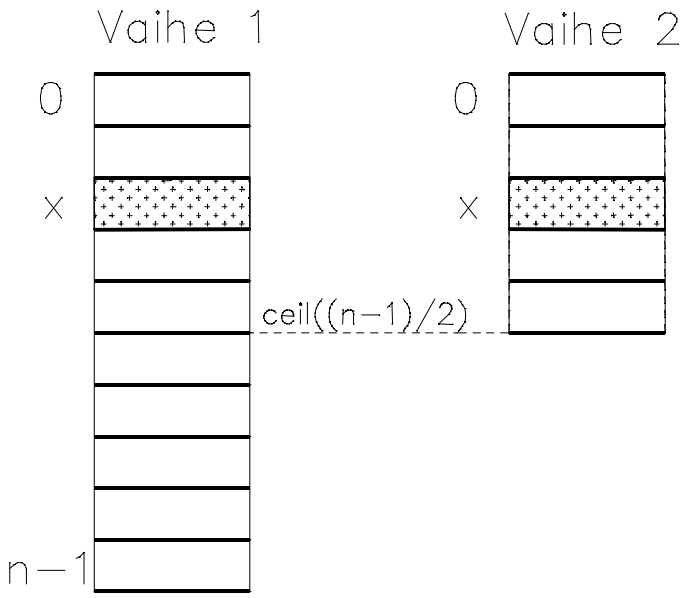
Lineaarihaku oliotaulukosta:

```
#include <iostream.h>
#include <stdlib.h>
int rand(void);

class HENKILO
{
public:
HENKILO() { hlokoodi = 0; palkka=5; }
~HENKILO() {}
int HaeKoodi() const { return hlokoodi; }
int HaePalkka() const { return palkka; }
void AsetaKoodi(int koodi) { hlokoodi = koodi; }
friend main();
private:
int hlokoodi;
int palkka;
};

int linhaku(HENKILO Hlosto[], int koodi);

int x;

int main()
{
HENKILO Hlosto[10]; // Taulukkoon 10 HENKILO-oliota
int i;

for (i = 0; i < 10; i++)
    Hlosto[i].AsetaKoodi(rand() % 9);

for (i = 0; i < 10; i++)
      cout << "Hlon #" << i+1<< ": " << Hlosto[i].HaeKoodi() <<
endl;

cout << "Anna haettava koodi: \n";
cin >> x;
int p = linhaku(Hlosto, x);
if (p != 999) cout << "Löytyi kohdasta " << p ;
else cout << "Arvoa ei ole ";

return 0;
}
```

```
int linhaku(HENKILO Hlosto[], int koodi)
{
int i;
for (i=0; i<10; i++)
    {
    if (Hlosto[i].HaeKoodi() == koodi)
        return (i);
    }
    return 999;
}
```

Suurille taulukoille soveltuu *binäärihaku* peräkkäishakua paremmin. Tällöin taulukon tulee kuitenkin olla valmiiksi lajiteltu suuruusjärjestykseen. Menettelyssä katsotaan ensin, onko haettava luku taulukon ylemmässä vai alemmassa puolikkaassa. Jos haettu alkio on esimerkiksi ylemmässä taulukon osassa, alempi osa jätetään huomioimatta ja sen sijaan katsotaan taas ylemmästä osasta, kummassa puoliskossa haettu arvo on siinä osataulukossa. Tätä menettelyä jatketaan, kunnes arvo on löydetty.

Kuvassa havainnollistetaan menettelyä:



Seuraavana on binäärihaun algoritmi:

Binäärihaku:

```
Annetaan haettava arvo
Puolitetaan taulukko loogisesti
Verrataan, onko arvo < kuin yläosan viimeinen alkio
    Jos <, niin arvo löytyy ylemmästä osasta,
    muutoin alemmasta osasta
(Arvo voi 'vahingossa löytyä suoraan, joten mahdollisuus
huomioitava)
Jatketaan, kunnes arvo löytyy
```

Seuraavana on kokonainen ohjelma, jossa lajitellaan taulukko ensin quicksort-menetelmällä ja haetaan taulukosta arvoa:

```
#include <iostream.h>
#include <conio.h>

const int TAULUKOKO= 100;

int luvut[TAULUKOKO];

void vaihda(int *x,int *y)
{
int temp;
temp = *x;
*x = *y;
*y = temp;
}

void tulosta( int luvut[])
{
int i;
for (i=0; i<TAULUKOKO; i++)
cout << luvut[i] << "\n";
}
// QuickSort-lajittelu
void lajittele(int eka, int viime,int luvut[])
{
int alku, vasen, oikea, temp;
vasen = eka;
oikea = viime;
alku = luvut[(eka + viime) / 2];
do
{
while (luvut[vasen] < alku)
```

```
vasen = vasen +1;
while (alku < luvut[oikea])
oikea = oikea - 1;
if (vasen <= oikea)
{
vaihda (&(luvut[vasen]), &(luvut[oikea]));
vasen = vasen + 1;
oikea = oikea - 1;
}
}
while ((oikea > vasen));
if (eka < oikea) lajittele(eka, oikea, luvut);
if (vasen < viime) lajittele(vasen, viime, luvut);
}


int binhaku(int luku, int luvut[])
{
int mediaanialkio;
int mediaani;
int eka = 0;
int viime = TAULUKOKO -1;
do
{
mediaani = (viime + eka) / 2;
mediaanialkio = luvut[mediaani];
if (luku < mediaanialkio)
viime = mediaani-1;
else
eka = mediaani +1;
} while (!(luku == mediaanialkio || eka > viime));
return (luku == mediaanialkio) ? mediaani : -1;
}


void main()
{
int s;
for (s = 0; s < TAULUKOKO; s++)
luvut[s] = rand() % 100;

lajittele(0,TAULUKOKO,luvut);

cout << "HAETAAN ARVO, nyt arvona luku 90:  \n\n";
int L = binhaku(90, luvut);
cout << "Löytyy kohdasta" <<  L << "\n";

}
```

Haettaessa *tiedoston tietueita* käytetään yleisesti hyväksi indeksoimista. Indeksoinnissa ei tiedostoa lajitella fyysisesti, vaan muodostetaan osoitetaulukko. *Indeksihaussa* käytetään aputaulukoita, jotka sisältävät osoitteita tiedoston tietueisiin tai tietueryhmiin. Näin päästään (joskus suurtenkin) tiedostojen käsittelyssä vähemmälle, jolloin haku on nopeampaa ja tietokoneen kuormitus vähäisempää.

Seuraavana on binäärihakuohjelma, jossa taulukkona on oliotaulukko. Binäärihaku-funktio on luokan ystävä.

```
#include <iostream.h>

class KANA
{
public:
KANA() { ika = 1; paino=5; }
~KANA();
int HaeIka() const { return ika; }
int HaePaino() const { return paino; }
void AsetaIka(int i) { ika = i; }
friend int binhaku(int luku, KANA oliotaulu[]);

private:
int ika;
int paino;
};

int binhaku(int luku, KANA oliotaulu[]);

KANA :: ~ KANA ()
{
}

int main()
{
KANA * kanala = new KANA[10];
int i;
KANA * pKana;

for (i = 0; i < 10; i++)
{
  pKana = new KANA;
  pKana->AsetaIka(2*i +1);
  kanala[i] = *pKana;
  delete pKana;
}

int  hakuarvo = 3;
```

```
int paikka = binhaku(hakuarvo, kanala);
cout << "Haettava jäsenmuuttujan arvo on oliossa \n";
cout << "joka sijaitsee kohdassa " << paikka << "\n";

delete [] kanala; // taulukko tuhotaan
return 0;
}

int binhaku(int luku, KANA oliotaulu[])
{
KANA mediaanialkio;
int mediaani;
int eka = 0;
int viime = 9;
do
{
mediaani = (viime + eka) / 2;
mediaanialkio.ika = oliotaulu[mediaani].HaeIka();
if (luku < mediaanialkio.HaeIka())
viime = mediaani-1;
else
eka = mediaani +1;
} while (!(luku == mediaanialkio.HaeIka() || eka > viime));
return (luku == mediaanialkio.HaeIka()) ? mediaani : -1;
}
```

2.2 Taulukoiden lajittelu

Taulukoiden lajittelu voidaan tehdä useallakin eri tavalla. Sopivin tapa riippuu esimerkiksi vaaditusta lajittelunopeudesta. Erityisesti suurten taulukoiden kohdalla nopeus on tärkeä tekijä. Seuraavassa esitellään muutama lajittelualgoritmi. Voit itse kokeilla algoritmeja ja lisätä niihin ajanlaskun, jolloin voit vertailla myös lajittelunopeuksia. Muistanet käyttää vertailuajoissa samaa taulukkoa.

Useimmissa lajittelualgoritmeissa tarvitaan lukujen vaihtamisproseduuria, jonka algoritmi voi olla esimerkiksi seuraavanlainen:

Arvojen keskinäinen vaihtaminen:

```
void vaihda (float *arvo1, float *arvo2)  // osoitimet
{
float temp
temp = *arvo1;
```

```
*arvo1 = *arvo2;
*arvo2 = temp;
}
```

Tai viittauksia käytettäessä:

```
void vaihda (int &rx, int &ry)     // viittaukset
{
int temp;
temp = rx;
rx = ry;
ry = temp;
}
```

2.2.1 Vaihtolajittelu

Vaihtolajittelu on yksinkertaisin lajittelumenetelmä, jonka vuoksi sitä myös käytetään eniten. Voit itse kokeilemalla verrata tämän menetelmän tehokkuutta muihin menetelmiin. Vaihtolajittelussa alkiot asetetaan suuruusjärjestykseen alkuperäiseen taulukkoon, mutta mikään ei myöskään estä käyttämästä toista taulukkoa, johon alkiot asetetaan suuruusjärjestyksessä.

Menetelmässä verrataan vierekkäisiä alkioita keskenään. Useimmiten aloitetaan taulukon ensimmäisestä alkiosta, jota aletaan verrata muihin alkioihin. Jos vertailun kohdealkio on pienempi, se vaihdetaan ensimmäiseksi ja jatketaan edelleen vertailua, kunnes on tultu taulukon loppuun. Sen jälkeen siirrytään vertailemaan taulukossa toisena olevaa alkiota muihin edellä oleviin alkioihin. Näin jatketaan, kunnes tullaan taulukon loppuun, jolloin taulukko on lajiteltu.

Simuloimme seuraavassa vaihtolajittelua. Muistanet, että taulukon ensimmäisen alkion indeksi on nolla. Lajittelemme esimerkissämme taulukon kasvavaan järjestykseen.

Oletetaan, että meillä on seuraava taulukko.

2	4	8	9	3	1	7	5	11	6
alkio0	alkio1	alkio2	alkio3	alkio4	alkio5	alkio6	alkio7	alkio8	alkio9

Vaihtamislajittelu tapahtuu seuraavasti:

2	4	8	9	3	1	7	5	11	6

Alkiota nolla verrataan ensin kaikkiin muihin alkioihin.

2>4 Ei ole, verrataan seuraavaan alkioon
2>8 Ei ole, verrataan seuraavaan alkioon
2>9 Ei ole, verrataan seuraavaan alkioon
2>3 Ei ole, verrataan seuraavaan alkioon
2>1 On, vaihdetaan alkioiden arvot keskenään
1>7 Ei ole, verrataan seuraavaan alkioon
1>5 Ei ole, verrataan seuraavaan alkioon
1>11 Ei ole, verrataan seuraavaan alkioon
1>6 Ei ole, verrataan seuraavaan alkioon

Ensimmäisen kierroksen tuloksena on seuraava taulukko:

1	4	8	9	3	2	7	5	11	6

Toisella kierroksella verrataan alkiota yksi kaikkiin sen edellä oleviin alkioihin.
4>8 Ei ole, verrataan seuraavaan alkioon
4>9 Ei ole, verrataan seuraavaan alkioon
4>3 On, vaihdetaan alkioiden arvot keskenään
3>2 On, vaihdetaan alkioiden arvot keskenään
2>7 Ei ole, verrataan seuraavaan alkioon
2>5 Ei ole, verrataan seuraavaan alkioon
2>11 Ei ole, verrataan seuraavaan alkioon
2>6 Ei ole, verrataan seuraavaan alkioon

Toisen kierroksen jälkeen taulukko näyttää seuraavalta:

1	2	8	9	4	3	7	5	11	6

Kolmannen kierroksen jälkeinen tulos:

1	2	8	9	4	3	7	5	11	6

Ja niin edelleen, kunnes koko taulukko on lajiteltu pienemmästä suurimpaan.

Lopputuloksena on siis lajiteltu taulukkko:

1	2	3	4	5	6	7	8	9	11

Algoritmi ja ohjelma:

Vaihtolajittelu:

```
#include <iostream.h>

int luvut[6];
int maara;

void lue(int luvut[])
{
int i;
cout << "Anna alkiot \n";
for (i = 0; i <6; i++)
{
cout << "Anna luettelon " << i << ". alkio\n";
cin >> luvut[i];
}
}

void vaihda(int* a, int* b)
{
int temp;
temp = *a;
*a = *b;
*b = temp;
}



void tulosta(int luvut[])
{
int i;
cout << "Onko jarjestyksessa: \n";
cout << "Luettelon alkiot: \n";
for (i = 0; i <6; i++)
cout << "Luettelon " << i << ". alkio on: " << luvut[i] << "\n";
}

void lajittele(int luvut[], int maara)
{
int k,l;
for (k=0; k<maara-1;k++)
```

```
    for (l=k+1; l<maara; l++)
        if (luvut[k] > luvut[l])
            vaihda (&luvut[k], &luvut[l]);
}


main()
{
lue(luvut);
lajittele(luvut, 6);
tulosta(luvut);
return 0;
}
```

Edellisen algoritmin vaihda()-funktio on kuvattu aiemmin.

2.2.2 Quicksort

Quicksort on nimensä mukaisesti usein paras lajittelumenetelmä. Suoritusaika pahimmassa tapauksessa $O(n^2)$. Ongelma jaetaan osaongelmiin ja kokonaisratkaisu muodostuu osaongelmien ratkaisuista.

Oleellista on se, kuinka jako kahteen tapahtuu. Ideana on valita lajiteltavasta joukosta arvo v, joka jakaa joukon kahtia siten, että toiseen joukkoon tulevat v:tä pienemmät ja toiseen v:tä suuremmat alkiot. Sitten molemmat osajoukot lajitellaan erikseen.

Helpoin tapa olisi valita joukon jakaja-alkioksi ensimmäinen alkio. Tällöin kuitenkin lajittelunopeus on pahin mahdollinen.

Periaatteessa paras tapa on valita v:ksi joukon mediaani, mutta vain periaatteessa. Käytännössä on havaittu, että v:ksi kannattaa valita kolmen alkion mediaani, eli jos lajiteltavat alkiot ovat a[i], ... ,a[j], niin v:ksi valitaan alkioiden a[i], a[ceil(i+j)/2], a[j] mediaani (ensimmäisen, viimeisen ja keskellä olevan alkion mediaani). Muitakin tapoja valita v on olemassa.

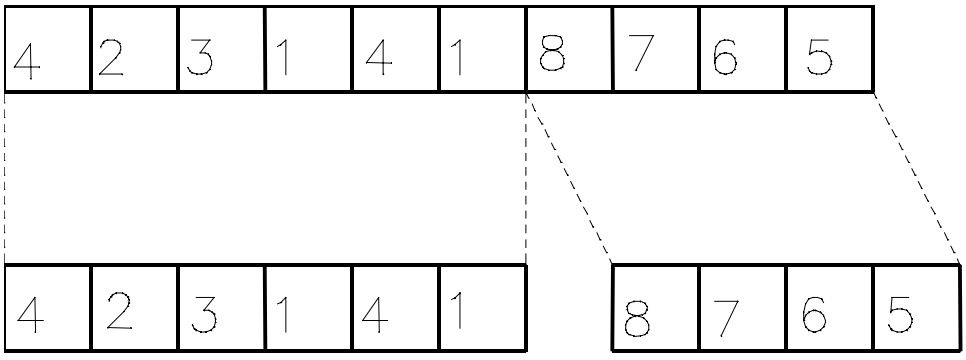
Manuaalinen simulointi antaa parhaimman käsityksen Quicksort-menetelmästä. Oletetaan, että meillä on seuraavanlainen taulukko, jossa on 10 alkiota:

1	2	3	4	5	6	7	8	9	10
4	2	3	1	5	6	8	7	1	9

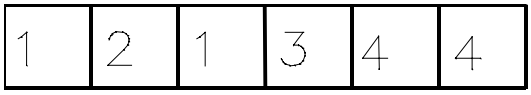
a[1] = 4

a[10] = 9
a[5] = 5

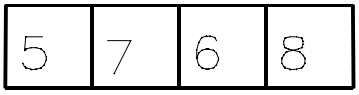
Joukon 4, 5, 9 mediaani on **5**, joka valitaan pivot-alkioksi eli alkioksi, jonka mukaan taulukko jaetaan ensimmäisellä tasolla. Jaon jälkeen muodostuu kaksi osajoukkoa:



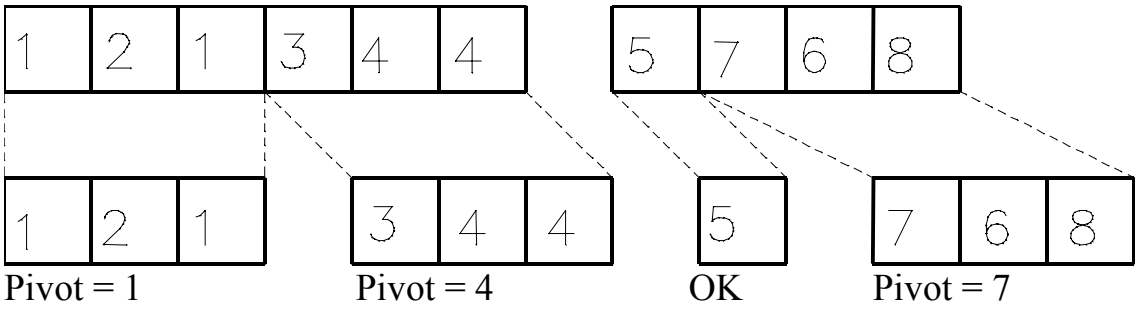
Pivot on 3



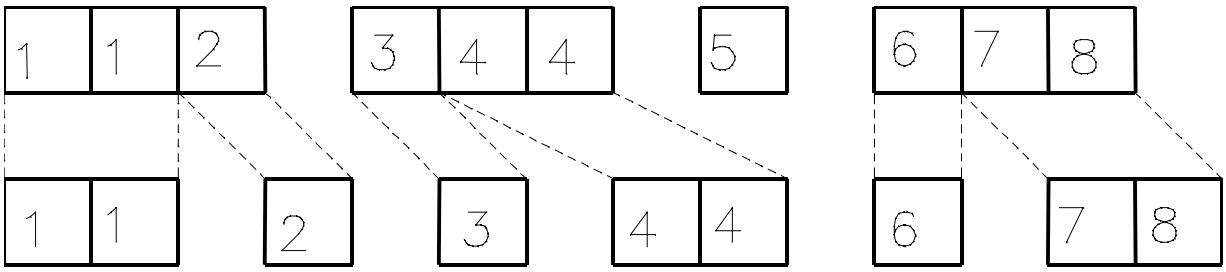
Pivot on 6



Saadaan uudet osajoukot:



Saadaan lopputulos:



Quicksort-lajittelu:

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

int luvut[1000];

void vaihda(int *x,int *y)
{
int temp;
temp = *x;
*x = *y;
*y = temp;
}

void tulosta( int luvut[])
{
int i;
for (i=0; i<1000; i++)
cout << luvut[i] << "\t";
getche();
}

void lajittele(int eka, int viime,int luvut[])
```

```
{
int alku, vasen, oikea, temp;
vasen = eka;
oikea = viime;
alku = luvut[(eka + viime) / 2];
do
{
while (luvut[vasen] < alku)
vasen = vasen +1;
while (alku < luvut[oikea])
oikea = oikea - 1;
if (vasen <= oikea)
{
vaihda (&(luvut[vasen]), &(luvut[oikea]));
vasen = vasen + 1;
oikea = oikea - 1;
}
}
while ((oikea > vasen));
if (eka < oikea) lajittele(eka, oikea, luvut);
if (vasen < viime) lajittele(vasen, viime, luvut);
}


void main()
{
int s;
for (s = 0; s < 1000; s++)
luvut[s] = rand() % 99;

cout << "Lajittelematon taulukko:  \n\n";
tulosta(luvut);

for (s = 0; s < 5; s++)
lajittele(0,999,luvut);

cout << "Lajiteltu taulukko:  \n\n";
tulosta(luvut);
}
```

2.2.3 Shell-lajittelu

Donald Shell kehitti tämän algoritmin vuonna 1959. Algoritmi on vaihtolajittelua nopeampi, mutta samalla monimutkaisempi. Algoritmin perusideana on vertailla jonkin matkan päässä toisistaan olevia alkioita keskenään. Alkioiden välinen etäisyys taulukossa on aluksi melko suuri (puolet alkioiden lukumäärästä) ja ensimmäisen kierroksen jälkeen välimatka yleensä puolitetaan. Eli, jos alkioiden lukumäärä on esimerkiksi 32, niin ensimmäinen vertailu suoritetaan taulukossa 1. ja 17. alkion välillä. Tämän jälkeen verrataan 2. ja 18. alkion välillä ja niin edelleen. Toisella kierroksella alkioiden välinen etäisyys on 16/2 eli kahdeksan. Lopulta tullaan siihen, että välimatka on yksi, eli vertailu suoritetaan kaikkien alkioiden kesken, kuten vaihtoalgoritmissa. Nyt kuitenkin vaihtamisten määrä on minimaalinen, joka osin selittää algoritmin tehokkuuden.

Seuraavana havainnollistetaan Shell-lajittelua esimerkkitaulukon avulla:

2	5	9	1	3	8	4	6

Taulukossa on siis 8 alkiota, eli vertailuetäisyydeksi otetaan nyt aluksi 4 alkiota.

Alkiota nolla verrataan ensin alkioon neljä, eli lukua 2 lukuun 3. Sen jälkeen verrataan lukua viisi lukuun 8 ja tämän jälkeen lukua 9 lukuun 4. Tässä viimeisessä vertailussa tehdään vaihto-operaatio, koska 4 on pienempi kuin 9. Lopuksi verrataan lukua 1 lukuun 6.

Ensimmäisen kierroksen jälkeen näyttää taulukko siis tältä:

2	5	4	1	3	8	9	6

Toisella kierroksella alkioiden välimatka on puolet edellisestä eli 2. Proseduuri jatkuu tämän jälkeen kuten ensimmäisellä kierroksella.

Toisen kierroksen jälkeen taulukko on seuraavassa järjestyksessä:

2	1	3	5	4	6	9	8

Viimeisellä eli kolmannella kierroksella on välimatka yksi ja taulukko asettuu lopullisesti suuruusjärjestykseen.

Algoritmi ja ohjelma:

Shell-lajittelu

```
#include <iostream.h>

int luvut[6];
int maara;

void lue(int luvut[])
{
int i;
cout << "Anna alkiot \n";
for (i = 0; i <6; i++)
{
cout << "Anna luettelon " << i << ". alkio\n";
cin >> luvut[i];
}
}

void vaihda(int* a, int* b)
{
int temp;
temp = *a;
*a = *b;
*b = temp;
}


void shell(int luvut[], int maara)
{
int k, valimatka, vaihto=1;
valimatka = maara / 2;
do
    do
    {
    vaihto = 0;
    for (k = 0; k < (maara - valimatka); k++)
    if (luvut[k] > luvut[k + valimatka])
        {
        swap(&(luvut[k]), &(luvut[k + valimatka]));
        vaihto = 1;
        }
    } while (vaihto);
while ( (valimatka /= 2) > 0);}
}

void tulosta(int luvut[])
{
int i;
cout << "Onko jarjestyksessa: \n";
cout << "Luettelon alkiot: \n";
```

```
for (i = 0; i <6; i++)
cout << "Luettelon " << i << ". alkio on: " << luvut[i] << "\n";
}

main()
{
lue(luvut);
shell(luvut, 6);
tulosta(luvut);
return 0;
}
```

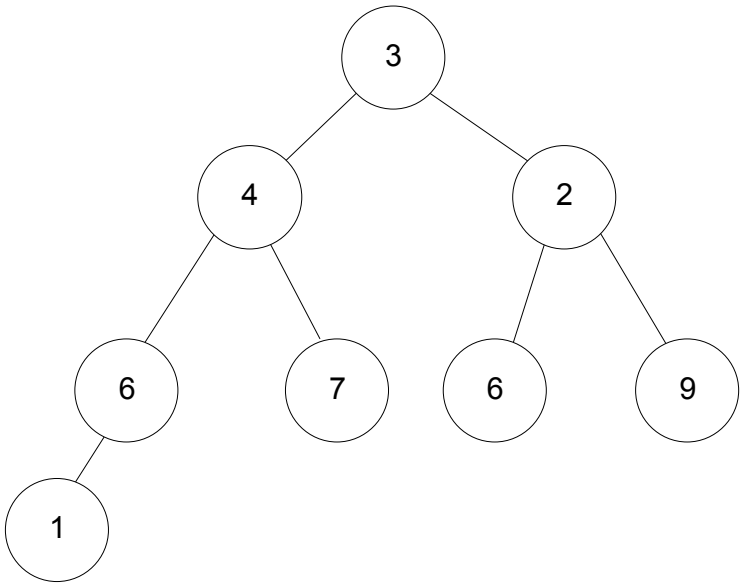
2.2.4 Kasalajittelu

Kasalajittelu perustuu binääripuihin. Lajittelunopeus on O(nlogn). Lajittelumenetelmän hyvänä puolena on esimerkiksi lomituslajitteluun verrattuna se, että lisämuistipaikkoja tarvitaan erittäin vähän.

Kasalajittelu liittyy binääripuihin ja käymme tässä samalla hieman läpi kasalajitteluun liittyvää perustietoa. *Binääripuun* sanotaan olevan *kasa*, jos se täyttää ehdon:
Jokaisen haarautumasolmun sisältämä arvo >= molempien solmun poikien arvo. Alipuun juuri sisältää siis aina alipuun suurimman arvon. Alipuiden avainten ei sen sijaan tarvitse olla järjestyksessä.

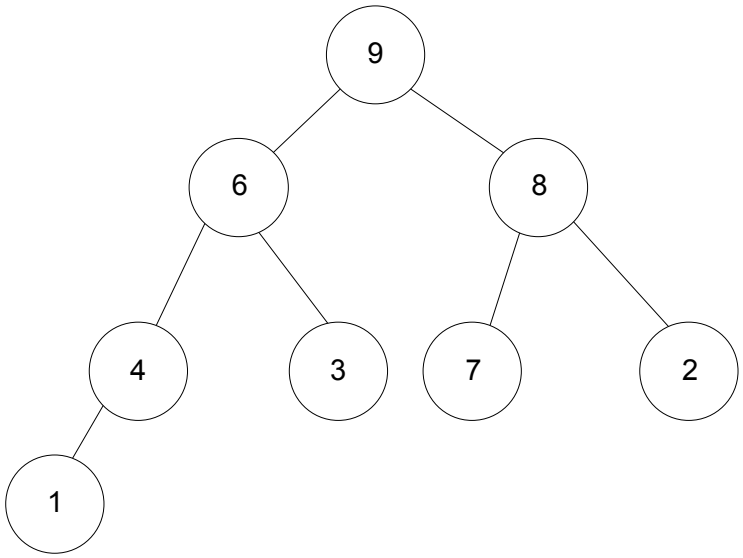
Kasalajittelussa taulukko kuvataan melkein täydellisenä binääripuuna. (Binääripuu on *melkein täydellinen*, jos puun kaikki lehdet ovat kahdella peräkkäisellä tasolla ja alemman tason lehdet ovat mahdollisimman vasemmalla ja jos ylempänä olevilla tasoilla on kaksi ei-tyhjää poikaa.)

Jos taulukko X[1..n] esimerkiksi sisältää alkiot: 3, 4, 2, 6, 7, 8, 9, 1, niin se kuvattaisiin (kasalajittelua varten) melkein täydellisenä binääripuuna seuraavasti:

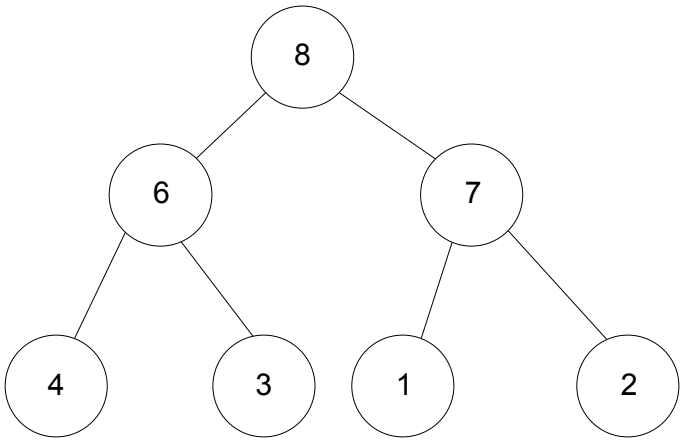


Kasalajittelussa muunnetaan binääripuu ensiksi kasaksi. Tällöin aloitetaan alimmalta tasolta ja kuljetaan ylöspäin, jolloin ylemmälle tasolle siirryttäessä vastaava alipuu muunnetaan kasaksi. Ajankohtaisen alipuun juuren avainta verrataan tällöin poikien avaimiin. Jos kasan ehto ei täyty (ks. ylempänä), vaihdetaan juuren avaimen ja suuremman poikien avaimen arvoja keskenään. Näin juuren avain siirtyy alaspäin, kunnes kasaehto täyttyy.

Simuloimme seuraavassa kasalajittelua manuaalisesti:
Jos teemme edellä kuvatun ensimmäisen vaiheen muunnoksen yllä olevalle melkein täydelliselle binääripuulle, saamme tulokseksi seuraavanlaisen puun:



Nyt on siis suoritettu kasalajittelun ensimmäinen vaihe ja taulukon suurin arvo on puun juuressa eli taulukkomme on nyt järjestyksessä X[9,6,8,4,3,7,2,1] Toisessa vaiheessa juuressa oleva suurin arvo vaihdetaan taulukon viimeisen alkion kanssa. Tämän jälkeen puu muunnetaan uudelleen kasaksi paitsi, että viimeisenä oleva alkio jätetään 'rauhaan', jolloin sitä ei enää esitetä puurakenteessa.
Seuraavan vaiheen tuloksena syntyy seuraava rakenne:



Nyt suurin arvo on jälleen puun juuressa. Seuraavaksi toistetaan toisen vaiheen menettelyä niin usein, että koko taulukko tulee lajitelluksi. Viimeisimpänä on toisen vaiheen jälkeen arvo 2, joka siis kolmannessa vaiheessa vaihdetaan arvon 8 kanssa ja jätetään sitten rauhaan. Puu muunnetaan uudelleen kasaksi jne.

2.2.5 Kuplalajittelu

Kuplalajittelussa aloitetaan luettelon oikeasta reunasta ja kuljetaan vasemmalle vaihtaen samalla arvoa silloin, kun esiin tullut arvo on suurempi kuin oikealta otettu arvo. Ensimmäisen kierroksen jälkeen ensimmäisenä on pienin arvo. Voidaan tietenkin aloittaa myös ensimmäisestä alkiosta ja kuljettaa suurin alkio viimeiseksi. Näin tehdään tässä ohjelmassa.

Kuplalajittelu:

```
#include <iostream.h>

void lue(int taulu[])
```

```
{
int i;
  cout << "Anna luettelon alkiot: \n";
     for (i=0; i < 6; i++)
     {
  cout << "Anna luettelon" << i << ". alkio: \n";
     cin >> taulu[i];
     }

}

void kupla(int taulu[], int maara)
{
int ok, i, temp;
    do
  {
  ok = 1;
    for (i=0; i < maara-1; i++)
    {
      if (taulu[i] > taulu[i+1])
      {
        temp = taulu[i];
        taulu[i] = taulu[i+1];
        taulu[i+1] = temp;
        ok = 0;
      }
    }
    }
  while (!ok);
}

void tulosta(int taulu[])
{

int i;
cout << "Onko järjestyksessä: \n";
cout << "Luettelon alkiot: \n";
for (i=0; i < 6; i++)
cout << "Luettelon " << i << ". alkio on " << taulu[i] << "\n";

}

void main()
{

int taulu[6] = {8, 9, 3, 2, 5, 1};
int maara;
      lue(taulu);
      kupla(taulu,6);
      tulosta(taulu);
```

```
}
```

2.2.6 Lisäyslajittelu (Insertion-sort)

Insertion-lajittelussa pidetään luettelon vasen puoli järjestyksessä ja siinä uusi alkio otetaan oikealta aina järjestyksessä. Tällöin oikealta ei etsitä pienintä arvoa, vaan otetaan aina seuraava alkio sen suuruudesta riippumatta. Sen jälkeen kuljetaan vasemmalle ja sijoitetaan arvo oikealle paikalleen. Ellei vasemmalta löydykään pienempää arvoa, sijoitetaan uusi alkio ensimmäiseksi ja liu'utetaan luetteloa oikealle. Tätä jatketaan, kunnes luettelo on lajiteltu.

Lisäyslajittelu:

```
#include <iostream.h>
void inssort(int taulu[], int maara);
void tulosta(int taulu[], int maara);

void main()
{
int taulu[] = {3, 6, 2, 5, 1, 8};
int maara = 6;
tulosta(taulu, maara);
inssort(taulu, maara);
tulosta(taulu, maara);

}

void tulosta(int taulu[], int maara)
{
int i;
for (i = 0; i < maara; i++)
cout << taulu[i] << "\n";
}


void inssort(int taulu[], int maara)
{
int a, i, temp, pos, min, uusiarvo, uusipaikka, nykypaikka;
min = taulu[0];
for (i = 0; i < maara; i++)
    if (taulu[i] <= min)
    {
        min = taulu[i];
        pos = i;
    }
```

```
        temp = taulu[0];
        taulu[0] = min;
        taulu[pos] = temp;
        for (uusipaikka = 1; uusipaikka < maara; uusipaikka++)
        {
          uusiarvo = taulu[uusipaikka];
          nykypaikka = uusipaikka;
          while (taulu[nykypaikka - 1] > uusiarvo)
          {
              taulu[nykypaikka] = taulu[nykypaikka - 1];
              nykypaikka = nykypaikka - 1;
          }
            taulu[nykypaikka] = uusiarvo;
        }
}
```

Lajittelusovellus

Tietenkin myös tietueita voidaan lajitella. Yksi mahdollisuus on tällöin lukea tiedoston tietueet ensin 1-dimensioiseen taulukkoon, jonka alkiot ovat tietuetyyppisiä. Tämän jälkeen taulukon alkiot lajitellaan ja lopuksi luetaan tyhjennettyyn tai uuteen tiedostoon.

Seuraavaksi lajittelemme oliotaulukon. Taulukkoon sijoitetaan 10 oliota, joilla on jäsenmuuttuja itsAge. Kun kunkin taulukossa olevan olion jäsenmuuttujaan on sijoitettu satunnainen arvo, taulukko lajitellaan.

Oliotaulukon lajittelu sovellusesimerkkinä:

```
#include <iostream.h>
#include <stdlib.h>

class HENKILO
{
public:
HENKILO() { hlokoodi = 0; palkka=5; }
~HENKILO() {}
int HaeKoodi() const { return hlokoodi; }
int HaePalkka() const { return palkka; }
void AsetaKoodi(int koodi) { hlokoodi = koodi; }
friend main();
private:
int hlokoodi;
int palkka;
```

```
};
void vaihda(HENKILO& a, HENKILO& b);

int main()
{
HENKILO Hlosto[10]; // Taulukkoon 5 HENKILO-oliota; taulukko pinoon
int i;

for (i = 0; i < 10; i++)
    Hlosto[i].AsetaKoodi(rand() % 9);

for (i = 0; i < 10; i++)
      cout << "Hlon #" << i+1<< ": " << Hlosto[i].HaeKoodi() <<
endl;

for (i = 0; i < 9; i++)
    for (int j = i + 1; j < 10; j++)
        {
        if (Hlosto[i].hlokoodi > Hlosto[j].hlokoodi)
        vaihda (Hlosto[i], Hlosto[j]);
        }
for (i = 0; i < 10; i++)
      cout << "Hlon #" << i+1<< ": " << Hlosto[i].HaeKoodi() <<
endl;


return 0;
}

void vaihda(HENKILO& a, HENKILO& b)
{
HENKILO Temp;
Temp = a;
a = b;
b = Temp;
}
```

2.2.7 Tehokkuusnäkökohtia

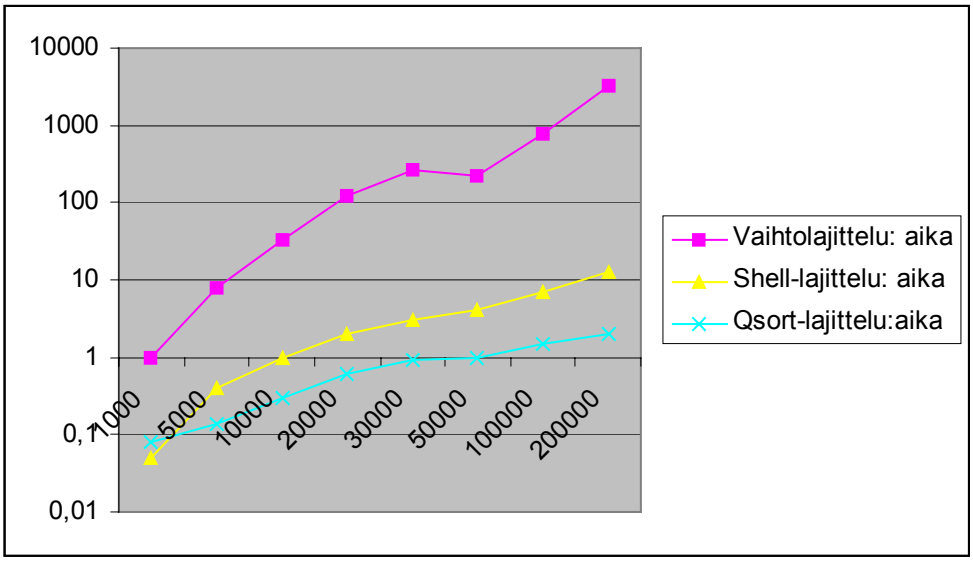
Taulukko voidaan lajitella usealla eri tavalla. Seuraavassa taulukossa on aikoja (sekunneissa), kun eri kokoisia taulukoita on lajiteltu 3 eri algoritmilla. Alustana oli satasen pentti. Meitä kiinnostavat lähinnä aikojen suhteelliset muutokset.

Lajittelutavat ovat seuraavat:
Vaihtolajittelu
Shell-lajittelu

stdlib-kirjaston QuickSort-lajittelu

Alkioita	Vaihtolajittelu: aika	Shell-lajittelu: aika	Qsort-lajittelu: aika
1000	1	<1	<1
5000	8	<1	<1
10000	32	1	<1
20000	124	2	<1
30000	268	3	<1
50000	218	4	1
100000	776	7	1,5
200000	3236	13	2

Tulos graafina:



Ylin käyrä kuvaa graafissa vaihtolajittelun tehokkuutta. Logaritminen asteikko antaisi tietenkin paremman kuvan muiden lajittelumenetelmien (shell, qsort) lajitteluajoista, mutta muokkaisi samalla käyrien muotoja.

Standardikirjasto stdlib.h sisältää qsort()-funktion, joka käyttää lajittelussaan QuickSort-algoritmia. Juuri tuota funktiota on käytetty lajittelun tehokkuuden arvioinnissa kolmantena menettelytapana.

stdlib.h sisältää funktion prototyypin:

```
void qsort(void *base, size_t nelem, size_t width, int (_USERENTRY
*fcmp)(const void *, const void *));
```

Seuraavana on ohjelma, joka hyödyntää stdlib-kirjaston qsort()-lajittelufunktiota. Huomaa erityisesti void-tyyppisten osoittimien ulkoiset tyyppimuunnokset vertaa()-funktion rungossa.

stdlib.h: qsort():

```
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>

int vertaa( const void *a, const void *b);
int lista[5]= { 1, 2, 8, 3, 4 };

int main(void)
{
    int  x;
```

```
    qsort(list, 5, sizeof(list[0]), vertaa);
    return 0;
}

int vertaa( const void *a, const void *b)
{
if ( *(int *)a == *(int *) b ) return 0;
if ( *(int *)a > *(int *) b ) return 1;
if ( *(int *)a < *(int *) b ) return -1;

}
```

STL-kirjaston sort()

Olisimme voineet käyttää stdlib.h-kirjaston qsort()-funktion sijaan STL-kirjaston vastaavaa sort()-funktiota.

Seuraavana on ohjelma, joka käyttää STL (Standard Template Library) -kirjaston lajittelumenettelyä.

STL: sort():

```
#include <iostream.h>
#include <stdlib.h>
#include <algorithm>
#include <time.h>
#include <conio.h>

using namespace std;

void main()
{
randomize();
int i;

const int lkm = 10000;

int luvut[lkm];

int * an = luvut + lkm;

for (i=0; i < lkm; i++)
{
luvut[i] = rand();
}
```

```
/*
cout << "Ennen lajittelua \n";
for (i=0; i < lkm; i++)
cout << luvut[i] << "\n";   */

time_t t1, t2;
t1 = time(NULL);
cout << t1 << "\n";

sort(luvut, an);

t2 = time(NULL);
cout << t2 << "\n\n";

cout << (t2 - t1) << "\n";

cout << "Lajittelun jälkeen \n";

for (i=0; i < lkm; i++)
cout << luvut[i] << "\n";

getch();

}
```

Ainakin vaihtolajittelu (jossa on runsaasti vaihtoja) sujuu hieman nopeammin, kun arvojen vaihtamisen toteuttava funktio laitetaan **inline**-funktioksi. Tällöin funktion koodi puretaan kutsujan koodivirtaan eikä erillistä hyppyä funktioon tarvitse tehdä.

Huomautus
Mikäli myös pienten alkiomäärien lajitteluaikoja haluttaisiin siepata, on myös sekuntien murto-osat saatava selville. Se voidaan toteuttaa esimerkiksi otsikkotiedoston dos.h kautta saatavalla gettime()-funktiolla, joka ottaa parametrikseen tietuemuuttujan, jonka kenttinä ovat myös sekuntien murto-osat.

struct time –rakenne:

```
struct time {
   unsigned char ti_min;      // minuutit
   unsigned char ti_hour;     // tunnit
   unsigned char ti_hund;     // sadasosasekunnit
   unsigned char ti_sec;      // sekunnit
};
```

Seuraavana on esimerkki sadasosasekuntien sieppaamisesta:

```
#include <iostream.h>
#include <conio.h>
#include <dos.h>

main()
{
struct  time t1, t2;

int tun1, min1, sek1, sad1, sek2, sad2, tun2, min2;
int kaikki1, kaikki2;
gettime(&t1);
sek1 = t1.ti_sec; sad1 = t1.ti_hund;
tun1 = t1.ti_hour; min1 = t1.ti_min;
kaikki1 = (3600 * tun1 + 60 * min1 + sek1) * 100 + sad1;

for (int k=0; k < 1900000; k++);

gettime(&t2);
sek2 = t2.ti_sec; sad2 = t2.ti_hund;
tun2 = t2.ti_hour; min2 = t2.ti_min;
kaikki2 = (3600 * tun2 + 60 * min2 + sek2) * 100 + sad2;

cout << "Aikaa kului sadasosasekunneissa\n";
cout << (kaikki2 - kaikki1);


getch();
}
```

Tulostus, kun tyhjää silmukkaa ajettiin pari miljoonaa kertaa:

2.2.8 Yleiskäyttöinen lajittelufunktio

Olisi tietenkin mukavaa, jos voisimme käyttää samaa funktiota sekä kokonaislukujen että liukulukujen lajitteluun. Muutammekin seuraavassa vaihtolajitteluohjelmaa yleiskäyttöisemmäksi parametrisoinnin avulla eli määrittämällä funktion parametreiksi mallin (template) mukaiset parametrit:

Geneerinen lajittelufunktio:

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

template<class T>
void swap(T * a, T * b);

template<class T>
void lajittele(T luvut[], long maara);

main()
{
randomize();
int i, j;
const long int lkm = 10;
int luvut[lkm];
double desit[lkm];

for (i=0; i < lkm; i++)
{
luvut[i] = rand() % 29950;
desit[i] = 1.13 * ( rand() % 29950);
```

```
}

cout << "Ennen lajittelua \n";
for (i=0; i < lkm; i++)
cout << desit[i] << "\n";
//cout << luvut[i] << "\n";

time_t t1, t2;
t1 = time(NULL);
cout << t1 << "\n";

lajittele(desit, lkm);

t2 = time(NULL);
cout << t2 << "\n";

cout << "Lajittelun jälkeen \n";

for (i=0; i < lkm; i++)
cout << desit[i] << "\n";
//cout << luvut[i] << "\n";

getch();

}

template<class T>
void lajittele(T luvut[], long maara)
{
    for (int k = 0; k < (maara - 1); k++)
        for (int s = k+1; s < maara; s++)
        {
            if (luvut[k] > luvut[s])
                swap(&(luvut[k]), &(luvut[s] ));
        }
}

template<class T>
void swap(T * a, T * b)
{
T temp;
temp = *a;
*a = *b;
*b = temp;
}
```