11 TEKSTIALGORITMID

Enamik seni käsitletud algoritme töötavad arvulistel andmetüüpidel, mis on oma olemuselt „lihtsad“: üks väärtus vastab üldjuhul ühele mälupesale ning protsessori käsud on optimeeritud nendega operatsioonide sooritamisele. Inimesed eelistavad aga enamasti tekstiga töötamist arvudele, seetõttu ei pääse ka arvutid teksti käsitlemise kohustusest.

Arvutustehnika levides ja abstraktsioonitaseme tõustes on väga levinud ka mitmesuguste andmete esitamine „inimloetaval“ kujul, näiteks XML või JSON formaadis, sealhulgas ka juhul, kui samade andmete kahendesituses käitlemine oleks tuhandeid kordi kiirem. See tähendab, et kaasaegne arvuti kulutab enamiku oma protsessoriajast just nimelt tekstitüüpi andmetega tegeledes.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="points">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="point">
          <xs:complexType>
            <xs:attribute name="x" type="xs:unsignedShort" use="required" />
            <xs:attribute name="y" type="xs:unsignedShort" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Kui arvudel on tavalisteks operatsioonideks liitmine, korrutamine, bitinihutamine ja muud aritmeetilised operatsioonid, siis teksti puhul on põhikohal erinevate tekstijuppide võrdlemine, kokkupanek või siis ühest tekstilõigust teise otsimine. Et aga neid operatsioone võimalikult efektiivselt sooritada, muutub väga tähtsaks see, kuidas me teksti arvuti mälus hoiame. Kui arvu liigutamine ühelt mäluaadressilt teisele võtab protsessoril vaid nanosekundi, siis pika teksti kopeerimine on palju pikem protsess.

Enamiku siin peatükis toodud algoritmide peamine idee seisnebki selles, kuidas selliseid liigutamisi ja kopeerimisi võimalikult palju vältida.

11.1 TEKSTI PARSIMINE

Enamik tekstist, millega inimesed tegelevad, on „proosatekstid“, mis koosnevad sõnadest ja lausetest, kuid millel pole kuigi palju täiendavaid reegleid. Peale selle on olemas aga väga palju tekstivormis andmeid, millel on keerulisem struktuur: programmeerimiskeeltes kirjutatud kood, algebralised avaldised, HTML-tekst jne.

$$\frac{\frac{1}{3a}-\frac{1}{3b}}{\frac{a}{b}-\frac{b}{a}} = \frac{\left(\frac{b-a}{3ab}\right)}{\left(\frac{a^2-b^2}{ab}\right)}$$

$$= \frac{b-a}{3ab} \cdot \frac{ab}{a^2-b^2} = \frac{(b-a)}{3ab} \cdot \frac{ab}{(a+b)(a-b)}$$

$$= \frac{(b-a)^{-1}}{3ab} \cdot \frac{ab}{(a+b)(a-b)} = -\frac{1}{3(a+b)}$$

Inimene näeb sellise teksti struktuuri tervikuna, kuid arvuti jaoks on tegu märgijadaga nagu iga teine. Et infoga midagi intelligentset teha, tuleb tekst teisendada mingiks andmestruktuuriks (näiteks puuks), millel saab juba täiendavaid operatsioone läbi viia. Sellist teksti paremini käsitletavale kujule viimist nimetatakse **parsimiseks**.

Vaatame järgmist klassikalist ülesannet:

11.1.1 Ülesanne: arvutustehe

On antud aritmeetiline avaldis, mis sisaldab positiivseid täisarve, liitmis-, lahutamis, korrutamis- ja jagamismärki (`+,-,*,:`) ning sulge (`(,)`). Leia tehte vastus.
NÄIDE:
`4*(5+3)-(2*6+3):15+11`
Vastus:
42

Tavapäraselt matemaatikas kasutatav tehtekuju, kus tehtemärk ehk operaator on arvude (argumentide) vahel, nimetatakse **infiks**-kujuks.

Infiks-kujul tuleb tehteid sooritada üsna suvalises järjekorras, näiteks siin ei saa esimest tehet (korrutamine) enne sooritada, kui oleme teinud sulgudes oleva liitmistehte. Sellele järgnev lahutamistehe jääb aga juba järgneva kolme tehte ootele.

Seetõttu on antud ülesandes mugavam viia tehe **postfiks**-kujule. Postfiks-kuju ehk pööratud Poola notatsioon on selline tehete esitus, kus operaator järgneb argumentidele. Näiteks tavapärane liitmistehe $5 + 6$ on postfiks-kujul $5\ 6\ +$. Näiteavaldisest esimene osa $4 * (5 + 3)$ saab kuju $5\ 3\ +\ 4\ *$ ning kogu tehe on postfiks-kujul järgmine:

$$4\ 5\ 3\ +\ *\ 2\ 6\ *\ 3\ +\ 15\ :\ -\ 11\ +$$

Üheks postfiks-kuju eeliseks on see, et puudub vajadus sulgude järele ning tehetel ei ole erinevat kaalu – nii võib „unustada“ algklassides õpitud reegli: „Enne korrutan ja jagan, siis liidan ja lahutan“.

Kuidas siis teisendada avaldis postfiks-kujule ja leida selle väärtus? Selleks on siiski vaja teada just eelpool nimetatud reeglit. Siin on väike abifunktsioon, mis tehetele prioriteedid annab:

```
int pref(char ch)
{
     switch (ch) {
          case '*':
          case ':': return 1;
          case '+':
          case '-': return 2;
          case ('('): return 3;
          default: return 0;
     }
}
```

Kui vaadata näidet, on näha, et arvud on tehtes ikka samas järjekorras, vaid operaatorid ja nende omavaheline järjestus muutub. Seega arvud saab kohe nö ühest stringist teise sõidutada,

tehtemärgid tuleb aga vahepeal kõrvalteele (pinusse) ootele panna, et need siis õigel ajal arvude sekka tagasi lükata:

```
string postfix(string s) {
	string vastus = "";
	stack<char> op;

	int index = 0;
	while (index < s.length()) {
		while (index < s.length() && isdigit(s[index])) {
			vastus += s[index];
			index++;
		}
		vastus += ' ';

		if(index < s.length()) {
			char mark = s[index];
			if (mark == '+' || mark == '-' || mark == '*' || mark == ':') {
				while (!op.empty() && pref(op.top()) <= pref(mark)) {
					vastus += op.top();
					op.pop();
				}
				op.push(mark);
			}
			else if (mark == '(') op.push(mark);
			else if (mark == ')') {
				while (op.top() != '(') {
					vastus += op.top();
					op.pop();
				}
				op.pop();
			}
		}
		index++;
	}

	while (!op.empty()) {
		vastus += op.top();
		op.pop();
	}
	return vastus;
}
```

Postfiks-kujul avaldist on lihtne arvutada, kasutades taas pinu:

```
int arvuta(string s) {
	stack<int> vastus;
	int index = 0;
	while (index < s.length()) {
		if (isdigit(s[index])) {
			int d = s[index++] - '0';
			while (index < s.length() && isdigit(s[index])) {
				d = d * 10 + s[index++] - '0';
			}
			vastus.push(d);
		}
		if (s[index] == ' ') index++;
		else {
			int d2 = vastus.top();
			vastus.pop();
			int d1 = vastus.top();
			vastus.pop();
			switch (s[index++]) {
			case '+': vastus.push(d1 + d2); break;
			case '-': vastus.push(d1 - d2); break;
			case '*': vastus.push(d1 * d2); break;
			case ':': vastus.push(d1 / d2); break;
			}
		}
	}
	return vastus.top();
}
```

11.2 ALAMSTRINGI LEIDMINE

Nagu sissejuhatuses öeldud, on tekstitöötlusalgoritmide üheks peamiseks ülesandeks etteantud stringist mingi teise, samuti etteantud alamstringi leidmine.

11.2.1 Mõisted

Tuletame kõigepealt meelde põhimõisted. **String** on tervikuna käsitletav samalaadsete elementide, näiteks märkide, järjend. **Märk** on andmete esituseks, korralduseks või juhtimiseks kasutatava elemendihulga liige. **Märgistik** on määratud otstarbeks täielik erinevate märkide lõplik hulk.

Enamasti tähistame stringi $S = s_1 s_2 \dots s_n$, kus s_i on märk kasutatavas märgistikus M ehk $\forall s_i \in M$.

Stringi $S = s_1 s_2 \dots s_n$ **alamstringiks** nimetame sellist järjendit $A = a_1 a_2 \dots a_m$, kus $m \leq n$ ja leidub selline indeks i, et $\forall a_j \in A, a_j = s_{i+j}$ ja $s_{i+j} \in S$. Näiteks stringi `pesukaru` alamstringiks on string `suka`, aga mitte string `suru`.

Stringi S **sufiksiks** nimetatakse sellist stringi S alamstringi, mis lõpeb stringi S lõpus. Näiteks stringi `pesukaru` sufiksid on `pesukaru`, `esukaru`, `sukaru`, `ukaru`, `karu`, `aru`, `ru` ja `u`. Stringil pikkusega n on täpselt n sufiksit.

Stringi S **prefiksiks** nimetatakse sellist stringi S alamstringi, mis algab stringi S algusest. Näiteks stringi pesukaru prefiksid on p, pe, pes, pesu, pesuk, pesuka, pesukar ja pesukaru. Nii nagu sufikseidki, on stringil pikkusega n täpselt n prefiksit.

11.2.2 Lihtne alamstringi otsimise algoritm

On antud kaks stringi T ja S. Leia kõik stringi S esinemised stringis T.
Sisendi esimesel real on string T. Teisel real on string S.
Väljastada ühel real tühikutega eraldatuna kõik asukohad tekstis T, millest algab alamstring S. Kui string S ei sisaldu stringis T, väljastada -1.
NÄIDE 1:
varvas
va
Vastus:
1 4
NÄIDE 2:
varvas
aas
Vastus:
-1

Kõige vahetum lähenemine on muidugi hakata stringe märk-märgilt omavahel võrdlema: kui esimesed märgid langevad kokku, võrdleme teisi jne, kuni alamstring saab võrreldud (ja on leitud!) või kuni märgid ei lange kokku, millisel juhul otsime edasi kõrvutades alamstringi esimest märki stringi teise märgiga jne.

Siin on funktsioon, mis seda teeb:

```
string tekst, sona;

void lihtne_alamstring() {
    for (int i = 0; i <= tekst.length() - sona.length(); i++) {
        for (int j = 0; j < sona.length(); j++) {
            if (tekst[i + j] == sona[j]) {
                if (j == sona.length() - 1) {
                    cout << i + 1 << " ";
                };
            }
            else break;
        }
    }
}
```

Huvitav on see, et suvalise teksti korral on selline lähenemine päris hea ja keskmise keerukusega $O(n)$, kus n on stringi ehk teksti pikkus. Suvalises juhuslikus tekstis on meil enamasti kasutusel ca 200 erinevat tähemärki ja tõenäosus, et esimene võrreldav märk kokku langeb, on vaid $\frac{1}{200}$. Tõenäosus, et kaks esimest kokku langevad, aga juba vaid $\frac{1}{200*200}$ jne – seega tekib üliharva olukord, kus tuleb mitmeid märke võrrelda otsustamaks, et kokkulangevust tegelikult ei ole.

Samas isegi tavalised kirjatekstid ei ole täiesti juhuslikud. Esiteks ei esine kõik märgid sama sagedusega, teiseks on ka tähejärjenditel oma esinemissagedused. Näiteks, kui otsida siit raamatust

alamstringi `ktsea`, siis olenemata sellest, et kõik kasutatud märgid on päris sagedased, ei tule ilmselt teha väga palju võrdlusi, kuna ei leidu eriti sõnu, mis algaks täheühendiga `kt`. Samas, kui otsida stringi `arvustus`, siis tuleb teha märkimisväärselt võrdlusi, mis tulemust ei anna, kuna ilmselt on siin raamatus küllalt sõnu `arv`, `arvu`, `arvus` ja `arvust`.

Ka siis kasvab prefiksite kokkulangevuse osakaal, kui kasutusel on väiksem märgistik. Näiteks, kui otsime telefoninumbrite seast mingit konkreetset numbrit, siis on küllaltki suur tõenäosus, et otsitava numbri algus langeb kokku päris mitmete teiste telefoninumbrite osadega. Eriti muidugi algustega, kuna selleski osas ei ole tegemist täiesti suvaliste arvujärjenditega.

Võid kindel olla, et programmeerimisvõistlustel on testid nii koostatud, et tekstid ei ole sugugi juhuslikud ja sellisel juhul läheneb keerukus halvimale juhule ehk $O(mn)$, kus m on otsitava alamstringi pikkus ja n teskti pikkus.

11.3 KNUTH-MORRIS-PRATTI ALGORITM

1970. aastatel mõtlesid Stanfordi ülikooli professor (ja üldse üks kuulsamaid arvutiteadlasi) Donald Knuth ja tema õpilane Vaughan Pratt välja algoritmi, kuidas tekstist efektiivsemalt alamstringi leida. Enam-vähem samal ajal jõudis sarnase algoritmini ka kolmas mees, James H. Morris Carnegie-Melloni ülikoolist. Kolmekesi avaldasid nad vastavasisulise töö aastal 1977. Sellest ajast tuntakse seda kui Knuth-Morris-Pratti algoritmi (lühendatult tihti lihtsalt KMP-algoritm).

Algoritmi peamiseks mõtteks on see, et juba vaadatud teksti ei peaks uuesti üle vaatama. Oletame, et meil on tekst `pannkook kohupiimaga kook koorega pannkook kodujuustuga` ja otsime sellest alamstringi `kook kodu`. Tähistame parasjagu vaadeldava asukoha tekstis i-ga ja parasjagu vaadeldava asukoha alamstringis j-ga

Alguses ei erine algoritm sugugi eelnevalt tutvustatud lihtsast algoritmist, sest esimene täht juba kokku ei lange ja sellest oluliselt rohkem infot saada ei ole. Seisuks on `i=0`, `j=0` ja `s[i]≠a[j]`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
kook kodu
```

Järgmiseks suurendame `i-d` ühe võrra ja `j` jääb samaks:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
 kook kodu
```

Kuna ka siin kokkulangevust pole, jätkame samamoodi, kuni `i=4`. Kuna `s[4]=a[0]`, siis suurendame edasi nii i-d kui ka j-i, kuni enam kokkulangevust pole. See juhtub kohal `j=7`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
    kook kodu
```

Nüüd lihtne algoritm alustaks taas kohast `i=5` ning `j=0`. KMP aga jätkab kohast `i=11 ja j=2`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
         kook kodu
```

Kuna siingi kokkulangevust pole, siis edasi on veel ükshaaval minekut kuni `i=21`, millele järgnevad taas kokkulangevused, kuni `i=28` ja `j=7`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
                     kook kodu
```

Nüüd saame taas jätkata kohalt `i=28` ja `j=3` ja kuigi `s[28]=j[3]`, siis juba `s[29]≠a[4]`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
                         kook kodu
```

Kuna ka siin kokkulangevust pole, liigume alates `i=29` ja `j=0` ükshaaval kuni `i=34`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
                                  kook kodu
```

Sellest näitest on ilusti näha, kuidas tekstis tagasi liikumist ei toimu, kuigi võrdusi võib sama positsiooni korral olla vaja teha rohkem kui ühe korra. Pigem aga on küsimus selles, et kuidas me teame, kui palju tuleb otsitavat alamstringi nihutada, et arvestaksime esiosa ehk prefiksi kokkulangemisega?

11.3.1 Kokkulangevuste tabel

Kokkulangevustega arvestamiseks kasutatakse osaliste kokkulangevuste tabelit. Selleks töödeldakse enne otsitavat stringi.

i	1	2	3	4	5	6	7	8	9
a[i]	K	o	o	k	’’	K	o	d	u
nihe	0	1	1	0	1	0	1	3	1

Vaatame näitena juba esimesest peatükist tuttavat ülesannet Eesti 2016. aasta informaatikaolümpiaadi eelvoorult.

11.3.2 Hulknurgad

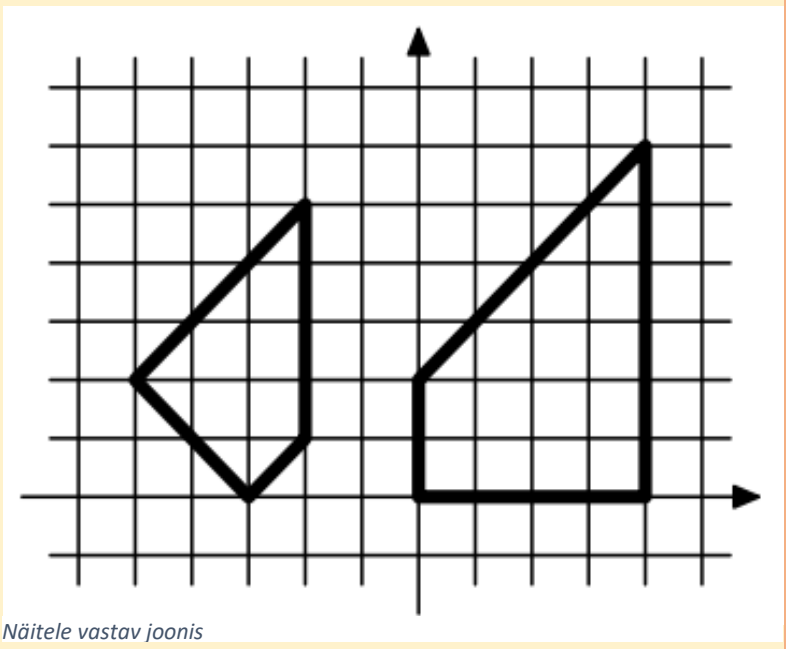
Juku õpib koolis hulknurkade sarnasust ja saab teada, et hulknurgad on sarnased, kui nende vastavate nurkade suurused on võrdsed ja vastavate külgede pikkused võrdelised. Sarnased hulknurgad võivad olla omavahel pööratud, peegeldatud ja nihutatud. Sarnaste hulknurkade vastavate külgede pikkuste jagatist nimetatakse nende sarnasusteguriks.
Kodutööna saab ta hulga hulknurki, mille sarnasustegureid on vaja määrata. Jukul on fanaatiline matemaatikaõpetaja, kes andis tööna väga paljude nurkadega hulknurki. Aita Juku hädast välja.
Sisend. Tekstifaili esimesel real on hulknurga tippude arv N (3<=N<=200 000).
Faili teisel real on 2 · N täisarvu lõigust −10^9 kuni 10^9: esimese hulknurga tippude x- ja y-koordinaadid. Kolmandal real on samuti 2·N arvu: teise hulknurga tippude koordinaadid. Tipud võivad olla antud nii päripäeva kui vastupäeva järjekorras. Antud punktid moodustavad alati hulknurga, milles pole ühtelangevaid punkte, sirgnurki, ega endaga lõikumisi.
Väljund. Kui hulknurgad on sarnased, siis kirjutada väljundi esimesele reale täpselt üks reaalarv, mis näitab, mitu korda on esimene hulknurk suurem kui teine (kui esimene hulknurk on väiksem, on ka vastus väiksem kui 1).
Teisele reale kirjutada täisarv, mis näitab, mitmes teise hulknurga tipp vastab esimese hulknurga esimesele tipule (mõlema hulknurga tipud on nummerdatud alates ühest nende failis esitamise järjekorras).
Kui hulknurgad ei ole sarnased, kirjutada väljundi ainsale reale −1.

NÄIDE:

```
4
0 0 4 0 4 6 0 2
-2 5 -2 1 -3 0 -5 2
```

Vastus:

```
1.414213
3
```



Näitele vastav joonis

Vastuse teine osa on 3, sest teise hulknurga kolmas punkt (−3; 0) vastab esimese hulknurga esimesele punktile (0; 0).

Sellel ülesandel oli kaks poolt: geomeetriline ja tekstitöötluse pool. Kui esimest käsitlesime raamatu alguses, siis nüüd uurime teksti osa.

Kui mõelda küljepikkuse ja nurgasuuruse kombinatsioonist kui tähemärgist, siis taandub ülesanne kontrollile, kas kaks stringi on saadud üksteise esimese ja tagumise jupi äravahetamise teel. Näiteks stringid CABAABBA ja BBACABAA on selles mõttes ekvivalentsed.

Üks võimalus sellise ekvivalentsuse kontrollimiseks oleks proovida stringi kõiki rotatsioone. Kavalam on aga tähele panna, et üks string on teise rotatsioon juhul, kui ta on teise stringi kahekordse koopia alamstring. Seega eeltoodud näites CABAABBA ja BBACABAA ekvivalentsus järeldub sellest, et CABAABBA on BBA**CABAABBA**CABAA alamstring (ja ka BBACABAA on CABAA**BBACABAA**BBA alamstring).

Nüüd aga on juba lihtne näha, et ekvivalentsuse kontroll taandub lihtsalt efektiivsele alamstringi leidmisele. Kasutame seekord Knuth-Morris-Pratti algoritmi.

Kõigepealt funktsioon, mis arvutab kokkulangevuste tabeli:

```
Segment** pat;
Segment** str;
int* lps;
int m;
int n;

void ComputeLpsArray()
{
     int len = 0;
     int i = 1;

     lps[0] = 0;

     while (i < m) {
          if (pat[i]->Equals(pat[len])) {
               len++;
               lps[i] = len;
               i++;
          }
          else {
               if (len != 0) {
                    len = lps[len - 1];
               }
               else {
                    lps[i] = 0;
                    i++;
               }
          }
     }
}
```

Stringide sobimist kontrolliv osa:

```
int KmpMatch() {
      int i = 0;
      int j = 0;
      lps = new int[m];

      ComputeLpsArray();

      while (i < 2 * n) {
            if (pat[j]->Equals(str[i % n])) {
                  j++;
                  i++;
            }

            if (j == m) {
                  int vas = i - j + 1;
                  return vas;
            }
            else if (i < 2 * n && !pat[j]->Equals(str[i % n])) {
                  if (j != 0)
                        j = lps[j - 1];
                  else
                        i++;
            }
      }

      return -1;
}
```

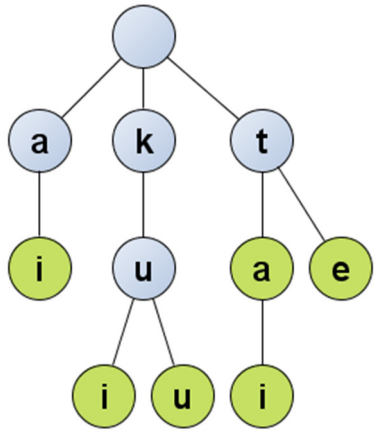
11.4 SUFIKSIPUUD

Mitmesugustes tekstiotsingualgoritmides on kasulik organiseerida stringid nii, et nende kattuvad osad on arvuti mälus samas kohas. Loomulik struktuur selliste kattuvuste kokkupanekuks on puu.

11.4.1 Prefiksipuu

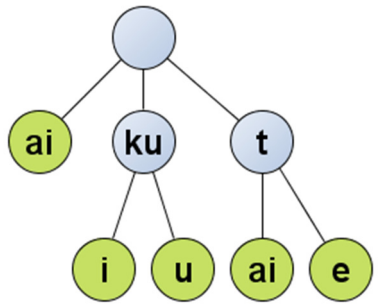
Prefikispuu (i.k *trie*, *digital tree, prefix tree*) ingliskeelne nimetus *trie* tuleb sõnast *re**trie**ve*, mis viitabki otsimisele. Tuletame meelde, et otsingupuu on puu, kus hoitakse võtme järgi väärtusi ning oluline ongi, et väärtus on võtme järgi kiiresti ja efektiivselt leitav, st võtmed on mingil moel sorteeritud. Kahendotsingupuus näiteks kehtis reegel, et tipu vasakus alampuus on selle tipu võtmest väiksemad võtmed, paremas aga suuremad võtmed. Igas tipus hoitakse üht võtit.

Juhul, kui võtmeteks on sõnad, saabki kasutada otsingupuuks prefiksipuud. Prefiksipuu vahetippudes hoitakse osa võtmest ning kogu võti moodustub läbitavates tippudes või servades oleva info põhjal. Iga tipu kõigil järglaste võtmel on sama prefiks.



Prefiksipuu. Rohelisega on tähistatud need tipud, milles on ka väärtused. Võtmed selles puus on ai, kui, kuu, ta, tai ja te.

Kompaktne prefiksipuu (i.k *compact prefix tree*, *radix tree*) on selline prefiksipuu, kus iga alluv, mis on oma vanema ainsaks alluvaks, ühendatakse vanemaga:



Kompaktne prefiksipuu

Mõned prefiksipuu eelised paisktabeli ees:

- Halvim juht info leidmiseks on O(m) , kus m on võtme pikkus
- Kokkupõrgete ehk kollisioonide puudumine
- Saab esitada võtmed kiiresti tähestikulises järjekorras

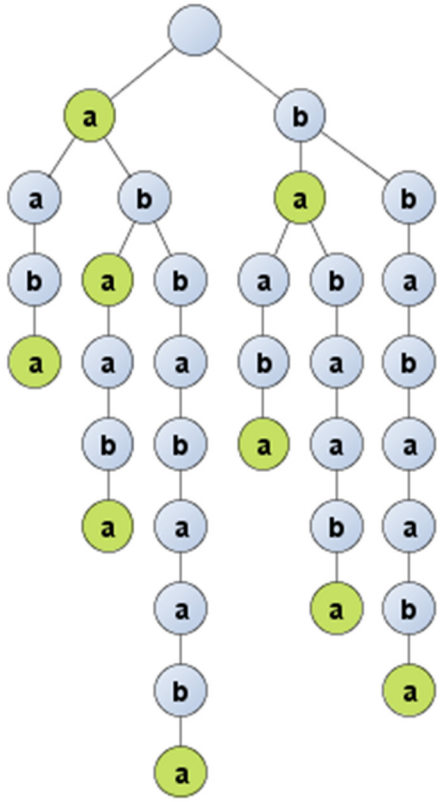
Prefiksipuid kasutatakse näiteks sõnade äraarvamisel (*autocomplete*) ja spellerites.

11.4.2 Sufiksipuu

Nimetame stringi S i-ndaks sufiksiks sellist stringi s sufiksit, mis algab positsioonilt i. Näiteks stringi S=’abbabaaba’ sufiksid on:

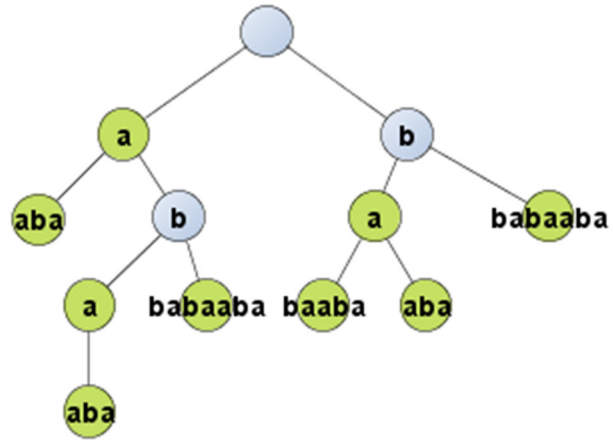
I	1	2	3	4	5	6	7	8	9
i-s sufiks	abbabaaba	bbabaaba	babaaba	abaaba	baaba	aaba	aba	ba	a

Sufiksipuuks (i.k *suffix trie*) nimetatakse stringi sufiksitest moodustatud prefiksipuud.



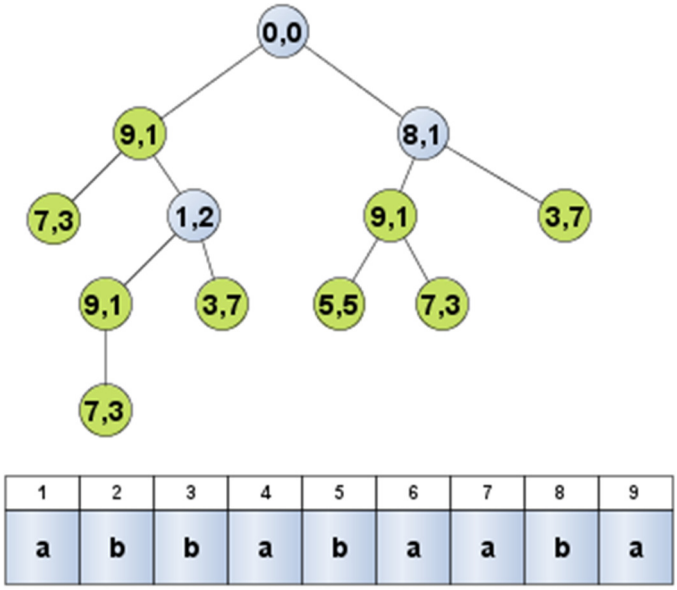
Stringi 'abbabaaba' sufiksipuu

Samuti nagu prefiksipuust sai teha kompaktse prefiksipuu, võime esitada ka sufiksipuu kompaktsel kujul:



Stringi 'abbabaaba' kompaktne sufiksipuu

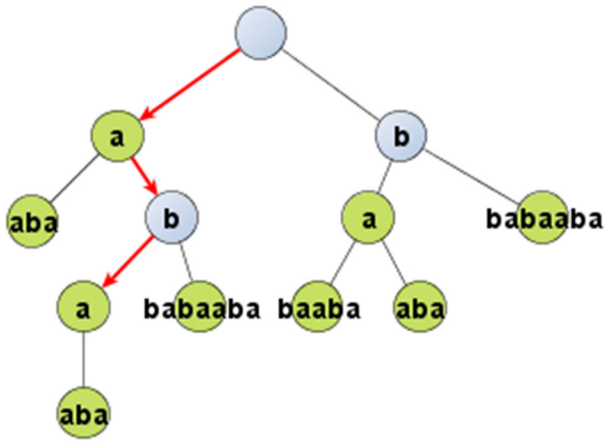
Veelgi enam: me võime puus hoida ainult algusindeksi ja pikkuse:



11.4.3 Sufiksipuu kasutamine

11.4.3.1 Alamstringide leidmine

Kui alamstring A leidub tekstis, siis see on mingi selle teksti sufiksi prefiksiks. Proovime näiteks leida alamstringi ’aba’ eelpool toodud stringist ’abbabaaba’. Kui vaadata kõiki selle stringi sufikseid, siis pole raske veenduda, et ’aba’ on täpselt kahe – 4. ja 7. sufiksi prefiksiks.



Alamstringi ’aba’ leidmine sufiksipuust.

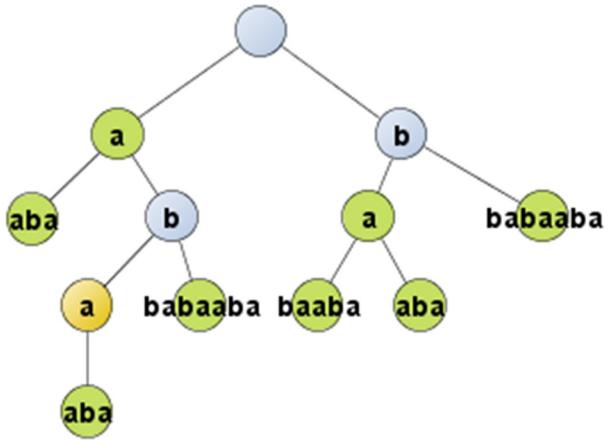
Seda teeb ka järgnev kood:

```
using namespace std;
struct Tipp {
	char* voti;
	map<char, Tipp> alamad;
};

int leia_alamstring(Tipp *tipp, char* str, int indeks) {
	if (tipp == NULL) {
		return -1; // ei leidnud
	}
	int vas = -1;
	//kui tipp t ei ole juur
	if (tipp->voti != 0)
	{
		int vastus = kontrolli(str, indeks, tipp->voti);
		if (vastus != 0)
			return vastus;
	}
	//suurenda indeksit
	indeks += strlen(tipp->voti);
	//Kui on õige algusega väljuv serv
	if (tipp->alamad.find(str[indeks]) != tipp->alamad.end())
		return leia_alamstring(&(tipp->alamad[str[indeks]]), str, indeks);
	else
		return -1;  // ei leidnud
}


int kontrolli(char *str, int indeks, char *voti)
{
	if (str[indeks] == '\0')
		return 1;  //korras

	//kontrollime, kas on kattuvus
	for (int i = 0; i < strlen(voti); i++, indeks++)
	{
		if (voti[i] != str[indeks])
			return -1;  // ei kattu
	}
	return 0;  // siiani korras, vaatame edasi
}
```

Alamstringide leidmine on keerukusega $O(m + k)$, kus m on alamstringi pikkus ja k alamstringi esinemise arv tekstis. Pane tähele, et keerukus ei sõltu üldse teksti pikkusest!

11.4.3.2 Pikim korduv alamstring

Pikim korduv alamstring on selline alamstring, mis esineb tekstis vähemalt kaks korda. Kui meil on olemas sufiksipuu, siis tuleb leida kõige sügavamal asuv hargnemine:

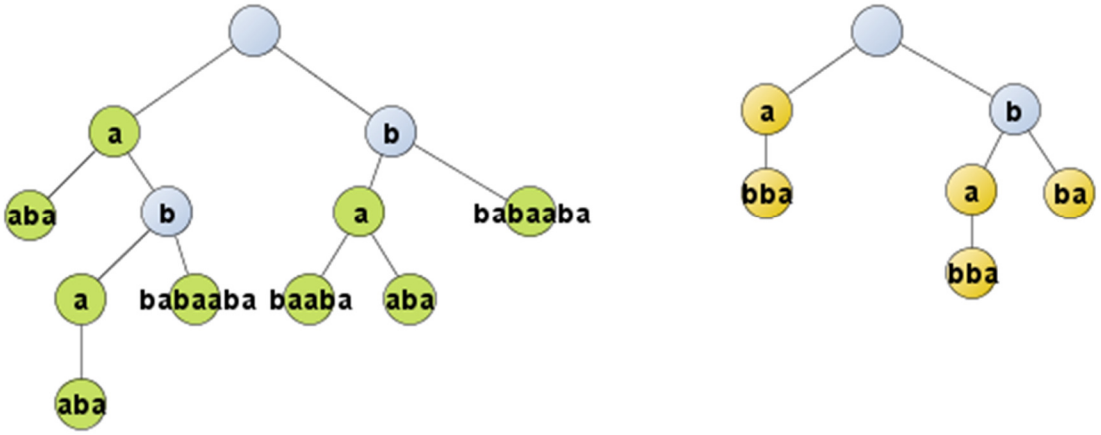



```
void leia_pikim_korduv(Tipp *tipp, int sygavus, int* max_sygavus,
      int* algus)
{
      if (tipp == NULL) {
            return;
      }
      if (tipp->voti == '\0') { //sisemine tipp
            for (int i = 0; i < MAX_CHAR; i++) {
                  if (tipp->alamad.find(str[indeks]) != tipp->alamad.end()) {
                        otsi_pikim_korduv(tipp->alamad[i], sygavus +
                              strlen(tipp->voti), max_sygavus,
                              algus);
                  }
            }
      }
      else if (tipp->voti > 0 &&
            (*max_sygavus < sygavus - strlen(tipp->voti)))
      {
            *max_sygavus = sygavus - strlen(tipp->voti);
            *algus = tipp;
      }
}

void otsi_pikim_korduv()
{
      int max_sygavus = 0;
      int indeks = 0;
      leia_pikim_korduv(puu, 0, &max_sygavus, &indeks);

      int k;
      for (k = 0; k < max_sygavus; k++)
            cout << tekst[k + indeks];
      cout << endl;
}
```

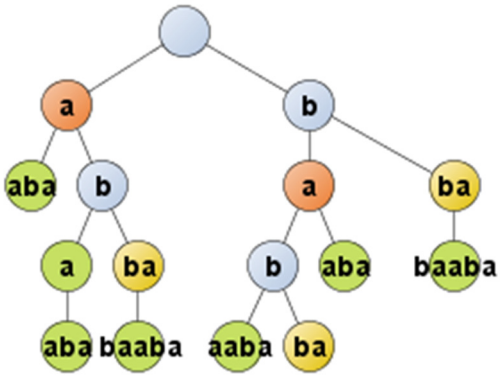
Selle lähenemise keerukus on $O(n)$.

11.4.3.3 Pikim ühine alamstring

Ehitame mõlema stringi ühise sufiksipuu. Ainsaks nipiks on, et tuleb eristada kuidagi, mis sufiksid millisesse puusse kuuluvad. Kõige sügavamal asuv hargnemine, mis on prefiksiks mõlemale (või kõigile) stringidele, ongi vastuseks:



Stringide 'abbabaaba' ja 'babba' sufiksipuud



Ja ühendatud sufiksipuu.

Pikim ühine alamstring on 'abba'.

Selle leidmise keerukus on $O(n)$.

11.4.4 Sufiksipuu loomine

Sufiksipuu on väga kasulik struktuur, kuid puu loomine ise ei ole just kõige lihtsam, kuigi see on teostatav keerukusega $O(n)$. Samas kuna võistlusel on oluline ka koodi kirjutamisele kuluva aja kokkuhoid, siis jõuame järgmise struktuuri juurde.

11.5 SUFIKSILOEND

Kuna sufiksipuu loomine ise on keeruline ning probleemiks on ka see, et puu on võrreldes tekstiga väga ruumimahukas, siis sageli kasutatakse selle asemel **sufiksiloendit** (i.k. *suffix array*).

Sufiksiloendi leiutasid Udi Manber ja Eugene Myers aastal 1990.

Vaatame taas tuttava stringi ’abbabaaba’ sufikseid:

Sufiksi number	Sufiks
1	abbabaaba
2	bbabaaba
3	babaaba
4	abaaba
5	baaba
6	aaba
7	aba
8	ba
9	a

Sorteerime need tähestiku järjekorras:

Sufiksi number	Sufiks
9	a
6	aaba
7	aba
4	abaaba
1	abbabaaba
8	ba
5	baaba
3	babaaba
2	bbabaaba

Sellist loendit nimetataksegi sufiksiloendiks.

11.5.1 Loomine

Sufiksiloendi saab konstrueerida otse sufiksipuust seda laiuti läbides. Ajakulu on vaid $O(n)$. Aga selle jaoks peab muidugi olemas olema sufiksipuu, mis võtab palju ruumi ning mille loomise algoritm on küllaltki keerukas. Seega ei ole selline sufiksiloendi koostamine kuigi praktiline.

Esimese ideena tuleb vast pähe, et leiame kõik sufiksid (lihtne ju!) ning sorteerime ära. Sorteerimise keerukus on ju $O(n\ log\ n)$. Siin tuleb aga tähele panna, et kuna kahe pika sufiksi võrdlemine ei võta mitte $O(1)$ vaid $O(n)$, siis selline triviaalne meetod on keerukusega $O(n^2\ log\ n)$.

Selle asemel sorteerime sufiksid ainult esimese tähe järgi, seejärel esimese kahe tähe järgi, siis esimese nelja tähe järgi jne. Kui meil on sorteeritud sufiksid 2^iesimese tähe järgi, siis saame ühe võrdlemisega paari kohta sorteerida need esimese 2^{i+1}tähe järgi.

Tavalist sorteerimist kasutades on sellise lähenemise keerukuseks sufiksipuu loomisel $O(n\ log^2 n)$. Enamasti on see võistlustel piisav ning koodikirjutamise ja testimise aja kokkuhoiu mõttes optimaalne. Kui aga on vaja veelgi kiiremini, siis saab kasutada kiiremat sorteerimisalgoritmi.

Siin on funktsioon sufiksiloendi loomiseks koos kiire sorteerimisega:

```
string tekst, sona;
int* sufiksid, *tmpS;
int* kohad, *tmpK;
int loendur[256];

void sorteeri(int k, int n) {
    memset(loendur, 0, sizeof(loendur)); //nullime loenduri
    for (int i = 0; i < tekst.length(); i++) {
        if(i + k < n) loendur[kohad[i + k]]++; //loendame
          else loendur[0]++;
     }
    int sum = 0;
    for (int i = 0; i < 256; i++) {
        int t = loendur[i];
        loendur[i] = sum; //kumulatiivsed algused
        sum += t;
    }

    for (int i = 0; i < tekst.length(); i++) {
          if (sufiksid[i] + k < n)
               tmpS[loendur[kohad[sufiksid[i] + k]]++] = sufiksid[i];
          else
               tmpS[loendur[0]++] = sufiksid[i];
     }

    for (int i = 0; i < tekst.length(); i++) {
        sufiksid[i] = tmpS[i];
    }
}
```

```
void loo_sufiksiloend(int n) {

    sufiksid = new int[n];
    kohad = new int[n] {};
    tmpS = new int[n] {};
    tmpK = new int[n] {};
     for (int i = 0; i < n; i++) {
        sufiksid[i] = i; //kõik sufiksid, igast indeksist alates
        kohad[i] = tekst[i]; //algsed kohad
    }
     for (int k = 1; k < n; k <<= 1) {
          sorteeri(k, n);
          sorteeri(0, n);
          tmpK[sufiksid[0]] = 0;
        int r = 0;
        for (int i = 1; i < n; i++) {
            if (kohad[sufiksid[i]] != kohad[sufiksid[i - 1]] ||
                kohad[sufiksid[i] +k] != kohad[sufiksid[i-1] + k]) {
                r++;
            }
            tmpK[sufiksid[i]] = r;
        }
        for (int i = 0; i < tekst.length(); i++) {
            kohad[i] = tmpK[i];
        }
        if (kohad[sufiksid[n - 1] == n - 1]) break;
    }
}
```

11.5.2 Sufiksiloendi kasutamine

11.5.2.1 Alamstringi leidmine

Alamstringi otsimisel sufiksiloendist kasutame sama omadust, et iga otsitav string, mis tekstis leidub, on mingi selle teksti sufiksi prefiksiks. Kuna sufiksiloendis on sufiksid sorteeritud, siis saame kasutada kahendotsingut. Selle keerukuseks on $O(m * log\ n)$, kus m on alamstringi pikkus ja n teksti (ja ka sufiksiloendi) pikkus.

```
int leia_prefiks(string str)
{
     int m = str.length();  // otsitava stringi pikkus

     //kahendotsing
     int vasak = 0, parem = tekst.length()-1;
     while (vasak <= parem)
     {
          int mid = vasak + (parem - vasak) / 2; // leiame keskmise
          char* prefiks = new char[m];

          int res = strncmp(str, tekst + sufiksid[mid], m);

          // leitud
          if (res == 0)
          {
               return suffArr[mid];
          }

          // Otsi vasakult
          if (res < 0) parem = mid - 1;

          // otsi paremalt
          else vasak = mid + 1;
     }

     return -1; ei leidnud;
}
```

Seegi on enamasti piisav, kuid vajadusel saab sedagi kiiremini teha, kui luua abitabel, kus on kirjas, mitu märki on võrreldavate sufiksite alguses samad. Siis saab võrdlemist alustada juba sellest positsioonist, kust vajalik.

11.6 Räsi

Veel üks stringitöötluses sageli vajaminev operatsioon on kahe stringi võrdsuse kontroll. Kui juba stringide esimesed märgid on erinevad, on see operatsioon kiire, aga mida teha juhul, kui esimene erinev märk on alles tuhandendal positsioonil? Sel juhul saab arvutada välja stringide räsid (i.k. *hash*) ja hoopis neid omavahel võrrelda.

Räsifunktsioon on funktsioon, mis seab mingitele andmetele, sageli suvalise pikkusega, vastavusse mingi väärtuse ehk räsi, mille pikkus ja formaat on fikseeritud.

Ideaalis saaksid samad andmed sama räsi ja vastupidi- ühele räsile vastavad samad andmed. Kuna aga räsid on üldjuhul kompaktsemad kui andmed ise, esineb olukordi, kus erinevatele andmetele võib vastata sama räsi. Sellist olukorda nimetatakse **kokkupõrkeks** ehk **kollisiooniks**.

Põhimõtteliselt võib räsifunktsioon olla milline iganes, aga hea räsifunktsioon peab olema kiiresti arvutatav ning ei tohi tekitada palju kollisioone.

Vaatame näiteks stringi $S = s_1 s_2 s_3 \dots s_n$. Milline võiks olla sellele vastav räsifunktsioon? Üheks lihtsaks funktsiooniks on kokku liita iga tähemärgi kood, kuid sellisel juhul tekib palju kokkupõrkeid, sest näiteks stringid 'OAR' ja 'MAT' saaksid sama räsi.

Enamasti on tekstitöötlusalgoritmide juures oluline ka märkide järjekord stringis, seega tuleks valida kindlasti selline räsifunktsioon, mis arvestab tähemärgi positsiooni.

Üheks heaks funktsiooniks on

$$f(S) = \sum_{i=0}^{n-1} s_i * p^i$$

kus p on vabalt valitud arvud.

Vältimaks kokkupõrkeid võiks p olla suurem kui märkide arv kasutatavas tähestikus.

11.6.1 Alamstringi leidmine räsi abil

Vaatame aga, mis kasu on räsist alamstringide leidmisel. Oletame, et meil on tekst $S = s_1 s_2 s_3 \dots s_n$ ja me soovime leida sellest alamstringi $A = a_1 a_2 a_3 \dots a_k$ Otsitava alamstringi räsi saame lihtsalt välja arvutada: $f(A) = \sum_{i=0}^{k-1} a_i * p^i$, kuid kuidas leida seda tekstist? Oletame, et otsitav alamstring algab tekstis positsioonilt u ja lõpeb positsioonil v. Sellisel juhul

$$f(A) = f(S, 0, v-u) = \sum_{i=0}^{v-u} s_{i+u} * p^i$$

Sellest üksi ei ole palju kasu. Aga pane tähele, et

$$f(S, 0, v) = \sum_{i=0}^{v} s_i * p^i$$

ja

$$f(S, 0, u-1) = \sum_{i=0}^{u-1} s_i * p^i$$

Ning nende vahe on:

$$f(S, 0, v) - f(S, 0, u-1) = \sum_{i=0}^{v} s_i * p^i - \sum_{i=0}^{u-1} s_i * p^i = \sum_{i=u}^{v} s_i * p^i = f(A) * p^u.$$

Sellest on muidugi kasu vaid siis, kui peame meeles kõigi tekstis esimesest positsioonist algavate alamstringide räsid. Samas teksti räsi arvutades on see vaid meelespidamise kulu. Vaatame ka pisikest näidet. Oletame arvutuste lihtsuse mõttes, et meie tähestik koosneb neljast märgist (a=1,b=2,c=3,d=4), sellisel juhul võime valida p=5. Oletame, et meil on tekst S = abcab.

$$\begin{aligned} f('abcab') &= a * 5^0 + b * 5^1 + c * 5^2 + a * 5^3 + b * 5^4 \\ &= 1 * 1 + 2 * 5 + 3 * 25 + 1 * 125 + 2 * 625 = 1461 \end{aligned}$$

Arvutuste käigus saame ka kõik prefiksid:

A	Ab	abc	Abca	abcab
1	11	86	211	1461

Leiame nüüd 'ca' räsi kõigepealt tabelist

$$\frac{f('abca') - f('ab')}{5^2} = \frac{211 - 11}{25} = 8.$$

Ja kontrolliks otse:

$$f('ca') = c * 5^0 + a * 5^1 = 3 * 1 + 1 * 5 = 8$$

11.6.2 Räsi moodularvutusega

Kuna aga päris tekstid võivad olla pikad, siis selliselt arvutatud räsi läheb väga suureks. Selle vältimiseks võetakse tihti räsi leidmisel jääk mingi suure arvuga jagamisel ehk mooduli m järgi.

Mooduli m valikul tuleb silmas pidada, et see oleks küllaltki suur, et vähendada kokkupõrkeid, samuti võiks m olla algarv. Eelpool toodud räsifunktsioon omandab siis kuju

$$f(S) = \left(\sum_{i=0}^{n-1} s_i * p^i\right) mod\ m,$$

mis omakorda on samaväärne

$$f(S) = \left(\sum_{i=0}^{n-1} (s_i * p^i) mod\ \ m\right) mod\ m.$$

Arvutamine moodulitega:

1. $a\ mod\ m = a,\ \ kui\ 0 \leq a < m$
2. $(a\ mod\ m)\ mod\ m\ =\ a\ mod\ m$
3. $(a + b)\ mod\ m\ =\ [(a\ mod\ m) +\ (b\ mod\ m)]\ mod\ m$
4. $(a - b)\ mod\ m\ =\ [(a\ mod\ m) - (b\ mod\ m) + m]\ mod\ m$
5. $(ab)\ mod\ m\ =\ [(a\ mod\ m)(b\ mod\ m)]\ mod\ m$
6. $a^b\ mod\ m\ =\ (a\ mod\ m)^b\ mod\ m$
7. $(a \div b)\ mod\ m = [(a\ mod\ m) \cdot (b^{-1}\ mod\ m)]\ mod\ m$, kus b^{-1} on arvu b modulaarne

Uurime, kuidas muutub meie eelpool toodud arvutuskäik, kui leiame räsi mooduli m = 101 järgi. (Nii väikese mooduli valime muidugi arvutuste näitlikustamiseks.)

Liitmine, lahutamine, korrutamine ja isegi astendamine eriti probleeme ei tekita, kuid mis saab jagamisega? Üheks võimaluseks on leida modulaarne pöördväärtus (vt meeldetuletuseks vajadusel arvuteooria peatükki) ehk selline arv p', mille korral $p * p' \equiv 1\ (mod\ m)$.Kui $p = 5$, nagu meie näites, siis sobib $p' = 81$, sest

$$5 * 81\ (mod\ 101)\ =\ 405(mod\ 101)\ =\ 1.$$

Sellisel juhul saab prefiksite tabel mooduli 101 järgi järgmise sisu:

A	Ab	Abc	Abca	abcab
1	11	86	9	47

Arvutame nüüd tabeli põhjal moodularvutuse reegleid kasutades stringi $'ca'$ räsi:

$$f('ca') = \big(f('abca') - f('ab')\big) * (p')^2 \ (mod\ 101) = \big((9-11) * 81^2\big)(mod\ 101)$$
$$= (9-11) mod\ 101 * 81^2\ mod\ 101 = 99 * 97\ mod\ 101 = 8.$$

11.6.3 Jagamise vältimine

Kuna aga modulaarse vastandarvu leidmine pole triviaalne ja nõuab ka kas koodikirjutamist või kusagilt järele vaatamist, siis üheks võimaluseks on ehitada räsifunktsioon ja prefiksitabel üles nii, et jagamise asemel saame kasutada korrutamistehteid. Näiteks kasutades räsifunktsioonina

$$f(S) = (\sum_{i=0}^{n-1} s_i * p^{n-1-i})\ mod\ m.$$

Prefiksite arvutamine läheb nüüd veidi põnevamaks, kuna iga prefiksi pikkus on ju erinev. Esimene prefiks a on ikka $a * 5^0 (mod\ 101) = 1$. Järgmine, $f('ab') = a * 5^1 + b * 5^0 (mod\ 101) = 7$ ja kolmas, $f('abc') = a * 5^2 + b * 5^1 + c * 5^0 (mod\ 101) = 38$. Selline ükshaaval algusest peale arvutamine on muiudgi liiga ajamahukas, aga taas saab arvutada järgmise prefiksi räsi eelmist väljaarvutatud räsi kasutades. Pane tähele, et

$$f('abc') = a * 5^2 + b * 5^1 + c * 5^0 (mod\ 101) = 5(a * 5^1 + b * 5^0) + c * 5^0\ (mod\ 101)$$
$$= 5(a * 5^1 + b * 5^0)\ mod\ 101 + \ c * 5^0\ (mod\ 101)$$
$$= (5 * f('ab') + c)\ (mod\ 101)$$

Siin on taas uus prefiksite räsiväärtuste tabel:

A	Ab	Abc	Abca	abcab
1	7	38	90	48

Arvutus tabelist:

$$f('ca') = f('abca') - f('ab') * p^k (mod\ 101) = 90 - 7 * 5^2 = 16.$$

Veendume, et sama tulemuse saame ka otse arvutades:

$$f('ca') = c * 5^1 + a * 5^0 (mod\ 101) = 16.$$

11.6.4 Rabin-Karpi algoritm

Ülesannetes on enamasti vaja leida tekstist S alamstring A. Vaatame nüüd, kuidas seda teha kasutades räsi.

Jõumeetodiga lahendades peaksime hakkama võrdlema algusest peale kõigepealt s1 ja a1. kui need langevad kokku, siis s2 ja a2 jne. Kui kõik langevad kokku, oleme leidnud ühe alamstringi esinemise

tekstis. Kui mingil kohal i s_i <> a_i, siis hakkame otsima teksti järgmisest positsioonist ja võrdleme s2 ja a1 jne. Selle keerukuseks on O(n*m), kus n on teksti pikkus ja m ostitava alamstringi pikkus.

Räsi abil otsimine käib tegelikult samamoodi, aga selle asemel, et võrrelda alamstringi tähemärke ükshaaval, saab arvutada välja räsid ja võrrelda neid.

Esmalt arvutame alamstringi A räsi f(A). Seejärel arvutame teksti esimese m märgi räsi f(S,0,m-1). Kui need on võrdsed, oleme tõenäoliselt leidnud alamstringi tekstist. Kui mitte, siis kokkulangevust kindlasti ei ole ja meil on vaja leida järgmise n pikkusega alamstringi räsi tekstist ehk f(S, 1, m). Kui me peaksime kogu räsi uuesti välja arvutama, siis mingit võitu võrreldes jõumeetodiga ei tule. Parem on kasutada eelpool toodud prefiksite ettearvutamist või kui otsimine on ühekordne, siis libistada räsi mööda teksti: eemaldame eest esimese märgi väärtuse ja lisame lõppu uue märgi väärtuse. Nii saame rulluda läbi kogu teksti ja leida kõik kattuvused. Sellist meetodit alamstringi leidmiseks nimetatakse **Rabin-Karp**i algoritmiks ja selle mõtlesid välja Richard M. Karp California Berkeley Ülikoolist ja Michael O. Rabin Harvardi Ülikoolist 1987. aastal.

Vaatame, kuidas leida stringi ’ca’ tekstist ’abcab’. f(’ca’) = 16. Arvutame kõigepealt tavapärasel moel stringi ’ab’ räsi: f(ab) = 7. Kuna $16 \neq 7$, siis kokkulangevust kindlasti pole ja leiab järgmise räsi:

$$f('bc') = \big((7 - 1 * 5 * \ mod\ 101) * 5 + \ 3\big)\ mod\ 101 = 13$$

Kuna ka $16 \neq 3$, saame asuda järgmist räsi arvutama:

$$f('ca') = \big((13 - 2 * 5 * \ mod\ 101) * 5 + \ 1\big)\ mod\ 101 = 16$$

Kokkulangevuse korral kontrollime üle, kas on tegelik kokkulangevus või räsi kollisioon. Antud juhul on tegelik kokkulangevus.

11.6.5 Ülesanne

Mõnikord on vaja otsida ja /või loendada etteantud tekstidest mingeid teatud pikkusega sõnu: näiteks leida lepingutest etteantud isikukoode või teatud telefoninumbreid vms.
Sulle on antud tekst T ja n stringi, igaüks pikkusega k. Leia iga stringi kohta, mitu korda see tekstis T esineb.
Sisendi esimesel real on kaks täisarvu n – otsitavate stringide arv ja k – otsitavate stringide pikkus.
Järgmisel n real on igaühel string pikkusega k. Viimasel real on tekst T.
Väljundisse kirjutada n rida, kus iga real on otsitav string ja selle esinemissagedus tekstis T.
NÄIDE:

```
5 4
koer
hiir
kass
kukk
kana
kass ronis puu otsa ja kukkus alla koer viisakana aitas kassi.
```

Vastus:

```
koer 1
hiir 0
kass 2
kukk 1
kana 1
```

Kasutame selle lahendamiseks Rabin-Karpi algoritmi. Kuna antud ülesandes on paring ühekordne ja sõnad ette teada, siis tegelikult ei ole vaja kõiki räsisid alles hoida: võime lihtsalt teksti algusest räsid arvutada ja neid kohe otsingusõnade räsidega võrrelda. Aega see oluliselt kokku ei hoida, kuid ruumi küll.

```
#define ull unsigned long long
using namespace std;

char* tekst; // pikk tekst
ull m = 1000000007; // suur algarv jäägi jaoks
ull p = 256; // märkide arv
int k, n; //k-otsisõne pikkus, n-otsisõnede arv

//hoiame otsisõnu dünaamilises massiivis paaridena, kus esimeseks paaris on sona
//esinemissagedus, teisel kohal paar, milles esimesel kohal on sona räsi ja teisel
//kohal lõpuks sõna ise.
vector<pair<int, pair<ull, char*>>> otsisonad;

ull arvuta_rasi(char *str) { //arvutab k pikkusega räsi
    ull rasi = 0LL;
    for (int i = 0; i < k; i++) {
        rasi = (p*rasi + str[i]) % m;
    }
    return rasi;
}
```

```
void loenda_sonad()
{
    int teksti_pikkus = strlen(tekst);
    int teksti_rasi = arvuta_rasi(tekst); // esimese räsi väärtus teksti algusest
    int h = 1;

    // h väärtus on pow(d, k-1)%q"
    for (int i = 0; i < k - 1; i++)
        h = (h*p) % m;

    // libistame räsi
    for (int i = 0; i <= teksti_pikkus - k; i++)
    {
        for (int ii = 0; ii < n; ii++) { //iga otsisõna jaoks
            if (otsisonad[ii].second.first == teksti_rasi) { //kui on kattuvus
                int j = 0;
                for (j; j < k; j++)                    {
                    if (tekst[i + j] != otsisonad[ii].second.second[j])
                        break; // kollisioon
                }

                if (j == k) //kui kollisiooni polnud ja oli tegelik kattuvus
                    otsisonad[ii].first++; //sona leitud!
            }
        }
        // arvutame järgmise räsi eemaldades esimese tähe ja lisades järgmise
        if (i < teksti_pikkus - k)
        {
            teksti_rasi = (p*(teksti_rasi - tekst[i] * h) + tekst[i + k]) % m;

            if (teksti_rasi < 0) //kui jääk on negatiivne
                teksti_rasi += m; //teeme positiivseks
        }
    }
}
```

11.7 KONTROLLÜLESANDED

11.7.1 Telefoninumbrid

Kunagi ammustel aegadel, kui loomad veel rääkisid ning inimesed taevas lendasid, kasutati selliseid riistapuid, mida nimetati klahvidega telefonideks. Nendel telefonidel oli 10 numbriklahvi, millest osad võisid ka tähti tähistada järgneva skeemi põhjal:

```
2-ABC
3-DEF
4-GHI
5-JKL
6-MNO
7-PQRS
8-TUV
9-WXYZ
```

Siis taibati, et mingeid telefoninumbreid saab meelde jätta, asendades võimalusel numbri ühe tähega nende hulgast, mis vastavad sellele numbrile. Näiteks võib numbrile 340833 vastata tähekombinatsioon EI0TEE. Sulle on nüüd antud üks sarnasel viisil kirjutatud telefoninumber, mille pikkus ei ületa 30 tähemärki ning mis koosneb vaid suurtähtedest ning numbritest 0 ja 1, ning sa pead väljastama sellele vastava telefoninumbri.

NÄIDE 1:

EI0TEE

Vastus:

340833

NÄIDE 2:

MUKODUKEONTILLUKE1

Vastus:

685638536684558531

NÄIDE 3:

MINUARMASKASS

Vastus:

6468276275277

11.7.2 Enigma

Teise maailmasõja ajal oli sakslaste Enigma kodeerimismasina murdmine väga oluline samm, mis lubas liitlastel lugeda Saksa sõjaväe teateid. Teatavasti oli iga Enigmaga saadetud sõnumi pikkusega n ees n fibonacci arvu, millele järgnes suvalise pikkusega tekst, mis koosnes erinevatest märkidest, kuid milles oli täpselt n suurtähte. Enigma dešifreerimiseks pidi alguses lihttekstist kõik sümbolid, mis ei olnud suurtähed, välja viskama. Originaalse teksti saamiseks tuli vaadata kohal `i` olevat fibonacci arvu, mis on fibonacci jadas kohal `j`. See tähendas, et täht kohal `i` on originaaltekstis kohal `j`. Antud krüpteering eeldab, et fibonacci jada esimene arv on 1, teine on 2 ning `f(n+1) = f(n) + f(n-1)`, kus `f(x)` tähistab kohal x olevat fibonacci arvu. Kui mingile kohale `j` ei vasta ükski antud fibonacci arv, kuid eksisteerib koht k, kus `k > j` ning k-le vastab mingi fibonacci arv, siis oli sellel kohal originaaltekstis tühik. Näiteks sõnumit, mis koosnes jadast `317 987 1587 8 2 13 144 610 34 3 1 89 55` ning lausest „`Jah, MA OlEN RASkE SUU.`“, sai dekodeerida, eemaldades kõigepealt kõik märgid, mis ei ole suurtähed, et saada `JMAOENRASESUU.` Kuna `317` on 13. fibonacci arv, läheb täht `M` kohale 13. Protsessi jätkates, saame sõnumi `SEE ON SUUR JAMA.` Paneme tähele, et tühikud tekivad sellel põhjusel, et neile vastavaid arve (5, 21 ja 233) ei ole kasutusel. Sulle on ette antud üks Enigma abil saadetud sõnum, mille pikkus ei ületa 100 tähemärki. Leia, mis on originaaltekst.

Sisendi esimesel real on arv n - fibonaccide arvude hulk. Järgmisel real on n fibonacci arvu. Kolmandal real on saadetud tekst.
Väljundisse kirjutada dešifreeritud tekst.

NÄIDE 1:

```
13
317 987 1587 8 2 13 144 610 34 3 1 89 55
Jah, MA OlEN RASkE SUU.
```

Vastus:

```
SEE ON SUUR JAMA
```

NÄIDE 2:

```
14
55 13 144 89 5 1597 1 377 987 3 2 34 610 21
```

Vastus:

```
GLooBUs VALITsEs AIa TagaplaanIl
```

11.7.3 Universaalgeen

On olemas tulnukate rass, kelle genoomi kirjeldamiseks on vaja kasutada kõiki väikesi ladina tähti. Need tulnukad otsustasid otsida nn universaalgeeni, mis on defineeritud kui genoomis kõige tihedamini esinev alamstring pikkusega `n` (`0<n<11`). Leia see universaalgeen.
Sisendi esimesel real on arv `n`. Järgmisel real on string, mille pikkus on maksimaalselt 1000 – tulnukate genoomi kirjeldus.
Väljundisse kirjutada üks string: kõige tihedamini esinev alamstring. Kui vastuseid on mitu, väljasta neist leksikograafiliselt vähim.

NÄIDE 1:
```
3
baababacb
```
Vastus:
```
aba
```

11.7.4 Arelofansi keel

Arelofansi keel on austroneesia keelkonda kuuluv keel, mida kõneldakse peamiselt Arelofansi saarel Indoneesias ning mida kõneleb umbes 100-150 inimest. Keele üks omapäradest on see, et täishäälikutel vahet ei tehta, mistõttu eurooplased, kes on varem üritanud keele omapärasid kirja panna, ei suutnud kokku leppida sõnade tõlgetes. Aidake neid.
Sisendis on antud kahel real kaks sõna arelofansi keeles. Sõnade pikkused ei ületa 100 tähemärki. Leida, kas antud sõnad on sama tähendusega või mitte. Antud juhul on kaks sõna sama tähendusega parajasti siis, kui arelofansi keele kõneleja teeb neil vahet.
Väljundisse kirjutada üks sõna. Kui sõnad on sama tähendusega, väljastada `SAMA`, kui erinev, siis `ERINEV`.

NÄIDE 1:
```
arelofansi
erilufenso
```
Vastus:
```
SAMA
```
NÄIDE 2:
```
ai
ei
```
Vastus:
```
SAMA
```
NÄIDE 3:
```
mina
nina
```
Vastus:
```
ERINEV
```
NÄIDE 4:
```
saen
san
```
Vastus:
```
ERINEV
```

11.7.5 Koodigeneraator

Tänapäeval on olemas mitmeid keskkondi, kus inimene saab programmeerimist harjutada, saates ülesandeid testerile testimiseks. Paljud saidid aga kontrollivad vaid näidistesti lahendamise õigsust, mistõttu töötab ka programm, mis tagastab vaid näidisväljundi. Sinu eesmärk on katsetada, kas selline programm läheb läbi, kuid kuna ülesandeid on mitu, otsustasid teha oma koodigeneraatori. Näidisväljund asub `n (0 < n < 100)` real. Sinu programm peab vastama kujule

```
#include<string.h>
#include<stdio.h>
int main()
{
<koodiread>
printf(„\n“);
return 0;
}
```

kus `<koodiread>` asemel peab olema n rida kujul `printf(„<rida>\n“);` kus `<rida>` asemel on vastav rida näidisväljundis.
Sisendi esimesel real on arv n. Järgmisel n real on tekst, mille pikkus ei ületa 1000 tähemärki - näidisväljund.
Väljundisse kirjutada näidisväljundit kirjutav programm.

NÄIDE:

```
2
Mina Olen Tubli Tudeng
M.O.T.T.
```

Vastus:

```
#include<string.h>
#include<stdio.h>
int main()
{
printf(„Mina Olen Tubli Tudeng \n“);
printf(„M.O.T.T.\n“);
printf(„\n“);
return 0;
}
```

11.7.6 Automaathelistaja

Uus äpp lubab sul valida telefoninumbri nii, et kui ekraanil olev number on sinu kontaktnimekirjas olemas, hakkab see kohe antud numbrile helistama. See võib aga probleeme tekitada, sest su sõbra Juku telefoninumber on 11234478, mistõttu ühendatakse sind häirekeskusega, enne kui Juku numbrini jõuad. Nüüd mõtled sa, kas Jukul oleks sellisest äpist kasu, või on tal ka olemas selline numbrite paar, millest ühe valimisel annab see äpp poole pealt teise.
Sisendi esimesel real on täisarv `n` (`0<n<10001`) – Juku telefoniraamatu suurus. Järgmisel `n` real on `n` kuni kümnekohalist telefoninumbrit.
Väljundisse kirjutada üks sõna. Kui Jukul oleks äpist kasu, väljastada JAH, kui mitte, väljastada `EI`.

NÄIDE 1:

```
3
911
97625999
91125426
```

Vastus:

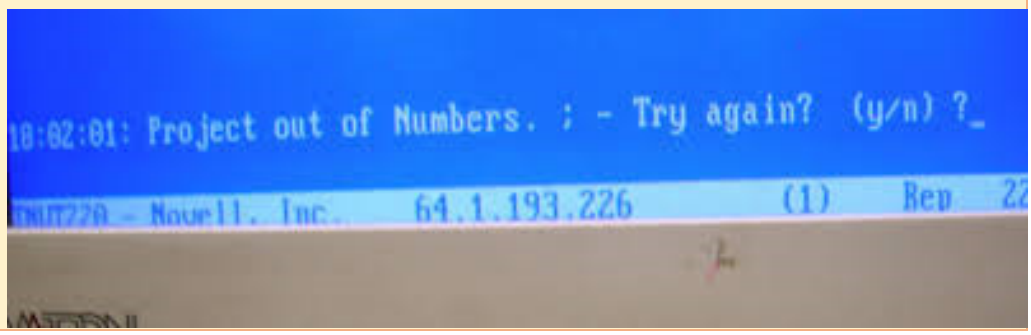
```
EI
```

NÄIDE 1:

```
5
113
12340
123440
12345
98346
```

Vastus:

```
JAH
```



11.7.7 Risti-rästi

Juhal on `n*m` tabel (`0 < n, m < 51`), kus igas lahtris on mingi ladina täht. Ta peab sealt tabelist üles otsima `k` (`0 < k < 21`) sõna. Sõna võib olla tabelis kirjutatud sirgjooneliselt mistahes kaheksast võimalikust suunast. Suur- ja väiketähed ei mõjuta sõna korrektsust. Näiteks kui otsitav sõna on `LEGO`, aga tabelis on `lego`, on tabelis olev sõna õige. Leida kõikide otsitavate sõnade esimese tähe koordinaadid tabelis. Esimene koordinaat tähistab rea numbrit, teine tähistab märgi numbrit. Kui otsitavat sõna esineb tabelis mitu korda, tuleb valida võimalikest koordinaatidest kõige väiksema reanumbriga ning seejärel kõige väiksema veerunumbriga koordinaat.
Sisendi esimesel real on täisarvud `n` ja `m`. Järgmisel `n` real on sõned pikkusega `m` – tabeli read. Siis tuleb üksikul real arv `k`. Järgmisel `k` real on otsitavad sõnad.

NÄIDE 1:

```
8 11
abcDEFGhigg
hEbkWalDork
FtyAwaldORm
FtsimrLqsrc
byoArBeDeyv
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
4
Waldorf
Bambi
Betty
Dagbert
```

Vastus:

```
2 5
2 3
1 2
7 8
```

11.7.8 Ahvitembud

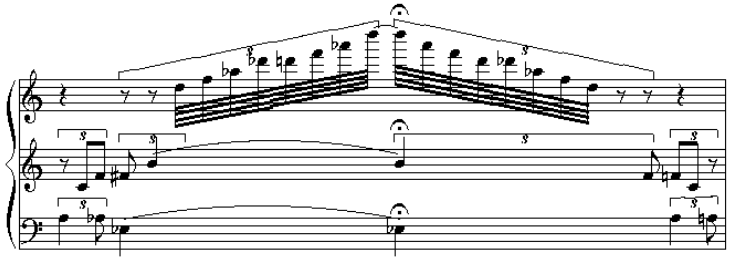
Kuulus zooloog Jane Goodall pani tähele, et päeva jooksul teevad ahvid maksimaalselt 26 erinevat tegevust, kuid iga ahv ei pea neid tegema samas järjekorras. Ta otsustas uurida, mis on maksimaalse pikkusega tegevuste nimekirja pikkus, mille kohta saab kindlalt väita, et mingi paar ahve teeb kõik nimekirjas antud tegevused järjekorras, st et tegevust loetakse tehtuks, kui ahv teeb parasjagu seda tegevust ning kõik varasemad tegevused nimekirjas on juba tehtud. Aidake teda. Sisendi esimesel real on kaks stringi, mille pikkused on kuni 100 tähemärki – kummagi ahvi tegevuskava.
Väljundisse kirjutada üks arv – pikima ühise tegevuskava pikkus.

NÄIDE 1:
```
abcd
acdb
```
Vastus:
```
3
```
NÄIDE 2:
```
abcd
dacb
```
Vastus:
```
2
```

11.7.9 Palindroomid

Rannarile meeldivad palindroomid. Tal on string pikkusega `n` (`0 < n < 1001`) ning ta tahab saada sellest palindroomi, lisades sellele võimalikult vähe tähti juurde. Aidake Rannarit ning leidke, mis on minimaalne vajalik tähtede arv.
Sisendi ainsal real on string pikkusega `n` – Rannari string.
Väljundisse kirjutada üks täisarv – minimaalne vajalik tähtede arv, mis on vaja lisada, et Rannaril oleks palindroom.

NÄIDE 1:
```
siga
```
Vastus:
```
3
```
NÄIDE 2:
```
ees
```
Vastus:
```
1
```



NÄIDE 3:
```
kuulilennuteetunneliluuk
```
Vastus:
```
0
```
NÄIDE 4:
```
pqrsabcdpqrs
```
Vastus:
```
9
```

11.7.10 Otsing

Andmebaasis on n märksõna (`0 < n < 20000`), mis on nummerdatud arvudega 1 kuni `n`. Iga märksõna on pikkusega kuni 100 tähemärki ning koosneb väikestest ladina tähtedest.
Otsingualgoritm peab võtma vastu stringi, mille pikkus on samuti kuni 100 tähemärki ning väljastama esimese kümne märksõna numbrid, milles on otsitav string märksõnas alamstringina. Märksõnad sorteeritakse alguses suuruse järjekorras – väiksemad enne. Sama pikkusega märksõnad sorteeritakse leksikograafiliselt ning kui andmebaasis on kaks ühesugust märksõna, tuleb enne väljastada väiksema numbriga märksõna number. Kui tingimusele vastavaid märksõnu on vähem kui kümme, tuleb väljastada kõikide märksõnade numbrid, mis vastavad antud tingimusele, ning kui tingimusele vastavaid märksõnu andmebaasis ei ole, tuleb väljastada arv `-1`. Kirjutage see otsingualgoritm.
Sisendi esimesel real on arv `n`. Järgmisel n real on andmebaasis hoitavad märksõnad. Märksõnad on antud nummerdamise järjekorras – esimene on numbriga `1`, viimane on numbriga `n`. Järgmisel real on arv `m (0 < m < 100001)` – päringute arv. Järgmisel `m` real on otsitavad stringid.
Väljundisse kirjutada `m` rida – iga päringu kohta õige arv tühikutega eraldatud numbreid.

NÄIDE 1:

```
17
tere
minu
nimi
on
kuuli
lennu
tee
tunneli
luuk
ja
ma
tulen
tartust
anne
linnast
anne
kanalist
5
a
an
jah
r
z
```

Vastus:

```
10 11 14 16 13 17
14 16 17
-1
1 13
-1
```

11.8 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk11 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
arvutus.cpp, arvutus.java, arvutus.py	Teksti parsimine
alamstring.cpp, alamstring.java, alamstring.py	Lihtne alamstringi leidmine
hulknurgad.cpp, hulknurgad.java, hulknurgad.py	Knuth-Morris-Pratti algoritm
spuu.cpp, spuu.java, spuu.py	Sufiksipuu
sloend.cpp, sloend.java, sloend.py	Sufiksiloend
rk.cpp, rk.java, rk.py	Rabin-Karpi algoritm