Eötvös Loránd Tudományegyetem
Matematika Intézet

Dandé Fanni

Titkosírások

BSc szakdolgozat

Témavezető: Sziklai Péter

ELTE Számítógéptudományi Tanszék

2017. Budapest

Köszönetnyilvánítás

Szeretném megköszönni az útmutatást témavezetőmnek, Sziklai Péternek, az ő ötletei és észrevételei nélkül ez a szakdolgozat nem jöhetett volna létre. Nagyon köszönöm családomnak és barátaimnak, akik végig támogattak és bátorítottak. Végül külön szeretnék köszönetet mondani Keresztury Bencének és Sárosdi Zsombornak a sok közös tanulásért és a folyamatos motivációért.

Tartalomjegyzék

Ábrák jegyzéke

1. fejezet

A kriptográfia alapjai

1.1. Alapfogalmak

A titkosírásokkal foglakozó tudományágat kriptográfiának nevezzük. A történelem során már az ókorban megjelent az igény egy olyan kommunikációs forma létrehozására, amely lehető teszi a két személy vagy csoport közötti információáramlást úgy, hogy az védve maradjon a külső felektől. Elsők között a hadászatban lett nagy szerepe, a háborúk alatt működő kémhálózatok által gyűjtött adatok országok sorsát dönthették el. Emellett egyre fontosabbá vált az államigazgatásban és a különböző diplomáciai területeken is. A 21. században a titkosításnak már sokkal nagyobb szerepe van a hétköznapi életében. A gazdaság, a bankok erre épülnek, illetve az elektronikus távközlés és kommunikáció területeire is kiterjed.

A titkosírások valójában két csoportra oszthatók, ezek a szteganográfia és kriptográfia. A szteganográfia célja az üzenet elrejtése, vagyis hogy annak létezéséről csak a feladó és a címzett tudjon. Ezt a módszert főként az ókorban alkalmazták, amikor a kriptográfia tudománya még nem volt annyira kifejlett. Néhány híres példáról írt műveiben a görög Hérodotosz, ilyen Demaratus viasztáblája. A spártai király figyelmezteti akarta országát Xerxész perzsa király támadásáról, ezért egy viasszal bevont tábláról lekaparta a viaszt, üzenetet írt a táblára, majd viaszt öntött rá úgy, hogy ez elrejtse az írást. Ezután már biztonságban el tudta juttatni a táblát a címzetthez. A szteganográfia egyik legismertebb példája a láthatatlan tinta, kezdetben egyszerűbb, majd a kémia fejlődésével egyre bonyolultabb eljárási és előhívási módszereket fejlesztettek ki. Ma már inkább a számítógépes kommunikációhoz köthető, gyakori előfordulásai az úgynevezett „zajos” kép, illetve a hangállományokba vagy kódrészletekbe rejtett üzenetek.

A kriptográfia a szteganográfiával szemben nem az üzenet létezésének eltitkolásáról szól, hanem annak átkódolásáról, vagyis egy, az eredetitől különböző szöveg elküldéséről. Ezt aztán címzett a birtokában lévő többletinformációval meg tudja fejteni.

A kriptográfia alapfogalmai a következők.

1. Definíció. A feltöréséhez szükséges többletinformációt a titkosírás kulcsának nevezzük.

2. Definíció. Az eredeti titkosítandó szöveget nevezzük nyílt szövegnek, ez a plaintext.

3. Definíció. A plaintext-ből egy eljárás segítségével kapjuk a titkosított szöveget, ez a ciphertext.

4. Definíció. Az előbbi eljárást során egyfajta algoritmust használtunk, amit kódolásnak vagy rejtjelezésnek hívjuk.

5. Definíció. A ciphertext plaintext-té való alakítása a dekódolás vagy visszafejtés.

A plaintext és ciphertext minden esetben valamilyen ABC-ben íródtak, melyeknek nem kell megegyezniük egymással. Sőt, ABC alatt itt érthetünk betűkből, számokból, vagy szimbólumokból álló halmazrendszereket is. Bevezetem a $pABC$ és $cABC$ jelöléseket, ahol $pABC$ a plaintext, $cABC$ pedig a ciphertext ABC-je. A gyakran használt 26 latin nagybetűből álló sorozatot nemzetközi ABC-nek nevezzük. Számítógépes titkosírásoknál a legegyszerűbb jelkészlet a kettes számrendszer, ugyanis bármilyen jelsorozat létrejön ezek kombinációjából. A titkosítás során az ABC tehát bármi lehet, a lényeg csak az, hogy azt kezdetben rögzítsük. Az ABC elemeit karaktereknek nevezzük. Mindkét szöveg üzenetegységekre tagolódik, ezen egységek lehetnek eltérő hosszúak is, ha például szavaknak különböző szimbólumokat feleltetünk meg. Ezek a részletek adják majd a dekódolás alapegységeit.

A titkosítás folyamata matematikai értelemben két függvényként írható le.

6. Definíció. A k kódoló függvény értelmezési tartománya a plaintext összes lehetséges üzenetegysége.

Továbbá fel kell tennünk a k függvény injektivitását, hiszen az egyértelmű visszafejthetőség megkívánja, hogy egy ciphertext-beli szóhoz csakis egy plaintext-beli tartozzon.

7. Definíció. A szöveg dekódolásánál a k függvény inverzét alkalmazzuk a titkosított szövegre, ezt d dekódoló függvénynek nevezzük. Ez a ciphertext lehetséges üzenetegységein van értelmezve.

Nem szükséges, hogy d értékkészlete az összes lehetséges üzenetegység legyen, elég a függvényt k értékkészletén értelmezni.

Valójában egy adott titkosírást nem egy ilyen függvénypárnak, hanem függvények egy családjának feleltetünk meg. Ez a család meghatározza milyen módszerrel titkosították a szöveget, azonban azt nem tudhatjuk konkrétan mi a titkosírás kulcsa, vagyis mi az a

paraméter ami elárulja, hogy a függvénycsalád melyik elemét kell használnunk. Legtöbb esetben, ha egy külső fél tudja is milyen módszerrel titkosítottak egy üzenet, az még nem jelenti, hogy meg is tudja fejteni azt. A kulcs megtalálása fejlettebb algoritmus esetén nagy erőfeszítéseket kíván és olyan időigényes, hogy mire azt megfejtik, az információ már könnyen elavulttá válhat.

Ezek után felmerülhet a kérdés, hogy mikor nevezünk jónak egy titkosírást. Összefoglalva egyrészt külső fél számára bizonyos időn belül megfejthetetlennek kell lennie, valamint a feladónak rövid idő alatt kell kódolnia és a címzettnek szintén rövid idő alatt tudnia kell dekódolni az üzenetet. Az előző módszerek összevonásával már biztonságosnak érezhetjük az információ átadását, azonban ezeket napjainkban már nem alkalmazzák. Az adatok áramlásának majdnem egésze elektronikus úton valósul meg, lényegében számítógépek segítségével kommunikálunk egymással. A kriptográfia tudománya ma már nem annyira a titokzatosságra, sokkal inkább a hasznosságra törekszik, így a számítógépek fejlődésével egyidejűleg az alkalmazott matematika fontos részterületévé vált. A modern titkosírás fogalmai és működése matematikai alapokra épülnek, melyek főként a számelmélet és az algebra.

1.2. A kriptográfia fajtái

A titkosírásoknak tágabb értelemben véve rengeteg fajtája van. Egy üzenetet kódolhatunk akár képekkel, mozdulatokkal vagy hangjelzésekkel, bár ezek nagy része csak bizonyos lekorlátozott információ továbbítására képes. Például egy fényjelzéssel vagy hosszú kürtszóval jelezhették, ha elesett egy vár vagy megtörtént valami más előre várt esemény. Néhány ilyen módszernek csupán történelmi jelentősége van és ma már könnyen megfejthető, viszont vannak, melyek továbbfejlesztése egy összetettebb kód alapjait adják. A kriptográfia tudományán belül általában kétféle megfeleltetést különítünk el aszerint, hogy betű helyett betűt vagy számot írunk.

Először tekintsük a betű helyett szám esetét. Itt lényegében akármilyen egyértelmű megfeleltetést alkalmazhatunk. A legegyszerűbb, amikor a $pABC$ betűin sorba menve mindegyikhez a sorszámát rendeljünk úgy, hogy az a betűnk 01 legyen, a b 02 és így tovább. Dekódoláskor a szöveg számpárokba rendezhető, melyek egyértelműen visszaadják a nekik megfeleltetett plaintext-beli elemet. Másik módszer, ha a számokat táblázatos formába írjuk, mintha egy mátrix elemei lennének, majd a szöveg kódolásánál egyszerűen egy betű helyett a mátrixban lévő helyét írjuk le. A sor és oszlopindex ismét számpárokat alkotnak, így a kódolás injektív lesz, tehát egyértelműen vissza lehet fejteni. A számítógépes kódolásnál használt általános kódkészlet az ASCII 256 karaktert tartalmaz, amelyek között szerepelnek a kis- és nagybetűk és a számjegyek. Ez minden karakternek egy bájtot

feleltet meg, a számítógép így tárolja az adatokat és utasítások, tehát tulajdonképpen ez egy közhasznú titkosítás.

A betű helyett betű esetén is rengeteg módszer áll rendelkezésünkre. A leghíresebb történelmi példa Julius Caesar titkosítása, ami arra alapult, hogy a ABC minden betűjét hárommal jobbra tolták, vagyis a hárommal utána lévő betűvel feleltettek meg. Erről a későbbiekben még külön szó fog esni. Lényegesen régebbi módszer a héber ATBAS. Érdekesség, hogy ennek éppen kultikus okai voltak, a szövegekben Jahve nevének lejegyzéséhez használták, akinek nevét tilos volt leírni. Ennek lényege a következő, az ABC első betűjét az utolsó betűvel cserélték fel, a másodikat az utolsó előttivel, és így tovább az ABC belseje felé haladva. Az sem okozott gondot, hogy az ABC éppen páros vagy páratlan sok karakterből állt, a különbség csak annyi, hogy páratlan esetben az ABC középső eleme egy fixpont. Ez előbb tárgyalt két eset a behelyettesítő kódolások csoportjába tartozik, melynek lényege, hogy a nyílt szöveg minden betűjét egy másikkal helyettesítjük. A behelyettesítés mellett alkalmazhatjuk egyszerűen a karakterek átrendezését, vagyis permutációját valamilyen szabály szerint. Ezek közül egy gyakori példa a fésűs rendezés, a betűket felváltva írjuk az első majd második sorba, majd a két sort egymás után fűzve kapjuk a titkosított szöveget. A módszer nagy előnye, hogy az üzenet hosszának növekedésével egyre csökken a megfejthetőség esélye.

További osztályozási elv szerint egy titkosítás lehet monoalfabetikus vagy polialfabetikus.

8. Definíció. Monoalfabetikusnak nevezzük az a behelyettesítési formát, amikor egy betűt egy másik lerögzített betűvel vagy szimbólummal helyettesítünk.

A héber ATBAS, a Polübiosz-négyzet (mátrix) és a Caesar-rejtjel is ebbe a kategóriába tartozik.

9. Definíció. A behelyettesítő rejtjelek közül minden olyat, amely többféle helyettesítő ABC-t használ polialfabetikus kódnak hívunk. Egy $pABC$-beli elemhez többféle $cABC$-belit is rendelhetünk egy másodlagos tulajdonság, pl a szövegbeli elhelyezkedése alapján.

Ennek egyik ismertebb példája a Vigenere-kód, amelyről szintén a későbbiekben lesz szó, illetve a módszer bonyolultabb példája a második világháború alatt használt Enigma.

Létezik egy másfajta betű helyett szám módszer, ami nem a titkosításra, hanem a kódok tömörítésére törekszik. Azonban a létrejött számsort tovább kódolhatták, így ezek tökéletes alapot adhatnak egy közel feltörhetetlen üzenet létrehozásához. A tömörítések legfőképp a számítástechnikában játszanak nagy szerepet. Rengeteg módszer létezik, amelyek arra törekszenek, hogy adott tárolókapacitás mellett minél több adatot lehessen tárolni.

1.3. Feltörési módszerek

Az monoalfabetikus helyettesítések közös tulajdonsága, hogy a plaintext egy eleméhez a ciphertext egyik elemét rendeljük hozzá, minden esetben ugyanazt, az tulajdonképpen mindegy, hogy ezek éppen betűk-e vagy számok. Ha tehát a kódolt szövegnek kiválasztjuk egy tetszőleges karakterét és arról megállapítunk bizonyos tulajdonságokat, akkor az a plaintext-beli megfelelőjére is igaz lesz. A feltörés során egy a lényeg, az adott karaktert mindig ugyanannak a karakternek feleltessük meg.

Amennyiben tudjuk milyen fajta titkosítással kódolták a szöveget, akkor a feladat a kulcs megtalálása. Ha kulcsra értelmes szöveget kapunk vissza, az azt jelenti, hogy sikerült dekódolnunk, hiszen ha véletlenszerűen felírunk egymás után betűket, annak a valószínűsége, hogy értelmes szöveget kapunk nagyon kicsi.

A ciphertext feltöréséhez elengedhetetlen, hogy kellően hosszú szöveg álljon rendelkezésünkre, illetve szükséges annak a nyelvnek az ismerete, amin a plaintext íródott. Esetünkben ez a magyar lesz. Így betűgyakoriság alapján ki lehet következtetni melyik karakternek melyik betű feleltethető meg. A magyar nyelvben (ékezet nélküli formában) a leggyakoribb betűk sorrendben a következők: $e, a, t, o, l, n, s, k, i$, a következő 9 betűnk: $r, m, z, g, d, u, v, b, h$, majd a legritkábbak: j, y, p, f, c, w, x, q.

A feltörés menete a titkosítás fajtájától függően sokféle lehet. Monoalfabetikus esetben az alábbi lépésekre bontható. Elsőként azt kell megsejtenünk, hogy miből áll a plaintext: lehet egyszerű szöveges üzenet, számok, adatok halmaza, vagy valamilyen számítógépes fájl, erre a ciphertext-beli karakterek számából tudunk következtetni. Ha 25-30 különböző karaktert látunk, akkor nagy valószínűséggel az üzenetünk egy szöveg valamelyik ABC-ben írva. Állítsunk fel egy gyakorisági sorrendet a ciphertext karakterei között, ebből következtetni lehet a feltevésünk szerinti plaintext szöveg mibenlétére. Feleltessük meg a $cABC$ leggyakoribb karakterét a $pABC$ leggyakoribb betűjével, természetesen itt több esetet is végig próbálva. Ha találunk értelmes szótöredékéket, akkor nézhetjük a következő leggyakoribb karaktereket és így tovább, amíg értelmes szöveget nem kapunk. Polialfabetikus esetben érdemes úgynevezett betörési pontokat keresni, ilyen lehet pl tagolt szöveg esetén a névelők, rövid szavak vagy ha tudjuk, hogy egy levéllel van dolgunk, akkor logikusan következtethetünk arra, hogy egy megszólítással kezdődik és elköszönéssel végződik ami alapján el lehet indulni.

1.4. Programozási feladatok

A későbbi fejezetekben programozási feladatokkal foglalkozom, a következő típusokat veszem sorra: Caesar-rejtjel, Vigenère-kód és permutációs titkosítás. Egy ilyen módszert

két külön programba szedtem aszerint, hogy a szöveget kódolni vagy dekódolni szeretnénk, illetve a Vigenère dekódoló részénél szerepel még egy továbbfejlesztett program is.

Minden esetben a szöveget egy `txt` fájlban kell megadni és eleget kell tennie bizonyos tulajdonságoknak. A feltörési folyamat kiinduló pontja a betűgyakoriságok meghatározása, ezért hosszabb szöveget adjunk meg, néhány szavas bemenetre a program nagy eséllyel nem fog értelmes eredményt adni. Továbbá a szóközöket meg kell hagyni, illetve ékezet nélkülinek kell lenni, amihez ezt az oldalt használtam:

`http://www.unit-conversion.info/texttools/remove-letter-accents/` .

A programok kizárólag betűket írnak át az adott módszer szerint miközben a nagybetűket kicsikké alakítja. A szöveg formázása, tagoltsága és egyéb karakterei változatlanul maradnak. A folyamat végén az eredményt egy a program során létrehozott `txt` fájlba mentem el. A két fájl tartalmát és a függvények főbb lépéseit a konzolra is kiírom, így a felhasználó könnyen nyomon tudja követni honnan indultunk és hogyan jutottunk a végállapotba.

A dekódolás során szükségünk van egyfajta ellenőrzésre, amely segítségével a program el tudja dönteni, hogy értelmes szöveget kaptunk-e. A Caesar- és Vigenère-titkosítás eltolásokkal dolgozik, ezért itt egy a magyar nyelv leggyakoribb 50 szavát tartalmazó szótárat használok, amely tartalmát a következő két honlap alapján állítottam össze:

`https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists/Hungarian_frequency_list_1-10000`

`http://corpus.nytud.hu/cgi-bin/mnszgyak?mode=word&ppos=all&focus=&context=c3&sort=total`

Ha a Caesar-rejtjelnél egy eltolási számra találtunk értelmes szavakat, akkor már az egész szöveget visszafejtettük, viszont a Vigenère-kódnál nem ez a helyzet. Ott több eltolási számmal dolgozunk, ezért külön ellenőrizni kell azt is, hogy minden szám helyes-e. Ez a módszer célravezető lesz, mert annak esélye, hogy értelmes szót találtunk rossz eltolási számokat alkalmazva nagyon kicsi. Ezeket az eseteket egy számláló bevezetésével kizárhatom a lehetséges megfejtések közül, hiszen egy-két értelmes szó még nagyon ritkán kaphatunk, de ennél többet csakis akkor ha a jó kulcsot használtuk. A permutációs dekódolásnál nincsen eltolási szám, itt minden betűhöz egy másikat rendelünk, ezért kicsit máshogy használom a szótárat, amit majd az erre vonatkozó részben fejtek ki részletesen.

2. fejezet

A Caesar-rejtjel

2.1. A Caesar-rejtjel működése és feltörése

A Caesar-rejtjel a titkosírások egyik legközismertebb fajtája, a monoalfabetikus behelyettesítéses módszerek közé tartozik. Minden betűt egy előre meghatározott értékkel eltolunk az ABC-ben, ezért ez egy bijektív leképezés. Összesen annyi titkosítási kulcs lehetséges ahány betűből áll az ABC, tehát könnyen feltörető, ez a módszer viszont alapjául szolgál több bonyolultabb titkosírásnak, ilyen például a Vigenère-rejtjel.

A szöveg feltörés jóval egyszerűbb, mint az 1.3 alfejezetben tárgyalt általános esetben, mivel a plaintext és ciphertext ABC-je megegyeznek, vagyis $pABC = cABC$. A kulcs megtalálása itt egyenértékű az eltolási szám megtalálásával. Érdemes a betűgyakoriságot alkalmazva megkeresni mindkét szöveg leggyakoribb karakterét, nagy eséllyel ezzel meg is kaptuk az eltolás mértékét. Fontos, hogy a gyakorisági táblázatok egy bizonyos nyelv köznapi szövegei alapján készültek, tehát például egy szakszöveg esetén könnyen kaphatunk más eredményeket is.

A plaintext elmeit átírhatjuk egy matematikai formába: az ABC betűit számértékekkel fogjuk megfeleltetni: $A = 0$, $B = 1$, $C = 2$, és így tovább. Jelölje c (ciphertext) a kódolt-, és p (plaintext) a kódolandó betű értékét, továbbá legyen n az ABC elemszáma, e pedig az eltolás mértéke. Ekkor: $c \equiv (p+e)\ (mod\ n)$. A kongruencia tulajdonságait felhasználva a visszafejtésnél ezt kapjuk: $p \equiv (c-e)\ (mod\ n)$.

A következő alfejezetekben két általam készített programot fogok bemutatni. Az első a kódoló program, amellyel titkosítani lehet egy fájlban leírt üzenetet a felhasználó által megadott eltolás mértékkel. A dekódoló programmal pedig visszanyerhetjük az eredeti szöveget. Az alapnyelv az ékezetek nélküli magyar nyelv a könnyebb megvalósíthatóság miatt. Az ellenőrzés egy szótár alapján történik, így a dekódolás főként hosszabb szövegekre alkalmazható.

2.2. A Caesar-rejtjel kódoló programja

A 2.1 ábrán a `main` felépítése és a többi függvény deklarációja látható.

```
void beolvas(string& input, int& input_hossza);
void mennyi_eltolas(int& eltolas_merteke);
void eltolas(string& input, ofstream& output, int input_hossza, int eltolas_merteke);

int main()
{
    string input;
    int input_hossza;
    int eltolas_merteke;
    ofstream output ("kodoltuzenet.txt");

    beolvas(input, input_hossza);
    mennyi_eltolas(eltolas_merteke);
    eltolas(input, output, input_hossza, eltolas_merteke);

    return 0;
}
```

2.1. ábra. A `caesar_kodolas` program függvényei

A titkosításra szánt szöveget a program egy `txt` fájlból olvassa be, amit a `beolvas` függvény valósít meg. Ezt az információt az `input` nevű globális változó tárolja. Ha a felhasználó helytelen fájlnevet adna meg, a konzolon megjelenik egy erre vonatkozó hibaüzenet és újra lehet próbálni a helyes név beírását. A `mennyi_eltolas` függvénnyel lehet megadni, hogy mennyivel szeretnénk eltolni a szöveget, ezt az `eltolas_merteke` változóban rögzítjük. A titkosítást az `eltolas` hajtja végre, ez minden betűt annyival tol el, amennyi a korábban megadott `eltolas_merteke`.

```
for(int szam = 0; szam < input_hossza; szam++)
{
    if(isalpha(input[szam]))
    {
        input[szam] = tolower(input[szam]);
        for(int i = 0; i < eltolas_merteke; i++)
        {
            if(input[szam] == 'z')
                input[szam] = 'a';
            else
                input[szam]++;
        }
    }
}
```

2.2. ábra. Részlet az `eltolas` függvény

A program átadja a megvalósításhoz szükséges adatokat, illetve a `main` elején létrehozott kimeneti fájlt, aminek segítségével a program bezárása után is elérhetővé válik a titkosított üzenet. A későbbiekben az így létrehozott fájlokkal lesz tesztelhető a dekódoló program. Az átláthatóság érdekében az eredeti és titkosított szöveg, illetve az eltolás mértéke a konzolon is megjelenik, így a fájlok megnyitása nélkül is nyomon követhető a folyamat.

2.3. A Caesar-rejtjel dekódoló programja

A program elején a `beolvas_szotar` függvény egy `szotar` nevű változóba rögzíti a magyar nyelv 50 leggyakoribb szavát tartalmazó listát. Ennek segítségével tudom ellenőrizni, hogy értelmes szöveget kaptam-e, épp ezért az *a* és *s* szavakat az egyszerűség kedvéért kihagytam ebből a listából. Ezek után a `beolvas` függvénnyel beolvasom a kódolt üzenetet, amit az `input`-ba tárolok. Fontos megemlíteni, hogy itt létrehozok egy `eredeti_input` nevű változót is, mert később az `eltolas` átírja az `input` tartalmát. Ha az adott eltolással a szöveg nem bizonyul értelmesnek, akkor másik eltolási számmal kell próbálkozni, amihez szükség van arra, hogy az `input` tartalma visszaálljon az eredeti bemenetre.

A beolvasások után a program `leggyakoribb` függvénye megadja a kódolt szöveg leggyakoribb betűjét, ami a konzolon is megjelenik. Ezt fogom megfeleltetni először *e*-nak, *a*-nak és így tovább az általános sorrend szerint.

Egy ciklusba szervezem a kódolt szöveg feltörésére irányuló próbálkozásokat, ami végigmegy az összes lehetséges eseten. A `megfelelo` függvény aszerint, hogy éppen hányadik ciklusban vagyok visszaadja a `megfelelo_betu`-t aminek a `leggyakoribb_betu`-től való távolsága lesz az adott ciklus `eltolasi_mertek`-e. Ezt az értéket a `mennyi_eltolas` függvény számolja ki, egyszerűen addig tolja a `leggyakoribb_betu`-t amíg az meg nem egyezik a `megfelelo_betu`-vel. A program ekkor létrehozza a `dekodoltuzenet.txt` nevű fájl, amiben az eredményt fogja rögzíteni.

Az `eltolas` függvény ezúttal visszafelé elvégzi a szükséges eltolást. Az `ellenoriz` pedig leellenőrzi a szótár segítségével, hogy értelmes szöveget kaptam-e. Ebben az esetben mivel csak egy eltolási szám van nagyon kicsi annak az esélye, hogy rossz mértékkel értelmes szót kapok, de a biztonság kedvéért bevezettem egy számlálót. Így az `ertelmes` változó csak háromnál több értelmes szó esetén tér vissza igaz értékkel. Ebben az esetben kilépek a ciklusból, hiszen megtaláltam a titkosítási kulcsot. Ha nem, akkor az `input`-nak értékül adom az eredeti kódolt szöveget és a következő leggyakoribb betűvel, illetve egy másik eltolási értékkel újraindul a ciklus.

A dekódolás minden fontos részlete látható a konzolon, ciklusonként kiírom, hogy melyik betű megfeleltetésével és eszerint milyen értékű eltolással próbálkozom. Értelemsze-

rűen az utolsó próbálkozás volt a helyes, hiszen ekkor kilépek a ciklusból, így az is látható hányadszorra kaptam meg az értelmes szöveget. A program végén kiíratom a megfejtett üzenetet, amit el is mentek az erre létrehozott fájlba.

A kódoló és dekódoló program más, ezzel a 26 betűvel leírható nyelvre is könnyen átvihető. Kódolásnál nem szükséges semmilyen módosítás, dekódolásnál pedig csak az ellenőrzésre fenntartott szótár tartalmát, illetve a leggyakoribb betűk általános sorrendjét kell átírni ahhoz, hogy hatékony program legyen, vagyis minél előbb megtalálja a titkosítási kulcsot.

```
void beolvas_szotar(vector<string>& szotar);
void beolvas(string& input, int& input_hossza);
char leggyakoribb(string input, int input_hossza);
char megfelelo(int index);
int mennyi_eltolas(char leggyakoribb_betu, char megfelelo_betu);
void eltolas(string& input, ofstream& output, int input_hossza, int eltolas_merteke);
bool ellenoriz(string input, vector<string> szotar);

int main()
{
    vector<string> szotar;
    beolvas_szotar(szotar);
    string input;
    string eredeti_input;
    int input_hossza;
    beolvas(input, input_hossza);
    eredeti_input = input;

    char leggyakoribb_betu = leggyakoribb(input, input_hossza);
    cout << "A leggyakoribb betu: " << leggyakoribb_betu << endl << endl;

    for(int i = 0; i < 26; i++)
    {
        cout << i+1 << ". proba: ";
        int index = i;
        char megfelelo_betu = megfelelo(index);
        cout << "a megfelelo betu: " << megfelelo_betu << " es ";
        int eltolas_merteke = mennyi_eltolas(leggyakoribb_betu, megfelelo_betu);

        ofstream output ("dekodoltuzenet.txt");
        eltolas(input, output, input_hossza, eltolas_merteke);

        bool ertelmes = ellenoriz(input, szotar);
        if(ertelmes == true) break;
        else input = eredeti_input;
    }

    cout << endl << "A dekodolt szoveg: " << endl << input << endl;
    return 0;
}
```

2.3. ábra. A `caesar_dekodolas` program függvényei

3. fejezet

A Vigenère-kód

3.1. A Vigenère-kód működése és feltörése

A Vigenère-kód a Caesar-rejtjel továbbfejlesztése, a módszert Blaise Vigenère fejlesztette ki a 16. században. A titkosítási eljárás ugyanúgy az eltolás maradt, de egy adott betűt a szövegbeli elhelyezkedése alapján különböző eltolási számokat is rendelek. Ez a polialfabetikus módszerek egyik legismertebb fajtája, könnyen látható, hogy a 1.3 alfejezetben ismertetett betűgyakoriságon alapuló feltörési módszer ilyen formában nem alkalmazható.

A szöveg kódolásához vegyünk egy kulcsszót, ezt írjuk egymás után annyiszor a plaintext alá amennyiszer csak lehet. Ekkor minden betűt annyival kell eltolni, mint amennyi az alá írt betű sorszáma az ABC-ben. A dekódolásnál pedig a kulcsszó megfejtése után ugyanezt kell visszafelé alkalmazni. Tekintsük a következő táblázat részletet, amelynek egésze tartalmazza az összes lehetséges eltolást, minden sorban egy 1 és 26 közötti szám szerint. Látható, hogy ez tulajdonképpen a Caesar-rejtjel esetit is magában foglalja.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X

Ezzel a módszerrel tulajdonképpen létrehoztunk egy műveletet, a betűk közötti összeadást. Az ábrán az eredményt a megfelelő sor és oszlop metszete adja meg. Az összeadás ellentett művelete, a kivonás is hasonlóan értelmezhető az ábra alapján. A táblázat szimmetriája miatt teljesül a kommutativitás, sőt az asszociativitás is. A csoport egységeleme az a betűvel való eltolás, szerencsés tehát ha a kulcsszó ezt a betűt nem tartalmazza.

A nemzetközi ABC elemeiből alkotott csoport izomorf a $\mathbb{Z}_{26}$ Abel-csoporttal, hiszen a csoporton értelmezett művelet kommutatív, ami az táblázat alapján is könnyen látható.

Ez a feltalálásakor feltörhetetlennek gondolt kódolási technika ma már korántsem megfejthetetlen. Az egyik legelső kísérlet az úgynevezett betörési pont keresése. Kicsit hasonlít az előző módszerhez, a lényeg ugyanis, hogy ha meg tudnánk sejteni egy szót, amit a kódolt szöveg tartalmaz, akkor azt összevetve az eredeti szöveggel megkapnánk a kulcsszót vagy annak részletét. Második esetben az adott részletet egyesével bővítve végül megkapjuk az kulcsszó hosszát. A helyes hossz felismerhető, mert értelmes szövegdarabok fognak megjelenni, innen a kulcs többi rész már kiegészíthető. Betörési pontokra alkalmas lehet például az aláírás vagy a megszólítás, illetve tagolt szöveg esetén a rövid szavak, névelők.

Ha a kulcsszó hossza ismert, akkor a feltöréshez érdemes a gyakoriságanalízis eszközeihez nyúlni. A Caesar-kódoláshoz hasonlóan itt is megegyezik a plaintext és a ciphertext ABC-je. Legyen k a kulcsszó hossza, ekkor a betűk modulo k szerint maradékosztályokba sorolhatók a szövegben elfoglalt helyük alapján. Így olyan halmazokat kapunk, amelynek elemeit ugyanakkora értékkel tolták el, vagyis a kulcsszó egy adott betűjét adták hozzá minden elemhez. Ezekre az osztályokra már külön-külön alkalmazható a gyakoriságanalízis.

Egy halmazon belül válasszuk ki a leggyakoribb betűket, hiszen valamelyik a $pABC$ egyik leggyakoribb betűjének a képe. Az a cél, hogy kitaláljuk mi szerepelhetett a kulcsszó azon helyén, amelynek maradékosztályát vizsgáljuk. Olyan betűt kell választani, amellyel a visszafelé való eltolás után a betűk gyakorisági sorrendben elfoglalt helye nem sokban különbözik. Ha például kiválasztottunk az a betűt, melyre a halmaz leggyakoribb betűjét dekódolva e-t kapunk, de második leggyakoribbnak viszont x-et, akkor tudjuk hogy nagy valószínűséggel nem a megfelelő betűt találtuk meg. Ezt az algoritmust minden halmazra elvégezve megkapjuk a kulcsszó egészét.

Látható tehát, hogy ha semmilyen információ nem áll rendelkezésünkre sem a szövegről sem a kulcsszóról, akkor ezt a kódolást feltörni már nagyon nehéz feladat. Minél hosszabb a szöveg és minél rövidebb a kulcsszó annál több elem kerül az egyes maradékosztályokba és annál pontosabb lesz a megállapított gyakorisági sorrend. Ha a szöveg rövid és a kulcs hosszú, akkor az üzenet közel feltörhetetlen.

3.2. A Vigenère-titkosítás kódoló programja

Ez a program egy `txt` fájlban megadott ékezet nélküli szöveget kódol az 3.1 fejezetben tárgyalt módszerrel. A tesztelés során a különböző bemeneteket a következő oldal segítségével tettem ékezet nélkülivé:

http://www.unit-conversion.info/texttools/remove-letter-accents/.

A titkosítás során a szöveg központozása, formázása és ékezetnélkülisége megmarad, a nagybetűk azonban kicsikké alakulnak. Ezek a kis változtatások megkönnyítik a kódolás folyamatát, miközben a szöveg értelmezhetőségét nem befolyásolják.

```
void beolvas(string& input);
void kulcsszo_megadas(string& kulcsszo);
void eltolas(string& input, ofstream& output, string kulcsszo);

int main()
{
    string input;
    string kulcsszo;
    ofstream output ("kodoltuzenet.txt");

    beolvas(input);
    kulcsszo_megadas(kulcsszo);
    eltolas(input, output, kulcsszo);

    return 0;
}
```

3.1. ábra. A `vigenere_kodolas` program függvényei

A titkosítás három globális változó és ugyanennyi függvény segítségével történik. Elsőként az `input`-ba beolvasom a titkosításra szánt fájl tartalmát. Ez a függvény tartalmaz egy hibakezelést. Ha a felhasználó hibás nevet adna meg, akkor a program újra kéri a helyes név begépelését. Az átláthatóság érdekében a megadott szöveget a konzolra is kiírom.

A `kulcsszo_megadas` függvény futása során a felhasználó meg tudja adni azt a kulcsszót, amivel a szöveget titkosítani szeretné. Miután ez megtörtént a felhasználónak nincs több feladata, a konzolon láthatóvá válik a már kódolt üzenet. Ezt a folyamatot, vagyis a program lényegi részét az `eltolas` függvény tartalmazza.

Létrehozok egy kulcs nevű egész értékekből álló vektort, ez fogja tartalmazni az eltolásokat jelző számértékeket. Egy ciklusban végig megyek a kulcsszó betűin és minden elemnél a neki megfelelő eltolási értéket hozzáadom a vektorhoz. Ezt egy 26 elágazásból álló `switch` segítségével valósítom meg, melynek minden ága egy lehetséges kulcsszóbeli betű esetét fedi le.

A következő 3.2 kódrészletben a kulcsszót annyiszor másolom egymás mögé, hogy hossza megegyezzen az input betűinek számával, így a tényleges eltolásnál már nem kell maradékosztályokkal foglalkozni. A vektorban a kulcsszónak megfelelő számsor egyszer már szerepel, így elég a számlálót a kulcsszó hosszától indítani. Egy ciklus során a `segedszam` jelzi, hogy a szöveg adott karaktere hányadik maradékosztályban van és eszerint a kulcsszó hányadik mezőjében lévő eltolási számot kell hozzáadni a `kulcs` vektorhoz.

```
int segedszam;
for(int j = kulcsszo.size(); j < input.size(); j++)
{
        segedszam = j % kulcsszo.size();
        kulcs.push_back(kulcs[segedszam]);
}
```

3.2. ábra. Részlet az `eltolas` függvényből

Ezek után már végrehajtható az `input` eltolása. Bevezetek egy `index` változót, ami biztosítja, hogy csakis akkor lépjek a `kulcs` következő elemére, ha az `input`-ban olvasott karakter betű, tehát lényegében ez egy külön betűszámláló. Az `input` minden elemére megvizsgálom hogy betű-e, ha igen, akkor annyiszor tolom el jobbra az ABC, amekkora számérték szerepel a neki megfelelő indexű helyen a kulcsban. Ez látható a 3.3 belső `for` ciklusban. Biztosítani kell továbbá, hogy ha az ABC végére értem, akkor a következő eltolásnál a folyamatot az a betűnél folytatódjon. A létrejött szöveget a `main` függvény elején deklarált `output` nevű `txt` fájlban rögzítem, így az a program bezárása után is elérhető lesz.

```
int index = 0;
for(int szam = 0; szam < input.size(); szam++)
{
    if(isalpha(input[szam]))
    {
        input[szam] = tolower(input[szam]);
        for(int k = 0; k < kulcs[index]; k++)
        {
            if(input[szam] == 'z') input[szam] = 'a';
            else input[szam]++;
        }
        index++;
    }
}
output << input;
```

3.3. ábra. Részlet az `eltolas` függvényből

3.3. A Vigenère-titkosítás dekódoló programja

Ez a program egy `txt` formátumú, az adott módszerrel titkosított szöveget dekódol. Egy megfelelő bemenetet a Vigenère-titkosítás kódoló programjával 3.2 elő lehet állítani. Betörési pontként a felhasználónak meg kell adnia az általa használt kulcsszó hosszát. Fontos megjegyezni, hogy az értelmesség ellenőrzéséhez a korábban összeállított 50 leggyakoribb magyar szóból álló listát használom, így a program magyar nyelvű szövegek vissza-

fejtésére alkalmas. A program más nyelvekkel is használható, ha ezt a szótárat lecseréljük például az angol nyelv leggyakoribb szavaira és módosítjuk az általános betűgyakorisági sorrendet.

```
void beolvas_szotar(vector<string>& szotar);
void beolvas(string& input);
void kulcsszohossz_megadas(int& kulcsszohossz);

vector<char> feloszt(string input, int kulcsszohossz, int hanyadik_halmaz);
vector<char> gyakorisag_sorrend(vector<char> adott_halmaz);
vector<int> mennyi_eltolas (vector<char> adott_halmaz, vector<char> gyakorisag);

vector<char> eltolas(vector<char> adott_halmaz, int mertek);
string visszarendez(string input, int kulcsszohossz, vector < vector<char> > eltolt_halmazok);
int ellenoriz(string eltolt_szoveg, vector<string> szotar);

string elso_kapu(vector < vector<char> > halmazok, vector < vector<int> > osszesmertek,
                 string input, int kulcsszohossz, vector<string> szotar);
void novel(vector<int>& szamlalo, int kulcsszohossz);
vector<int> masodik_kapu(string elso_szoveg, vector<string> szotar, int kulcsszohossz);
string hamradik_kapu(string elso_szoveg, int kulcsszohossz, vector<int> rossz_indexek,
                     vector<string> szotar);
void kiir(string vegso_szoveg);

int main()
{
    string input;
    ofstream output ("dekodoltuzenet.txt");
    int kulcsszohossz;
    vector<string> szotar;
    beolvas_szotar(szotar);
    beolvas(input);
    kulcsszohossz_megadas(kulcsszohossz);
    vector < vector<char> > halmazok;
    vector < vector<char> > sorbarendezett_halmazok;
    vector < vector<int> > osszesmertek;

    for(int i = 0; i < kulcsszohossz; i++)
    {
        int hanyadik_halmaz = i;
        vector<char> adott_halmaz = feloszt(input, kulcsszohossz, hanyadik_halmaz);
        halmazok.push_back(adott_halmaz);
        vector<char> gyakorisag = gyakorisag_sorrend(adott_halmaz);
        sorbarendezett_halmazok.push_back(gyakorisag);
        vector<int> mertekek = mennyi_eltolas(adott_halmaz, gyakorisag);
        osszesmertek.push_back(mertekek);
        cout << endl;
    }

    string elso_szoveg = elso_kapu(halmazok, osszesmertek, input, kulcsszohossz, szotar);
    vector<int> rossz_indexek = masodik_kapu(elso_szoveg, szotar, kulcsszohossz);
    string vegso_szoveg = hamradik_kapu(elso_szoveg, kulcsszohossz, rossz_indexek, szotar);
    kiir(vegso_szoveg);
    return 0;
}
```

3.4. ábra. A `vigenere_dekodolas` program függvényei

A program függvényei és a `main` alapvető felépítése a 3.4 képen látható. Elsőként létrehozom az `input` és `output` változókat, illetve a kulcsszó hosszát tároló `kulcsszohossz`-t. A `szotar` nevű vektorba feltöltöm a leggyakoribb szavakat tartalmazó listát. Beolvasom a dekódolni kívánt szöveget és bekérem a `kulcsszhossz`-t. Ezután deklarálom az `halmazok`, `sorbarendezett_ halmazok` és `osszesmertek` változókat. A követő ciklusban ezeket töltöm fel az `input`-ból kinyert megadott adatokkal. Egy ciklus egy adott maradékosztállyal foglalkozik.

A megadott szöveg betűit a kulcsszó hosszával megegyező számú halmazra osztom, aszerint, hogy melyik maradékosztályok kerülnek. Az adott halmazon belül gyakorisági sorrendbe rendezem az elemeket. Ennek segítségével meg tudom állapítani minden halmazra, hogy melyek a legvalószínűbb eltolási értékek. Ezután egy három lépcsős feltörés következik, amit a fejezet hátralévő részében részletesebben be fogok mutatni.

Vegyük sorra elsőként a `main for` ciklusában szereplő függvényeket. A `feloszt` függvény lényege, hogy az `input` betűit helyük szerint a megfelelő maradékosztályokba rendezze. A program során egy ilyen osztályt halmaz hívok, melyeket egy mátrixban fogok tárolni. Egy sor egy maradékosztálynak felel meg, így `kulcsszohossz` darab sor lesz, az oszlopok száma pedig az `input` betűinek száma osztva a kulcsszó hosszával és ennek a felső egész része. A memóriaszemét kiszűrése érdekében minden mezőt kezdetben feltöltök csillagokkal. Ezután végig megyek az `input` elemein, megint egy külön indexet használva a betűk megszámlálására.

```
int index = 0;
for(int szam = 0; szam < input.size(); szam++)
{
    if(isalpha(input[szam]))
    {
        input[szam] = tolower(input[szam]);
        int segedszam;
        for(int i = 0; i < kulcsszohossz; i++)
        {
            if(index % kulcsszohossz == i)
            {
                segedszam = (index - i)/kulcsszohossz;
                halmazok[i][segedszam] = input[szam];
                break;
            }
        }
        index++;
    }
}
```

3.5. ábra. Részlet a `feloszt` függvényből

Ha a program betűt olvas, akkor kicsivé alakítja, hiszen előfordulhat, hogy a felhasználó olyan üzenetet szeretne feltörni, melynek titkosítása során a megmaradtak a nagybetűk.

A `segedszam` adja meg, hogy az adott betű hányadik elemként fog a halmazba kerülni. A belső `for` ciklusban megkeresem, hogy az elem melyik halmazba kerüljön, ezt az `i` számláló jelöli. Ha megtaláltam a kellő mezőt a mátrixban, akkor kilépek a belső ciklusból és áttérek az `input` következő elemének vizsgálatára. Ha befejeződött ez a folyamat, akkor egy `adott_halmaz` nevű vektorba feltöltöm a kiválasztott sort, amit a `main`-ből átadott `hanyadik_halmaz` érték határoz meg. Ezt vektort a főprogrambeli `adott_halmaz` változónak adom értékül.

A `gyakorisagi_sorrend` egy halmaz betűit állítja csökkenő sorrendbe az előfordulásuk száma szerint. A `betukszama` tömb minden értékét nullára állítom, mert egy betű előfordulásakor a neki megfeleltetett indexű mezőt fogom növelni. Ezután egy ciklusban kiválasztom a tömb legnagyobb elemét és meghatározom, hogy ez az érték melyik betűhöz tartozik, majd ezt a betűt hozzáadom a `gyak` vektorhoz. A `betukszama`-ból kitörlöm a legnagyobb elemet és megismétlem a folyamatot mindaddig, amíg az összes a halmazban előforduló betű bele nem kerül a vektorba, melyet a függvény futása végén értékül adok a `gyakorisag_vektor` nevű globális változónak.

```
int index_kulonbseg;
bool rossz = false;
for(int j = 0; j < 5; j++)
{
    char betu = gyakorisag[j];
    int index1 = j;

    char eltolt_betu;
    int index2 = 0;
    for(int k = 0; k < mertek; k++)
    {
        if(betu == 'a')  betu = 'z';
        else betu--;
    }
    for(int k = 0; k < 26; k++)
    {
        if(eltolt_betu == betugyakorisagok[k]) index2 = k;
    }

    index_kulonbseg = abs(index1 - index2);
    if(index_kulonbseg > 10)
    {
        rossz = true;
        break;
    }
}
if(rossz) rossz_mertekek.push_back(mertek);
```

3.6. ábra. Részlet a `mennyi_eltolas` függvényből

A `mennyi_eltolas` függvény felállítja valószínűségi sorrendben a lehetséges eltolási értékeket. Ezen sorrend meghatározása két részletben történik. Elsőként a `betugyakorisagok`

tömbben felsorolom gyakoriságuk szerint a magyar nyelv betűit (ékezetek nélkül). Egy ciklusban a létrehozom a `leggyakoribb_betu`-t, ami a halmazban szereplő leggyakoribb betűt rögzíti, ezt fogom megfeleltetni a `betugyakorisagok` első, második, stb. betűinek. Minden megfeleltetésnél kapok egy mértéket, vagyis hogy mennyivel kell visszatolni a `leggyakoribb_betu`-t hogy a `betugyakorisagok` adott elemét kapjam. A gyakoriságanalízis szerint ez lesz a legvalószínűbb eltolás, de a második részletben végrehajtok még egy ellenőrzést. Megnézem, hogy a visszatolás előtti és utáni gyakorisági sorrendet nem különbözik-e nagyon egymástól, mert például ha a halmaz második leggyakoribb betűje eltolás után x lesz, akkor nagy bizonyossággal mondhatjuk, hogy ez mégsem egy jó eltolási szám.

Ez az ellenőrzés a 3.6 ábrán látható. A ciklusban a halmaz első 5 leggyakoribb betűn fogok végigmenni. Az `index1` jelöli a betű halmazán belül felállított gyakorisági sorrendben szereplő helyét, az `index2` pedig az eltolás utáni helyét mutatja a `betugyakorisagok` tömbben. Ha valahol az `index_kulonbseg` nagyobb, mint 10, akkor a mérték bekerül a `rossz_mertekek` vektorba. Ezután végignézem a `mertekek` elemeit, ha egy mező értéke szerepel a rossz mértékek között, akkor a azt áthelyezem a `mertekek` végére, tehát a legvalószerűtlenebb eltolási számok közé.

Most azt a három függvényt fogom bemutatni, amik a `main`-ben külön-külön nem szerepelnek, de a dekódolási kapuk futása során többször is meghívásra kerülnek. Az `eltolas` egy halmaz elemeit tudja eltolni egy számértékkel, például a korábban meghatározott `osszesmertek` egyik vektorának elemével.

```
istringstream iss(segedszoveg);
string sor;
int hanyadik_sor = 0;
while(getline(iss, sor))
{
    for(int k = 0; k < kulcsszohossz; k++)
    {
        if(hanyadik_sor % kulcsszohossz == k)
        {
            for(int l = 0; l < sor.length(); l++)
            {
                segedszoveg2[hanyadik_sor + kulcsszohossz*l] = sor[l];
            }
        }
    }
    hanyadik_sor++;
}
```

3.7. ábra. Részlet a `visszarendez` függvényből

A `visszarendez` függvény az eltolt halmazokat fűzi össze az eredeti input formájára. A `segedszoveg`-be soronként feltöltöm a maradékosztályokat, tehát lényegében egy `string`-

be írom át a mátrix elemeit. Ennek tartalmát fogom átmásolni a megfelelő sorrendben a `segedszoveg2` karakterekből álló tömbbe. A `hanyadik_sor` minden ciklusban egy adott halmazt jelöl. Beolvasás közben a külső `for` ciklus a halmazokon megy végig és az `if` feltételéből tudja a program, hogy éppen melyik mellékosztálynál tartunk. Ez biztosítja hogy a halmaz elemei a kellő helyre kerüljenek. A belső ciklus pedig a `sor` elemeit tölti fel a `segedszoveg2`-be, a `hanyadik_sor` számával megegyező indexnél kezdve és onnan mindig kulcsszó hossznyit léptetve előre.

Magyarul a folyamat a nulla maradékosztálynak megfelelő halmazzal kezdődik, ennek elemeit beírom sorban a `segedszoveg2` $0, 0 + \texttt{kulcsszohossz}, 0 + 2 * \texttt{kulcsszohossz}, \ldots$ indexű mezőire, amíg a halmaz végére nem értem. Ezután következik a második halmaz (1-es maradékosztály), ennek elemeit a következő helyekre írom: $1, 1+\texttt{kulcsszohossz}, 1+2*\texttt{kulcsszohossz}, \ldots$ és így tovább, amíg át nem írtam az összes halmaz összes elemét. Ezután már csak végig kell menni az input betűin és át kell írni azokat a `segedszoveg2`-ben azonos helyen lévő betűire. Így a halmazokra szabdalt, eltolt `input` felveszi az eredetileg megadott üzenet alakját, vagyis megkaptuk a szöveg egy lehetséges visszafejtését, aminek ellenőrizhetjük az értelmességét.

Az `ellenoriz` függvény a szöveg értelmességének leellenőrzéséért felel. Pontosabban visszaadja mennyi értelmes szót talált a szövegben. Az `eltolt_szoveg`-et szavanként átmásolom egy vektorba, majd dupla ciklussal minden szóra ellenőrizzük, hogy szerepel-e a szótárban, ha igen, akkor eggyel növelem az egyezések számát.

```
int egyezesek = 0;
for(int i = 0; i < szavak.size(); i++)
{
   for(int j = 0; j < szotar.size(); j++)
    {
        if(szavak[i] == szotar[j])
        {
            egyezesek++;
        }
    }
}
```

3.8. ábra. Részlet az `ellenoriz` függvényből

Most áttérek a feltörési folyamat ismertetésére, melynek első lépése az `elso_kapu` függvény. Lényege, hogy minden halmazra veszem a három legvalószínűbb eltolási értéket és végig próbálom az összes lehetőséget, ezekből választom ki azt, amely során a legtöbb értelmes szót találtam. Ehhez először létrehozok egy kulcsszó hosszúságú `szamlalo` vektort, ebben tárolom minden halmazra, hogy hányadik lehetséges eltolással próbálkozom. Ezt úgy kezelem, minta hármas számrendszerben számolnék, pl ha a `szamlalo` első számjegy

0, akkor az azt jelenti, hogy az adott próbálkozás során az első halmaznál a legvalószínűbb eltolási számmal dolgozom. Ha a csupa 0 számból indulok és minden ciklusban egyet adok a számhoz, amíg az csupa 2 nem lesz, akkor az összes kombinációt leellenőriztem.

```
for(int i = kulcsszohossz; 0 <= i; i--)
{
    if(szamok[i] = 2)
    {
        szamok[i] = 0;
        if(szamok[i - 1] != 2)
        {
            szamok[i - 1]++;
            break;
        }
    }
    else
    {
        szamok[i]++;
        break;
    }
}
```

3.9. ábra. Részlet a `novel` függvényből

Ezt a folyamatot egy külön `novel` nevű függvényben valósítom meg, amely egy részlete a 3.9 ábrán látható. A `szamalo` tartalmat átmásolom a `szamok` tömbbe, elvégzem az összeadást, majd a változásoknak megfelelően módosítom a `szamlalo`-t.

```
for(int j = 0; j < hatar; j++)
{
    vector< vector<char> > eltolt_halmazok;

    for(int k = 0; k < kulcsszohossz; k++)
    {
        vector<int> mertekek = osszesmertek[k];
        vector<char> eltolt_halmaz = eltolas(halmazok[k], mertekek[szamlalo[k]]);
        eltolt_halmazok.push_back(eltolt_halmaz);
    }

    string eltolt_szoveg = visszarendez(input, kulcsszohossz, eltolt_halmazok);
    int egyezes = ellenoriz(eltolt_szoveg, szotar);
    egyezesek[j] = egyezes;
    if(j != hatar - 1) novel(szamlalo, kulcsszohossz);
}
```

3.10. ábra. Részlet az `elso_kapu` függvényből

Az algoritmus összesen $3^{\texttt{kulcsszo\_hossza}}$ esetet különít el, ezt a számot az egyszerűség kedvéért `hatar`-nak nevezem el. Az `elso_kapu` függvényben olyan ciklust kell konstruálni, mely pont ennyiszer fog lefutni. Minden ciklusban a `szamlalo` által meghatározott mértékkel eltolom a halmazokat, majd összerendezem azokat és ellenőrzöm,

hogy mennyi értelmes szó szerepel a szövegben. Ezen számokkal töltöm fel a ciklus előtt létrehozott szintén `hatar` nagyságú `egyezesek` tömböt.

Ebből maximum kiválasztással meg tudjuk mondani minden halmazra melyik eltolás a legkedvezőbb. Ha egy csupa nulla vektort annyiszor növelek amennyi a maximum indexének száma, akkor éppen azt a `szamlalot` kapom, amely során létrejött az adott kombinációk közül a legértelmesebb. Alkalmazom az összes halmazra a neki megfelelő legjobb eltolást, összerendezem a kapott halmazokat és ezt a `string`-et adom vissza a `main`-nek `elso_szoveg` néven. Természetesen itt még nem biztos, hogy megkaptuk az eredeti, teljesen értelmes szöveget, ezt az eredményt a további kapuk során ellenőrzöm és ha szükséges javítom.

```
int betukszama = 0;
for(int i = 0; i < szavak.size(); i++)
{
   string adottszo = szavak[i];
   for(int j = 0; j < szotar.size(); j++)
    {
        if(adottszo == szotar[j])
        {
            for(int k = 0; k < adottszo.length(); k++)
            {
                int helyesindex = (betukszama + k) % kulcsszohossz;
                ertelmes[helyesindex]++;
            }
        }
    }
    for(int k = 0; k < adottszo.length(); k++)
    {
        if(isalpha(adottszo[k])) betukszama++;
    }
}
```

3.11. ábra. Részlet a `masodik_kapu` függvényből

A `masodik_kapu` feladata az, hogy megmondja mely halmazokat toltuk el jó mértékkel és melyeket nem. Sorban végig megyek az `elso_szoveg` szavain, mindegyikre ellenőrzöm, hogy benne van-e a szótárban. Ha igen, akkor megnézem, hogy az értelmesnek bizonyult `adott_szo` milyen indexeket fednek le a szövegben és azok melyik maradékosztályokba esnek a kulcsszó hosszával elosztva. Azokat a halmazoknak értékét növelem egyel a korábban létrehozott `ertelmesek` tömbben, melyek elemei szerepeltek az `adott_szo`-ban. Például ha az üzenet az *egy* szóval kezdődik, akkor `kulcsszohossz` hosszúságú tömb első három elemét 0-ról 1-esre írom.

Az `adott_szo` betűinek számát hozzáadom a `betukszama`-hoz, amivel a `helyesindex` meghatározásánál kell számolni, hiszen tudni kell éppen hányadik betűnél tartok a szövegben. Itt külön ellenőrizni kell, hogy a szó minden eleme betű-e, mert az `elso_szoveg`

szavakra bontása a szóközök alapján történik. Az értelmes szavak ellenőrzésénél erre nincs szükség, hiszen ha egy szó egyezik egy szótárbelivel, akkor már biztosan nem tartalmaz egyéb karaktereket. A szótárban főként névelők és kötőszavak szerepelnek, amik után nem teszünk vesszőt vagy pontot, ezért a szóközönkénti tagolás nem rontja az értelmes szavak találásának esélyét. Ezután végig megyek az `egyezesek` tömbön, amit kezdetben kinulláztam. Ha egy eleme kisebb, mint 3, akkor a ciklus száma jelölő index bekerül a `rossz_halmazok` vektorba, ami visszaadja azon halmazok sorszámát, ahol nem találtunk eddig helyes eltolást.

A `harmadik_kapu` függvényben az `masodik_kapu` által visszaadott rossz mértékkel szereplő maradékosztály eltolási számát javítom ki. Először halmazokra osztom az `elso_szoveg`et, úgy ahogy az a `main` függvényben is látható. Fontos megemlíteni, hogy itt nem az eredeti inputtal dolgozom, azaz a rossz halmazok már el vannak tolva, így a mostani helyes mérték nem az a szám, mint amennyivel az eredeti `input`-ban kellett volna dolgozni és ami a titkosítási kulcsban szerepelt. Egy ciklusban végig veszem a rossz halmazokat, mindegyikre kipróbálva az összes lehetséges eltolást, hogy megállapíthassam melyiknél találtam a legtöbb értelmes szót. Mivel mind a 26-féle eltolást meg kell nézni, ezért nem használom az `osszesmertek` tömböt.

```
int ertelmes_szavak[26];
for(int j = 0; j < 26; j++)
{
    vector<char> eltolt_halmaz = eltolas(halmazok[rossz_indexek[i]], j);
    segedhalmazok[rossz_indexek[i]] = eltolt_halmaz;
    string proba = visszarendez(elso_szoveg, kulcsszohossz, segedhalmazok);
    ertelmes_szavak[j] = ellenoriz(proba, szotar);
}
```

3.12. ábra. Részlet a `harmadik_kapu` függvényből

A `halmazok`-at átmásolom a `segedhalmazok` vektorba, az `elso_szoveg` tartalmát pedig a `segedszoveg` tömbbe. A függvény futása során ezeket a segédváltozókat módosítom, tehát miután ellenőriztem egy lehetőséget tudok újra az eredeti értékekkel dolgozni. Egy `rossz_halmaz` vizsgálata során az összes eltolás elvégzése után kiválasztom a legtöbb egyezést adó eltolást, mely megadja a halmaz végleges eltolását. Nagy valószínűséggel már az `elso_kapu` során megkapom az értelmes szöveget, esetleg egy-két halmaz lehet csak rossz. Mire elérek a `harmadik_kapu`-hoz, annak esélye, hogy egy rossz halmaz mindkét szomszédja rossz és így nem találok egyetlen értelmes szót sem már minimális. Így nagy biztonsággal kijelenthető, hogy kellően nagy szövegre a program megtalálja az eredeti titkosított üzenetet.

Végül a `kiir` függvény kiírja a dekódolt szöveget a konzolra, illetve elmenti azt a `main` elején létrehozott `txt` fájlba.

3.4. A dekódoló program továbbfejlesztése

A 3.3 Vigenère-dekódoló program elengedhetetlen lépése volt, hogy a felhasználó adja meg a titkosításhoz használt kulcs hosszát. A továbbfejlesztetés lényege, hogy a titkosított szöveg a kulcs megadása nélkül is visszafejthető legyen.

```
int main()
{
    string input;
    ofstream output ("dekodoltuzenet.txt");
    vector<string> szotar;
    beolvas_szotar(szotar);
    beolvas(input);

    vector<int> szamok;
    for(int i = 3; i < 9; i++)
    {
        cout << endl << i - 2 << ". probalkozas, a kulcsszo hossza: " << i << endl;
        int ertelmes_szavak = 0;
        string proba = dekodol(input, i, szotar);
        ertelmes_szavak = ellenoriz(proba, szotar);
        szamok.push_back(ertelmes_szavak);
        cout <<  "Az ertelmes szavak szama: " << ertelmes_szavak << endl;
    }

    string feltort_szoveg = eredmeny(input, szamok, szotar);
    output << input;
    return 0;
}
```

3.13. ábra. A `vigenere_tovabb` program

A dekódolás annyiban módosul, hogy a `main` függvényében végignézem a 3-től 8-ig terjedő eseteket a kulcsszó hossza szerint. Természetesen lehetne 8-nál nagyobb számot is írni felső korlátnak, de az ennél hosszabb kulcsszó ritka és jelentősen megnövelné a futási időt. Alapvetően az előző program függvényeit (3.4) használom, ezeken kívül csak a következőket hoztam itt létre:

`string dekodol(string szoveg, int szam, vector<string> szotar)`

`string eredmeny(string szoveg, vector<int> szamok, vector<string> szotar).`

A `dekodol` felel meg az előző program `main` függvényének a beolvasásokat leszámítva, ezek ebben az esetben is a `main` elején szerepelnek. A `dekodol` a `proba` nevű változónak adja értékül a ciklus száma által meghatározott `kulcsszohossz` alapján a visszafejtett szöveget. Erre futtatok egy ellenőrzést, amely megadja a `proba`-ban talált értelmes szavak számát. Ezeket az értékeket minden ciklus végén rögzítem a `szamok` vektorban, illetve kiírom a konzolra az egyes ciklusokban kapott eredményt, vagyis hogy milyen kulcsszó-hosszra hány értelmes szót talál a program.

Az `eredmeny` egy maximum kiválasztással megadja azt az esetet, mely során a legtöbb értelmes szó volt a `proba`-ban, a maximumhely alapján pedig megkapom a helyes kulcsszóhosszt. Ezután újból futtatom a `dekodol` függvényt, amely ezúttal az eredeti szöveget fogja visszaadni, amit a `kiir` segítségével szintén kiírok a konzolra.

4. fejezet

A permutációs titkosítás

4.1. Bevezetés

A szakirodalom nagy része a betűk összekeverésének módszerét hívja permutációs titkosításnak, ebben a fejezetben azonban a következőt fogja jelenteni. A plaintext minden betűjéhez hozzárendelünk egy, a kulcs által meghatározott betűt úgy, hogy a leképezés bijekció legyen. A titkosítás kulcsa, tehát az 1-től 26-ig szereplő számok egy permutációja, ahol egy adott helyen lévő szám az *abc*-ben azonos helyen szereplő betű képét fogja megadni. Például ha a permutáció első elem a 3, akkor az *a* betűk helyett a kódolt szövegben mindenhol *c*-t írok. A kódoló programnak egy külön `txt` fájlban kell megadni ezt a számsort, ami alapján az elvégzi a titkosítást. A dekódoló programnak pedig az a feladata, hogy a permutáció minél több elemét meghatározza.

4.2. A permutációs titkosítás kódoló programja

A kódoló program 3 db függvényt tartalmaz, ezek a `beolvas`, a `permutacio_megadas` és a `titkosit`. A `beolvas` az `input` változóban rögzíti a külön fájlban tárolt titkosítani kívánt üzenetet. A `permutacio_megadasa` során a program bekéri a felhasználótól a kulcsot tartalmazó fájl nevét. Ez a függvény elvégez egy ellenőrzést is, hogy a felhasználó tényleg egy 26 elemű permutációt adott-e meg. Ezt a tulajdonságot tárolja a `main`-ben létrehozott logikai `helyes` változó. Elsőként megnézem, hogy a kulcs 26 elemet tartalmaz-e, majd egy ciklusban ellenőrzöm, hogy 1-től 26-ig minden szám szerepel-e. Ha a fájl egy kellő számot nem tartalmaz, akkor a `helyes` változó értékét hamisra állítom és kiírom a konzolra az erre vonatkozó hibaüzenetet. Ebben az esetben a kulcs átírása után újra kell indítani a programot. Amennyiben a `helyes` igaz értékkel tér vissza, úgy a program végrehajtja a titkosítást. Egy karakterekből álló tömbben rögzítem az *abc*-t, ezután végig megyek az `input` összes elemén. Minden egymás-

tól különböző betűnek létrehozok egy külön esetet, ami alapján a program végrehajtja a leképzést. Így az *a* betű ága `input[i] = abc[permutacio[0] - 1]`, a *b* betű ága `input[i] = abc[permutacio[1] - 1]` és így tovább, ahol `i` az `input` éppen olvasott elemét jelöli. Az algoritmus végén a titkosított szöveget a `kodoltuzenet.txt`-be másolom és kiírom a konzolra.

```
void beolvas(string& input);
void permutacio_megadas(vector<int>& permutacio, bool& helyes);
void titkosit(string& input, vector<int> permutacio, ofstream& output);


int main()
{
    string input;
    vector<int> permutacio;
    ofstream output ("kodoltuzenet.txt");

    bool helyes = true;
    beolvas(input);
    permutacio_megadas(permutacio, helyes);
    if(helyes) titkosit(input, permutacio, output);

    return 0;
}
```

4.1. ábra. A permutációs titkosítás kódoló programja

4.3. A permutációs titkosítás dekódoló programja

A program célja a permutációs kulcs meghatározása és az kódolt szöveg visszafejtése, ezt a folyamatot több függvénycsoport segítségével valósítom meg. Először a `beolvas` függvény az `input` változóban rögzíti a kódolt szöveget, a `deklaral` pedig létrehozza a csupa nullákból álló `permutacio` vektort. Az `elso_kapu` több lépcsőt is tartalmaz, melyeket külön egységekbe foglaltam, ezek mindegyike egy-egy betű permutációs számát határozza meg. Az egyik lépcső közben meghívom a `gyakorisagi_sorrend`-et, amit a korábbi dekódoló programok során már használtam. A `masodik_kapu` a kódolt szöveg betűgyakoriságaira támaszkodva feltölti a `permutacio` üresen maradt mezőit megadva ezzel egy lehetséges titkosítási kulcsot. Ezek alapján kiírom a konzolra a kapott szöveget, az `elso_kapu` során lerögzített betűket fehérrel a többit kékkel. A `harmadik_kapu` egy interaktív felhasználói felületet hoz létre, aminek segítségével át lehet írni a rossz permutációs számmal szereplő kék betűket. A program részletes működését az alábbi bekezdésekben fogom kifejteni.

```
void beolvas(string& input);
vector<char> gyakorisagi_sorrend(string input);
vector<int> deklaral();

void elso_kapu(string input, vector<int>& permutacio);
void elso_lepcso(vector<string> szavak, vector<int>& permutacio);
void masodik_lepcso(vector<string> szavak, vector<int>& permutacio);
void harmadik_lepcso(vector<string> szavak, vector<int>& permutacio, string input);
bool lehet_e(vector<string> szavak, char betu);
void negyedik_lepcso(vector<string> szavak, vector<int>& permutacio);
void otodik_lepcso(vector<string> szavak, vector<int>& permutacio);
void hatodik_lepcso(vector<string> szavak, vector<int>& permutacio);

void masodik_kapu(string& input, vector<int>& permutacio, vector<int>& seged);
int megfelelo_index(char betu);

void harmadik_kapu(string& input, vector<int>& seged);

int main()
{
    string input;
    beolvas(input);
    ofstream output("dekodoltuzenet.txt");
    vector<int> permutacio = deklaral();
    elso_kapu(input, permutacio);
    vector<int> seged;
    masodik_kapu(input, permutacio, seged);
    harmadik_kapu(input, seged);
    output << input;

    return 0;
}
```

4.2. ábra. A permutációs titkosítás dekódoló programja

Az `elso_kapu` függvény az `input` tartalmát szóközönként átmásolja a `szavak` vektorba, majd sorban meghívja a különböző lépcsőket. Az `elso_lepcso` az a betű permutációs számát határozza meg mégpedig úgy, hogy a szavak közül kiválogatom az egy karakter hosszú és betűt tartalmazó szavakat. Hosszabb szöveg esetén előfordulhat néhány s vagy e szó, de legtöbbször biztosan az a névelő fog szerepelni. A kigyűjtött szavakat az `egybetus_szavak` vektorból egy ciklussal az `egybetuk`-be másolom, de itt már mindegyik csak egyszer fog szerepelni. Eközben az `index` vektorban számon tartom melyik betű mennyiszer szerepelt. Innen egy maximum kiválasztással megkapom azt a betűt, ami a titkosítás előtt az a volt. Ezután lerögzítem a permutációs számát, ami az a képének abc-beli sorszáma lesz, azaz `if(leggyakoribb_betu == "a") permutacio[0] = 1`, `if(leggyakoribb_betu == "b") permutacio[0] = 2` és így tovább végigmenve mind a 26 lehetőségen. Egy betű permutációs számát mindig a vektor azonos indexű mezőjébe írom, mint a leképezés előtti betű helye az abc-ben, ebben az esetben ezért a kapott számot az a-nak megfelelő nulladik mezőbe írom.

A `masodik_lepcso`-ben a z betű permutációs számát határozom meg. Kiválogatom az a betűvel kezdődő kétbetűs szavakat, ezek közül a leggyakoribb második betű lesz z-nek a képe. Ez az előző bekezdésben tárgyalt és az ábrán látható módon könnyen meghatározható. Ebben az esetben a leggyakoribb betű abc-beli helyét a z-nek megfelelő, azaz a permutáció 26-odik mezőjébe írom.

```
vector<string> egybetuk;
vector<int> index;
for(int i = 0; i < egybetus_szavak.size(); i++)
{
    if(i == 0)
    {
        egybetuk.push_back(egybetus_szavak[0]);
        index.push_back(1);
    }
    else
    {
        bool nemvolt = true;
        for(int j = 0; j < egybetuk.size(); j++)
        {
            if(egybetus_szavak[i] == egybetuk[j])t
            {
                index[j]++;
                nemvolt = false;
                break;
            }
        }
        if(nemvolt)
        {
            egybetuk.push_back(egybetus_szavak[i]);
            index.push_back(1);
        }
    }
}
```

4.3. ábra. Részlet az `elso_lepcso` függvényből

A `harmadik_lepcso` az e betű permutációs számát adja meg. A magyar nyelvben előforduló 3 leggyakoribb betű sorrendben a következők: e, a ,t, ezek közül az a-t már meghatároztuk, viszont az még előfordulhat, hogy például t lesz a leggyakoribb. Ennél a résznél használom a `gyakorisagi_sorrend`-et, illetve létrehoztam még egy `lehet_e` függvényt, ami visszaad egy logikai változót, hogy a gyakorisági sorrendben meghatározott elem eredetileg lehetett-e e. Ezt úgy lehet megmondani, hogy összeszámolom a szövegben azokat a maximum 3 betűs szavakat, amelyben az adott betű szerepel kizárva az a-nak megfelelő betűt. Egy általános szövegben körülbelül soronként fordul elő egy ez, egy, ezt, se, sem, stb, ezért a számlálót a szavak számának függvényében határozom meg. 3 betűs t-t (azt, ezt, itt) és más betűket tartalmazó szavak lényegesen kevesebbszer fordulnak elő, így biztosan meg lehet mondani, hogy melyik betű ősképe volt az e.

```
char a_betu = abc[permutacio[0] - 1];
char leggyakoribb_betu;
vector<char> sorrend = gyakorisagi_sorrend(input);
for(int i = 0; i < sorrend.size(); i++)
{
    leggyakoribb_betu = sorrend[i];
    bool valoszinu = lehet_e(szavak, leggyakoribb_betu);
    if(valoszinu && leggyakoribb_betu != a_betu) break;
}
```

4.4. ábra. Részlet az `harmadik_lepcso` függvényből

Az ezt követő lépcsők során hasonló módszerrel keresek permutációs számokat. A `negyedik_lepcso` az s számát határozza meg. Összeszámolom az e betűvel kezdődő kétbetűs szavakat, a leggyakoribb második betű lesz s képe. Az `otodik_lepcso` az i számát találja meg, összegyűjtve az s-re végződő kétbetűs szavakat úgy, hogy az e-vel kezdődőket nem számolom bele, a `hatodik_lepcso` pedig az azt/ezt alapján a t-hez tartozó számot mondja meg. Ha egy szövegben nem szerepelnek az is, azt, ezt szavak, akkor az adott permutációs szám egyszerűen nulla marad és a `masodik_kapu` során fog értéket kapni. Minél több az ilyen függvény, annál nagyobb eséllyel kapjuk meg az eredeti titkosítási kulcsot, azonban a ritkább betűknél ez már egyre kisebb eséllyel lesz célravezető.

A `masodik_kapu` feladata az üresen maradt permutációs számok meghatározása a betűk gyakorisági sorrendje alapján. Egy `seged` vektorba másolom a függvénynek átadott `permutacio`-t, így a későbbiekben tudni fogom melyek voltak a fix mezők, vagyis azok az értékek, amiket az `elso_kapu` során kaptam. Először feltöltöm a `gyakorisagok` vektort a szövegben nem szereplő betűkkel, hogy a hossza megegyezzen a `permutacio` hosszával. Ekkor rendelkezésemre áll a kiegészített `gyakorisagok` vektor, ami a halmazbeli előfordulásuk szerint csökkenő sorrendben tartalmazza a betűket, illetve egy a általános a magyar nyelvre vonatkozó gyakorisági sorrend. A feladat az, hogy ezen vektorok elemeit összepárosítsuk úgy, hogy közben a fix mezők változatlanul maradnak.

Az összepárosítás végrehajtása előtt egy `gyakorisagok2` nevű vektorba átmásolom a `betugyakorisagok` azon elemeit, amik helyén a `permutacio` vektorban egy nullától különböző szám áll. Majd a `gyakorisagok` közül kitörlöm azokat, amiknek ABC-beli sorszáma szerepel a `permutacio`-ban. Egy konkrét példán keresztül ez a következőképpen néz ki. Az általános sorrend leggyakoribb betűje az e, aminek permutációs számát a `harmadik_lepcso` során meghatároztam, így ez nem fog szerepelni a `gyakorisagok2`-ben. Aztán megnézem, hogy a `permutacio` 4. mezőjében milyen elem van, tegyük fel, hogy ez a szám az egyes. Ez azt jelenti, hogy a titkosított szöveg a betűi eredetileg e-k voltak, ezért a `gyakorisagok` közül kitörlöm az a betűt.

```
for(int i = 0; i < gyakorisagok.size(); i++)
{
    char honnan = gyakorisagok2[i];
    char hova = gyakorisagok[i];
    int szam1;
    int szam2;
    for(int j = 0; j < 26; j++)
    {
        if(honnan == abc[j]) szam1 = j;
        if(hova == abc[j]) szam2 = j + 1;
    }
    permutacio[szam1] = szam2;
}
```

4.5. ábra. Részlet az `masodik_kapu` függvényből

A folyamat végén kaptam egy megegyező elemszámú `gyakorisagok` és `gyakorisagok2` vektort. Most már csak az indexek szerint össze kell párosítani a betűket. A `honnan` jelöli a `gyakorisagok2` a `hova` pedig a `gyakorisagok` adott elemét. A `szam1` a `honnan`, illetve `szam2` a `hova` ABC-beli sorszáma. A ciklus végén a `permutacio` `szam1` mezőjébe fog kerülni a `szam2` értéke. Nézzük például a következő esetet. A `betugyakorisagok` elemei: e, a, t, o, l, stb., ezek közül az első három permutációs számát az `elso_kapu` függvény kiszámolta, ezért a `gyakorisagok2` első eleme az o lesz, aminek száma 14 (0-tól kezdve az indexelést), vagyis `szam1` $= 14$. A `gyakorisagok` első betűje legyen mondjuk r, ekkor `szam2` $= 18$ (1-től indexelve). Végül a `permutacio` 14. mezőjébe beírom a 18-at, ami pont azt jelenti hogy ami a kódolt szövegben r, ez a feltevésünk szerint eredetileg o volt.

A függvény végén átírom az `input` elemeit a kapott kulcs alapján, majd a szöveget kiírom a konzolra, a korábban lefixált betűket és egyéb karaktereket fehérrel, a `masodik_kapu` során feltöltött mező betűit pedig kékkel, hogy a felhasználó lássa mely betűk biztosan és melyeket lehet módosítani, ha az szükséges.

A `harmadik_kapu` egy interaktív felhasználói felületet valósít meg, amely során át lehet írni a kék betűket. A konzolon megjelenik a következő szöveg: `A felhasznaloi felulet aktivalasahoz irj be 1-et, kulonben 0-t:`, tehát egy nulla beírásával a függvény véget ér és a dekódolt szöveg a kulcs alapján visszafejtett input lesz. Majd a program megkérdezi, hogy melyik betűt milyenre írjon át, hiszen már vannak értelmes szavak és szótöredékek. A változás alapján újra kiírom a szöveget, de úgy, hogy a megadott betű már fehérrel szerepeljen. Könnyen létrejöhet olyan helyzet, hogy egy betűből van fehér és kék változat is, ezért a korábbiakban deklarált `seged` vektort használom. A szöveg minden elemére rögzítem, hogy fix mező volt-e vagy sem és átírásnál az adott betűk fix mezővé válnak. Így minden egyes ciklussal csökken a kék betűk száma és minden ciklus után a felhasználónak lehetősége van befejezni a javítást. Ha a felhasználó kilép a ciklusból, akkor az `input` értékét átmásolom az `output` fájlba, majd a program véget ér.

Irodalomjegyzék

[1] Megyesi Zoltán - Titkosírások (Kisújszállás, 1999, Szalay Kiadó)

[2] Simon Singh - The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography (New York, 1999, Doubleday Publishing Company)