Algoritmusok és adatszerkezetek:

Rúddarabolás

(mohó algoritmusok)

Készítette: Molnár Attila
Neptun-azonosító: IQ93HY
e-mail: molnar.attila@szlgbp.hu

Dokumentácio linkje: https://www.overleaf.com/read/tsrqvcgyxcnt
Kurzuskód: ITL-AA1G
Gyakorlatvezető neve: Menyhárt László Gábor

2021. április 24.

Tartalomjegyzék

1. fejezet

Felhasználói dokumentáció

1.1. Feladat

Rúddarabolás. Adott egy fémrúd, amelyet megadott számú és hosszúságú darabokra kell felvágni.

A darabok hosszát milliméterben kifejezett értékek adják meg. Olyan vágógéppel kell a feladatot megoldani, amely egyszerre csak egy vágást tud végezni. A vágások tetszőleges sorrendben elvégezhetőek.

Egy vágás költsége megegyezik annak a darabnak a hosszával, amit éppen (két darabra) vágunk. A célunk optimalizálni a műveletsor teljes költséget. Készíts programot, amely

- kiszámítja a vágási műveletsor optimális összköltségét;
- megad egy olyan vágási sorrendet, amely optimális költséget eredményez!

1.2. Futási környezet

IBM PC, exe futtatására alkalmas, legalább 32-bites operációs rendszer (pl. Windows 10). Nem igényel egeret.

1.3. Használat

1.3.1. A program indítása

A futtatható program (exe fájl windows-ra) a projektfájlban található a Debug könyvtárban.

1.3.2. A program bemenete

A program az adatokat a *billentyűzetről* olvassa be a következő sorrendben:

#	**Adat**	**Magyarázat**	**Lehetséges értékek**
1.	N	a darabok száma	$1 \leq N \leq 1000$
2.	T_i	a darabok hossza	$1 \leq T_i \leq 1000$

1.3.3. A program kimenete

A program először kiírja, hogy mennyi a vágások összköltsége, majd megadja a vágások sorrendjét is; soronként kiírja, hogy mekkora hosszúságú rudat mekkora darabokra vág.[1]

[1] A Bíró csak az egyik keletkező szakaszt kérte, de az error csatornán megjelenik a másik rész is.

1.3.4. Mintabemenet

```
Adja meg a vagando darabok szamat! (1 <= X <= 1000): 5
  Adja meg a(z) 1. darab hosszat! (1 <= X <= 1000): 2
  Adja meg a(z) 2. darab hosszat! (1 <= X <= 1000): 5
  Adja meg a(z) 3. darab hosszat! (1 <= X <= 1000): 2
  Adja meg a(z) 4. darab hosszat! (1 <= X <= 1000): 7
  Adja meg a(z) 5. darab hosszat! (1 <= X <= 1000): 10
```

1.3.5. Mintakimenet

```
Az osszkoltseg: 55
a(z) 1. lepesben kettevagando rud hossza 26 ,
    az igy keletkezo (kisebbik) rud hossza 10,
    a masik rud hossza pedig 16

a(z) 2. lepesben kettevagando rud hossza 16 ,
    az igy keletkezo (kisebbik) rud hossza 7,
    a masik rud hossza pedig 9

a(z) 3. lepesben kettevagando rud hossza 9 ,
    az igy keletkezo (kisebbik) rud hossza 4,
    a masik rud hossza pedig 5

a(z) 4. lepesben kettevagando rud hossza 4 ,
    az igy keletkezo (kisebbik) rud hossza 2,
    a masik rud hossza pedig 2
```

1.3.6. Hibalehetőségek

A következő hibák fordulhatnak elő:

- nem számot, hanem szoveget adnak meg.
- nem a megadott intervallumba eső számot adnak meg.
- nem az előre megadott mennyiségű számot adják meg.
- nem egész számot, hanem tizedestörtet adnak meg.

Az első két esetben a program hibaüzenetet ad, melyben részletezi a felhasználónak a probléma okát és újra bekéri az adatot. A program tehát nem indul újra, hanem ott folytatja, ahol a hiba történt.

Az utolsó előtti esetben, ha több számot adnak meg, akkor a program lefut a kevesebb adattal, ha pedig kevesebbet, akkor további bemenetre vár.

Ha felhasználó számot ad meg, de az nem egész szám, hanem tizedestört, akkor azt a program csonkolni fogja és az így kapott számmal fog lefutni. Erre a fejlesztési lehetőségeknél még visszatérek. A következő oldalon található mintafutásban is található példa a jelenségre.

Mintafutás hibás bemeneti adatok esetén

```
Adja meg a vagando darabok szamat! (1 <= X <= 1000): -4
 HIBA! Tul kicsi szamot adott meg. Probalkozzon ujra!
5
  Adja meg a(z) 1. darab hosszat! (1 <= X <= 1000): 3
  Adja meg a(z) 2. darab hosszat! (1 <= X <= 1000): blablabla
 HIBA! Nem egesz szamot adott meg. Probalkozzon ujra!
4
  Adja meg a(z) 3. darab hosszat! (1 <= X <= 1000): 5000
 HIBA! Tul nagy szamot adott meg. Probalkozzon ujra!
7
  Adja meg a(z) 4. darab hosszat! (1 <= X <= 1000): 0
 HIBA! Tul kicsi szamot adott meg. Probalkozzon ujra!
1
  Adja meg a(z) 5. darab hosszat! (1 <= X <= 1000): 2.9
Az osszkoltseg: 36
a(z) 1. lepesben kettevagando rud hossza 17 ,
    az igy keletkezo (kisebbik) rud hossza 7,
    a masik rud hossza pedig 10

a(z) 2. lepesben kettevagando rud hossza 10 ,
    az igy keletkezo (kisebbik) rud hossza 4,
    a masik rud hossza pedig 6

a(z) 3. lepesben kettevagando rud hossza 6 ,
    az igy keletkezo (kisebbik) rud hossza 3,
    a masik rud hossza pedig 3

a(z) 4. lepesben kettevagando rud hossza 3 ,
    az igy keletkezo (kisebbik) rud hossza 1,
    a masik rud hossza pedig 2
```

2. fejezet

Fejlesztői dokumentáció

2.1. Feladat

Rúddarabolás. Adott egy fémrúd, amelyet megadott számú és hosszúságú darabokra kell felvágni.

A darabok hosszát milliméterben kifejezett értékek adják meg. Olyan vágógéppel kell a feladatot megoldani, amely egyszerre csak egy vágást tud végezni. A vágások tetszőleges sorrendben elvégezhetőek.

Egy vágás költsége megegyezik annak a darabnak a hosszával, amit éppen (két darabra) vágunk. A célunk optimalizálni a műveletsor teljes költséget. Készíts programot, amely

- kiszámítja a vágási műveletsor optimális összköltségét;
- megad egy olyan vágási sorrendet, amely optimális költséget eredményez!

2.2. Specifikáció

$$[a..b] \stackrel{\text{def}}{=} [a,b] \cap \mathbb{N}$$

Bemenet: $N \in \mathbb{N}$,
$T : [1..N] \to \mathbb{N}$,

Kimenet: $K \in \mathbb{N}$,
Egész : $[1..N-1] \to \mathbb{N}$
Egyik : $[1..N-1] \to \mathbb{N}$
Másik : $[1..N-1] \to \mathbb{N}$

Előfeltétel: $N \in [1..1000] \wedge (\forall i \in [1..N])\,(T_i \in [1..1000])$

Utófeltétel:

$$\text{Darabolás}^{N,T}(\text{Egész}, \text{Egyik}, \text{Másik}, K) \wedge$$
$$\wedge\, (\forall K')\,(\text{Darabolás}^{N,T}(\text{Egész}, \text{Egyik}, \text{Másik}, K') \to K \leq K'),$$

ahol

$$\text{Darabolás}^{N,T}(\text{Egész}, \text{Egyik}, \text{Másik}, K) \stackrel{\text{def}}{\iff} K = \sum_{i=1}^{n} \text{Egész}_i \wedge$$
$$\wedge\, \text{Egész}_{N-1} = \sum_{i=1}^{N} T_i \wedge$$
$$\wedge\, (\forall i \in [1..N-1])\,(\text{Egész}_i = \text{Egyik}_i + \text{Másik}_i \wedge$$
$$(\exists j \in [i..N-2])\,\text{Egész}_i \in \{\text{Egyik}_j, \text{Másik}_j\})$$

Az utófeltétel azt mondja, hogy a kimenő paraméterek egy darabolást adnak meg, mégpedig a legjobbat (minden más K' költséggel járó darabolás olyan, hogy ez a K' többe vagy ugyanannyiba kerül). Itt N és T paraméterű K költséggel járó darabolás alatt olyan (Egész, Egyik, Másik) függvényhármast értünk, amelyre igaz, hogy

- Az Egész függvény értékkészletének összege K,
- bármely értelmezett i bemenetre

$$\text{Egész}_i = \text{Egyik}_i + \text{Másik}_i$$

- Az T bemeneti paraméter értékkészletének összege Egész legnagyobb egésszel történő behelyettesítési értéke,
- Egész bármely más i-vel történő behelyettesítéséhez van egy "korábbi" $j > n$ index, amellyel Egész_i megegyezik Egyik_j vagy Másik_j valamelyikével.

2.3. Fejlesztői környezet

Visual Studio 2019, Microsoft Visual C++ 2019.

2.4. Forráskód

Állomány	Magyarázat
`IQ93HY/RudDarabolas.exe`	futtatható állomány
`IQ93HY/RudDarabolas.cpp`	C++ kód.
`IQ93HY/tesztek/be1.txt`	teszt-bemeneti fájl$_1$
`IQ93HY/tesztek/be1_rossz.txt`	teszt-bemeneti fájl$_2$
`IQ93HY/tesztek/be2_rossz.txt`	teszt-bemeneti fájl$_3$
`IQ93HY/tesztek/be3_rossz.txt`	teszt-bemeneti fájl$_4$
`IQ93HY/dokumentacio.pdf`	dokumentáció

2.5. Megoldás

2.5.1. Heurisztika: DP, mohó, kupac

A feladat részproblémákra bomlik. A rúddarabolás során minden kettévágás után egy, az eredeti problémához hasonló részproblémábal találjunk szemben magunkat; ha a feladat egy T tömb szerinti feldarabolás, és az első vágás a tömböt egy T_1 és $T_2 (T_1 \cup T_2 = T)$ részre bontja, akkor az eredeti probléma költsége a két vágás költségétől és egy konstantól (a rúd hossza) függ.

A feladat legoptimálisabb megoldásának szükséges feltétele a részproblémák optimális megoldása. A legolcsóbb költségű vágássorozatot keressük, ehhez pedig szükséges (bár magában nem elégséges) az, hogy bárhogy is vágjuk szét a két részt, azok feldarabolása a legoptimálisabb módon történjen. (Ha ugyanis ez nem a legoptimálisabb lenne, akkor az optimális részprobléma feldarabolását felhasználva létezne egy még olcsóbb szétdarabolás is).

Optimalitás elégséges feltétele. A feladat nehézsége abban rejlik, hogy egy a fentihez hasonló, ám **elégséges** feltételt találjunk a darabolásra. Ha ezt a vágást az összes lehetséges megoldás áttekintése nélkül is meg lehet állapítani, akkor nagy mértékben lehet gyorsítani a keresési időn; exponenciális idő helyett polinomiális vagy akár lineáris idő dinamikus vagy mohó algoritmusokkal. Az ötlet e két megközelítést ötvözi.

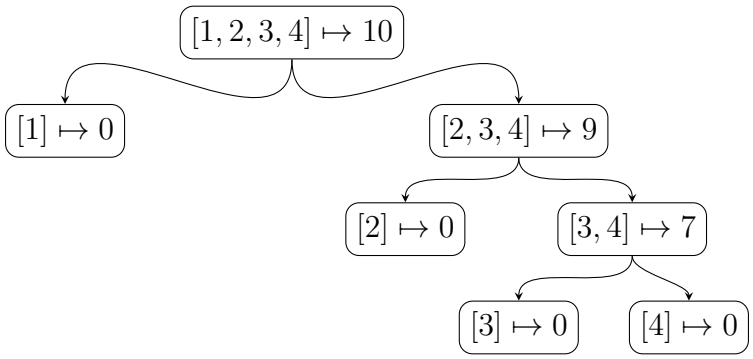
DP: Alulról építkezés. Dinamikus programozással gyorsíthatók azon optimumkeresési feladatok, ahol a részproblémák esetén a részproblémák egy könnyen áttekinthető tulajdonsága alapján lehetséges dönteni jó és még jobb megoldás között, illetve ilyenkor elegendő csak egyetlen (a legjobb) megoldást eltárolni. Ehhez általában arra van szükség, hogy a problémák részproblémákra bontása helyett inkább a részmegoldásokból való megoldásépítés felől érdemes megközelíteni a problémát. A mi feladatunkban ez úgy nézne ki, hogy a következő kérdést tesszük fel:

Egy megadott rúdhalmazból hogyan lehet a legolcsóbban egy hosszú rudat összeragasztani, ha a költség mindig a keletkező rúd hossza?"

Problémafa. Minden vágás két újabb részproblémára vezet, tehát a problémák egy bináris fát alkotnak.

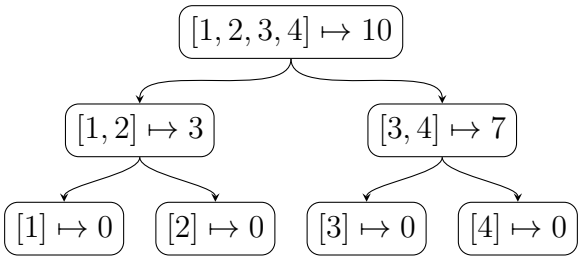
Például a $T = [1, 2, 3, 4]$ probléma két lehetséges feldarabolása (A nyíl az adott lépés költségét mutatja):

1. példa Az első példában a legkisebb számtól kezdve daraboljuk fel a tömböt:



Így a darabolás költsége $10 + 9 + 7 = 26$.

2. példa. Most mindig „középen” vágva daraboljuk fel a rudat:



Így a darabolás költsége $3 + 7 + 10 = 20$.

Lépésenkénti költségfüggvény. A f lépésköltség-függvény tehát lényegében a keletkező tömbök összege, feltéve, hogy a tömb nem 1 hosszúságú, mikor is a költség 0:

$$f(t) = \begin{cases} 0, & \text{ha hossz}(t) = 0, \\ \sum_{i=1}^{\text{hossz}(t)} t_i & \text{egyébként} \end{cases}$$

A fa költsége pedig, ahogy azt a példában is láttuk, az összes részprobléma f-értékenek összege.

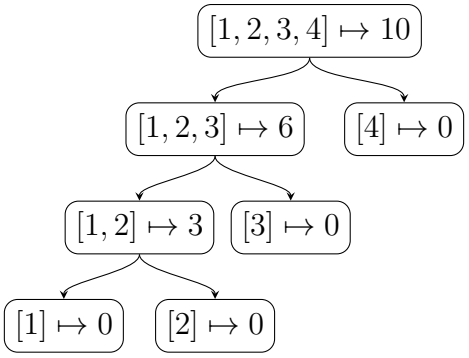
Az egyik példa nem optimális. Az első darabolás különösen nem hatékony, hiszen a 4-es érték sokáig részét képezi a továbbdarabolandó részproblémáknak, így ez a nagy érték minden részproblémában nagyot növel az összeadandó összegen.

Min múlott? Az első példában a 4-es a 7 db részproblémából 4 db-ban jelen van. A második példában a 4-es a 7 db részproblémából csak 3 db-ban van jelen. Cserébe az 1-es van több részproblémában jelen, de az kevésbé járul hozzá az összegekhez, így a végköltség is olcsóbb lesz.

Mire kellene törekedni? Tehát azt látjuk, hogy **a nagy számok részproblémákban való jelenlétét kell minimalizálni – azáltal, hogy a kis számok részproblémákban való jelenlétét kell maximálizálni**, hogy a nagy számoknak már „ne maradjon hely”.

A másik példa sem optimális. Lehetett volna jobban ragasztani: A 3-ast lehetett volna kevesebbszer szerepeltetni. Alulról való felépítés szerint nézve az első összeragasztás az [1,2]. Ezt követően három rúdunk maradt, amelyek 3, 3 és 4 hosszúak. Itt a példa a [3,4] rudat ragasztja össze 7 költségen ahelyett, hogy a két 3 hosszú rudat ragassza össze 6 költségen. Ha így tett volna és fejezte volna be, akkor olcsóbb fát kapott volna:

Optimális példa:



Így a darabolás költsége $3 + 6 + 10 = 19$.

Azt, hogy ennél jobb megoldás nincs, még nem bizonyítottuk.

Mohó stratégia. Alulról nézve tehát az tűnik jó stratégiának, ha az N darabból kiválasztjuk a két legkisebb darabot, és összeragasztjuk őket. Így már csak $N - 1$ darabot kell összeragasztunk. Ebből megint megkeressük a két legkisebbet, és azt ragasztjuk össze, és így tovább.

Ha van optimális megoldás, akkor az előáll mohó választások sorozataként. Az indirekt bizonyítás kedvéért tegyük fel, hogy ez az állítás nem igaz, azaz van egy olyan fa, amely nem a legkisebb elemek összeragasztásaként készült el, mégis optimális, azaz nincsen nála olcsóbb fa. Ha van olyan lépés, amikor az $a_1, a_2, ..., a_k$ részfákból nem a két legkisebbet ragasztja össze, akkor megcserélve ezt a legkisebbekkel egy olyan fát kapunk, amelyek ragasztási költsége kisebb kell legyen. Így az egész fa költsége is csökken, ami ellentmond az indirekt feltevésnek. Tehát az indirekt feltevés lehetetlen, az eredeti állítás pedig bizonyítottan igaz.

Prioritási sor. Az algoritmusban tehát alulról építkezve fogunk mohó stratégiát alkalmazni: minden lépésben kiválasztjuk a két legkisebb elemet, (megjegyezzük a választást, mivel azt is ki kell írni) összeadjuk őket, és az így keletkező számot visszarakjuk a rudak közé. Hogy gyorsítsuk az algoritmust, nem elég a halmazt egyszer lerendezni, mivel a rudak visszatételekor újra módosítani kell majd a rendezésen, amikor az új rudat a *helyükre* beszúrjuk. Hogy ne kelljen ezzel bajlódni, egy prioritási sort veszünk fel abba töltjük majd fel az elemeket és abba rakjuk vissza az összeragasztott elemeket is.

2.5.2. Programparaméterek

Konstansok:

```
max_N: Egész
min_N: Egész
max_T_i: Egész
min_T_i: Egész
```

Típusok:

```
Input = Rekord(
    T: Tömb[min_N..max_N],
    N: Egész)
Output = Rekord(
    Egész: Tömb[1..max_N-1],
    Egyik: Tömb[1..max_N-1],
    Másik: Tömb[1..max_N-1],
    N: Egészm
    K: Egész)
```

Változók

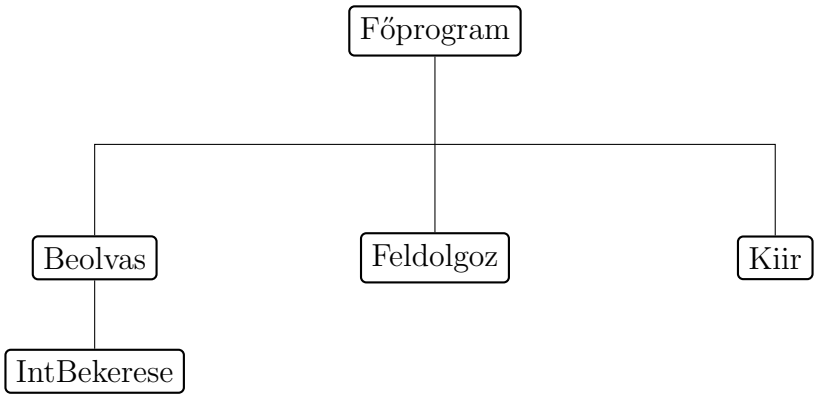
```
    (nincsenek globális változók)
```

2.5.3. Programfelépítés

A program által használt modulok:

`iq93hy_beadando.cpp`	program a forráskönyvtárban
`iostream`	képernyő és billentyűkezelés, a C++ rendszer része
`string`	karakterlánckezelés a beolvasások hibakezelése végett
`queue`	prioritási sor ebben a csomagban megtalálható
`stdlib.h`	általános rutinok, a C++ rendszer része (VisualStudio meghívás nélkül is használja)

2.5.4. Függvénystruktúra



2.5.5. A teljes program algoritmusa

```
Program RúdDarabolás:
    [programparaméterek fentebb megtalálhatók]

    Kiir(Feldolgoz(Beolvas()))
Program vége.

[alprogramok deklarációja:]

Függvény Beolvas(): Input
Függvény Feldolgoz(Input): Output
Eljárás Kiir(Output)

Függvény IntBekerese(Szöveg, Egész,Egész):Egész


[alprogramok:]

Függvény Beolvas(): Input
    Lokális változók
        be: Input

    be.N := Int_bekerese("Adja meg a vagando darabok szamat!", min_N, max_N
    Ciklus i := 1-töl be.N-ig 1-esével
        be.T[i] = IntBekerese("  Adja meg a(z) " + to_string(i + 1) + ". da
    Ciklus vége.
    Beolvas = be
Függvény vége.

Függvény Feldolgoz(be: Input):Output
    Lokális változók
        ki:Output(be.N) [output felveszi be.N alapján a kezdöértékeit]
        kupac: PrioritásiSor(Egész, >) [minimumkiválasztáshoz]

    Ciklus i := 1-töl be.N-ig 1-esével
        kupac.Felvesz(be.T[i])
    Ciklus vége.

    Ciklus i := N-töl 1-ig -1-esével
        ki.Egyik[i] = kupac.Kivesz();
        ki.Másik[i] = kupac.Kivesz();
        ki.Egész[i] = ki.Egyik[i] + ki.Másik[i];
        ki.költség += ki.Egész[i];
        kupac.Felvesz(ki.Egész[i]);
    Ciklus vége.

    Feldolgoz = ki
Függvény vége.
```

```
Eljárás Kiir(Output ki)
    Ki: "Az osszkoltseg: "+ ki.koltseg
    Ciklus i := 1-töl ki.N-1-ig 1-esével
        Ki: "a(z) i+1. lepesben kettevagando rud hossza ki.Egész[i]"
        Ki: "az igy keletkezo (kisebbik) rud hossza  ki.Egyik[i]"
        Ki: "a masik rud hossza pedig ki.Egyik[i]"
    Ciklus vége.
Eljárás vége.


Függvény IntBekerese(kérés:Szöveg, a:Egész, f:Egész):Egész
    Lokális változók
        nyers:Szöveg
        result:Egész
        szame:Logikai
        jo:Logikai

    Ki: kérés + " (a <= X <= f): "
    szame := Hamis
    jo := Hamis
    Ciklus (hátultesztelés)
        Be: nyers

        Hibakezelési környezet
            result := SzövegbölEgészbe(nyers)
        Hiba esetén:
            Ki: "HIBA! Nem egesz szamot adott meg. Probalkozzon ujra!"
            szame := Hamis
        Ha nincs hiba:
            szame := Igaz

        Elágazás szame
            Eset result < a
                Ki: " HIBA! Tul kicsi szamot adott meg. Probalkozzon ujra!"
            Eset result > f
                Ki: " HIBA! Tul nagy szamot adott meg. Probalkozzon ujra!"
            Egyébként
                jo = igaz
        Elágazás vége
    amíg nem jo
    Vissza result
Függvény vége.
```

2.5.6. A kód

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

struct Input
{
    static const int max_N = 1000;
    static const int min_N = 1;
    static const int max_T_i = 1000;
    static const int min_T_i = 1;
    int T[max_N];
    int N;
};

struct Output
{
    static const int max_N = 999;
    int* egesz;
    int* egyik;
    int* masik;
    int koltseg;
    int N;

    Output(const int db)
    {
        N = db - 1;
        egyik = new int[N];
        masik = new int[N];
        egesz = new int[N];
        koltseg = 0;
    }
};

Input Beolvas();
Output Feldolgoz(const Input&);
void Kiir(const Output&);
int Int_bekerese(string, int, int);


int main()
{
    Kiir(Feldolgoz(Beolvas()));
}

Input Beolvas()
{
    Input be;
```

```
    be.N = Int_bekerese("Adja␣meg␣a␣vagando␣darabok␣szamat!",
    Input::min_N, Input::max_N);

    for (int i = 0; i < be.N; i++)
    {
        be.T[i] = Int_bekerese("␣␣Adja␣meg␣a(z)␣"
            + to_string(i + 1)
            + ".␣darab␣hosszat!",
            Input::min_T_i,
            Input::max_T_i);
    }
    return be;
}

Output Feldolgoz(const Input& be)
{
    Output ki(be.N);
    priority_queue<int, vector<int>, greater<int>> kupac;

    // ezt majd a konstruktorba

    for (int i = 0; i < be.N; i++)
    {
        kupac.push(be.T[i]);
    }

    for (int i = ki.N - 1; i >= 0; i--)
    {
        ki.egyik[i] = kupac.top();
        kupac.pop();
        ki.masik[i] = kupac.top();
        kupac.pop();
        ki.egesz[i] = ki.egyik[i] + ki.masik[i];
        ki.koltseg += ki.egesz[i];
        kupac.push(ki.egesz[i]);
    }

    return ki;
}

void Kiir(const Output& ki)
{
    cerr << "Az␣osszkoltseg:␣";
    cout << ki.koltseg << endl;
    for (int i = 0; i < ki.N; i++)
    {
        cerr << "a(z)␣"
             << i+1
             <<".␣lepesben␣kettevagando␣rud␣hossza␣";
```

```
            cout << ki.egesz[i] << " ";
            cerr << ", \n     az igy keletkezo (kisebbik) rud hossza
            cout << ki.egyik[i];
            cerr << ", \n     a masik rud hossza pedig "
                 << ki.masik[i]
                 << endl;
            cout << endl;
        }
    }

    int Int_bekerese(string keres, int a, int f)
    {
        string nyers;
        int result;
        cout << keres;
        cout << " (" << a << " <= X <= " << f << "): ";

        bool szame = false;
        bool jo = false;

        do
        {
            cin >> nyers;

            try
            {
                result = stoi(nyers);
                szame = true;
            }
            catch (...)
            {
                cout << " HIBA! Nem egesz szamot adott meg. Probalk
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n')
                szame = false;
            }

            if (szame)
            {
                if (result < a)
                    cout << " HIBA! Tul kicsi szamot adott meg. Pro
                else if (result > f)
                    cout << " HIBA! Tul nagy szamot adott meg.  Pro
                else jo = true;
            }
        }   while (!jo);
        return result;
    }
```

3. fejezet

Tesztelés

3.1. Érvényes tesztesetek

3.1.1. teszteset: be1.txt

Bemenet – 5-*elemű tömb*
$N = 5$ $T = [2, 5, 2, 7, 10]$
Kimenet
$K = 55$ Egész = $[26, 16, 9, 4]$ Egyik = $[10, 7, 4, 2]$ Másik = $[16, 9, 5, 2]$

A K, Egész és Egyik értéke megegyezik a Bírón található ki1.txt-ben található értékekkel.

3.2. Érvénytelen tesztesetek

Ugyanaz a függvény gondoskodik az N és a T tömbben található értékek hibakezeléséről hibakezeléséről, így elég N esetében tesztelni a hibakezelést.

3.2.1. teszteset: be1_rossz.txt

Bemenet – *nem szám*
$N = tizenhat$
Kimenet
Újrakérdezés: $N =$

3.2.2. teszteset: be2_rossz.txt

Bemenet – *túl nagy szám*
$N = 5000$
Kimenet
Újrakérdezés: $N =$

3.2.3. teszteset: be3_rossz.txt

Bemenet – *túl kicsi szám*
$N = -3$
Kimenet
Újrakérdezés: $N =$

3.2.4. teszteset: be4_rossz.txt

Bemenet – *pozitív tizedestört*
$N = 2$ $T = [2, 2.7]$
Kimenet
$K = 4$ Egész = [4] Egyik = [2] Másik = [2]

A program az esetleges pozitív tizedestörteket alsó egész részre kerekítve értelmezi, de a feladat általánosságban értelmezhető lenne tizedestörtekkel is. A hibakezelés ezt mindenesetre jelen pillanatban nem szűri ki, hanem az átértelmezett feladatot oldja meg.

4. fejezet

Fejlesztési lehetőségek

- Adatok – a felhasználó igénye szerint – akár fájlból is fogadása.
- Hibás fájl-bemenetek felismerése, és a hiba helyének (sor sorszámának) kiírása.
- Többszöri futtatás megszervezése
- Pozitív tizedestörtek hibakezelése: ilyenkor kérjen újra be számot, ne értelmezze át a program magától az egészre kerekített részt, vagy ha átértelmezi, ennek tényéről értesítse a felhasználót.