P-III-1 Vodovodní síť

Jako první si předvedeme řešení, jehož časová složitost je $\mathcal{O}(N^2)$. Pro každé $k = 1, \ldots, N-1$ budeme hledat dvojice hodnot (c_k, d_k) takových, že mezi městy s čísly 1 až $k+1$ lze postavit vodovody v ceně d_k a zároveň platí:

- pokud $c_k > 0$, pak mezi městy s čísly k a $k+1$ je postaven vodovod a síť vodovodů postavená mezi městy s čísly 1 až k umožňuje transportovat c_k jednotek vody z města s číslem k do města s číslem $k+1$,
- pokud $c_k = 0$, pak síť vodovodů postavená mezi městy s čísly 1 až k je dostatečná k zásobování těchto měst vodou, a
- pokud $c_k < 0$, pak mezi městy s čísly k a $k+1$ je postaven vodovod a k dostatečnému zásobování měst s čísly 1 až k je potřeba transportovat $-c_k$ jednotek vody z města s číslem $k+1$ do města s číslem k.

Dvojici hodnot (c_k, d_k), která má výše uvedené vlastnosti, nazveme k-přípustnou.

Povšimněme si, že nemusíme hledat všechny přípustné dvojice hodnot. Pokud $(c_{k,1}, d_{k,1})$ a $(c_{k,2}, d_{k,2})$ jsou k-přípustné dvojice a platí $c_{k,1} \geq c_{k,2}$ a $d_{k,1} \leq d_{k,2}$, pak v libovolném řešení, kde vodovodní síť mezi městy 1 až $k+1$ odpovídá dvojici $(c_{k,2}, d_{k,2})$, lze tuto síť nahradit vodovodní sítí odpovídající dvojici $(c_{k,1}, d_{k,1})$, aniž by se zvýšila cena vodovodní sítě. Náš program pro každé $k = 1, \ldots, N-1$ nalezne seznam k-přípustných dvojic $(c_{k,1}, d_{k,1}), \ldots, (c_{k,\ell}, d_{k,\ell})$ takový, že pro každou k-přípustnou dvojici (c_k, d_k) existuje $i \in \{1, \ldots, \ell\}$ takové, že $c_k \leq c_{k,i}$ a $d_k \geq d_{k,i}$. Pokud je toto splněno, můžeme předpokládat, že optimální řešení pro propojení měst s čísly 1 až $k+1$ používá síť vodovodů odpovídající některé dvojici ze seznamu.

Dále můžeme předpokládat, že $c_{k,1} < \ldots < c_{k,\ell}$ a $d_{k,1} < \ldots < d_{k,\ell}$ (jinak lze některou z dvojic vynechat, aniž bychom porušili vlastnost uvedenou v minulém odstavci). Tato skutečnost nás vede k tomu, že k-přípustné dvojice budeme mít setříděné dle první (a tedy i druhé) souřadnice.

Nyní popíšeme samotný algoritmus. K určení seznamu 1-přípustných dvojic, rozlišíme dva případy. Pokud $a_1 < 0$, pak první město musí být napojeno na druhé město a jediná 1-přípustná dvojice je (a_1, b_1). Pokud $a_1 \geq 0$, pak první město můžeme, ale nemusíme, s druhým městem propojit. Tedy máme dvě 1-přípustné dvojice: $(0, 0)$ a (a_1, b_1).

Předpokládejme, že jsme již nalezli seznam k-přípustných dvojic s výše uvedenými vlastnostmi a tento seznam je $(c_{k,1}, d_{k,1}), \ldots, (c_{k,\ell}, d_{k,\ell})$. Všechny dvojice $(c_{k,i}+a_{k+1}, d_{k,i}+b_{k+1})$, $i = 1, \ldots, \ell$, jsou $(k+1)$-přípustné, neboť odpovídají rozšíření sítě vodovodů mezi městy s čísly 1 až $k+1$ o vodovod mezi městy s čísly $k+1$ a $k+2$. Navíc, pokud i_0 je nejmenší index takový, že $-c_{k,i_0} \leq a_{k+1}$, pak i dvojice $(0, d_{k,i_0})$

je $(k+1)$-přípustná. Pokud takový index i_0 neexistuje, pak hledaný seznam $(k+1)$-přípustných dvojic je $(c_{k,1}+a_{k+1}, d_{k,1}+b_{k+1}), \ldots, (c_{k,\ell}+a_{k+1}, d_{k,\ell}+b_{k+1})$. Pokud takový index i_0 existuje, pak hledaný seznam $(k+1)$-přípustných dvojic je tento seznam vzniklý rozšířením o dvojici $(0, d_{k,i_0})$ a vynecháním dvojic $(c_{k,i} + a_{k+1}, d_{k,i} + b_{k+1})$ s $c_{k,i} + a_{k+1} \leq 0$ a $d_{k,i} + b_{k+1} > d_{k,i_0}$).

Nyní předpokládejme, že jsme nalezli seznam $(N-1)$-přípustných dvojic. Pokud i_0 je nejmenší index takový, že $-c_{N-1,i_0} < a_N$, pak nejmenší cena vodovodní sítě, která splňuje podmínky zadání, je d_{N-1,i_0} a program tedy vypíše číslo d_{N-1,i_0} na výstup. Vzhledem k omezením na vstup ze zadání víme, že řešení vždy existuje.

Podívejme se blíže na časovou analýzu našeho algoritmu. V každém kroku do seznamu přípustných kroků přidáme nejvýše jednu dvojici a některé dvojice případně vynecháme. Tedy seznam k-přípustných dvojic vždy obsahuje nejvýše $k+1$ prvků. Při zpracování seznamu musíme ke každé dvojici ze seznamu přičíst hodnoty a_{k+1} a b_{k+1} a případně přidat do seznamu jednu dvojici a některé dvojice odebrat. Přímočará implementace tohoto postupu vede k řešení, které vyžaduje čas lineární v délce seznamu v daném kroku, a tedy celková časová složitost našeho řešení je $\mathcal{O}(N^2)$. Paměťová složitost je $\mathcal{O}(N)$.

Pokusme se časovou složitost navrhovaného řešení zlepšit. Vzhledem k tomu, že pracujeme se setříděnými posloupnostmi, ve kterých potřebujeme vyhledávat, přirozeně se nabízí použití binárních vyhledávacích stromů. Problémem je implementace operace simultánního přičtení hodnot ke všem prvkům ve stromě. Za tímto účelem si zavedeme zvláštní proměnnou, která bude určovat, o kolik se hodnoty ve stromě liší od těch, které by tam měly být, tj., hodnoty, které reprezentujeme, jsou hodnoty ve stromě zvýšené o hodnotu uloženou v této zvláštní proměnné. Operaci simultánního přičtení pak snadno implementujeme v konstantním čase změnou hodnoty této speciální proměnné. Hodnoty do stromu pak budeme vkládat snížené o hodnotu uloženou v této zvláštní proměnné.

Jak jsme již uvedli, v kroku algoritmu, kdy vytváříme seznam $(k+1)$-přípustných dvojic ze seznamu k-přípustných dvojic, je délka zpracovávaného seznamu nejvýše $k+2 \leq N+1$ a v každém z $N-1$ takových kroků do seznamu přidáme nejvýše jeden prvek. V jednom kroku algoritmu potřebujeme: (1) zvýšit hodnotu všech prvků ve stromě, (2) vyhledat vhodný index i_0, (3) pokud takový index existuje, vložit jeden nový prvek do stromu a (4) případně některé prvky odstranit ze stromu. Operace (1), (2) a (3) v jednom kroku algoritmu vyžadují čas $\mathcal{O}(\log N)$. Operací (4) je za celý běh algoritmu provedeno nejvýše N (ale v jednom kroku jich může být provedeno více) a každá z nich vyžaduje čas $\mathcal{O}(\log N)$. Celková časová složitost takto implementovaného algoritmu je pak $\mathcal{O}(N \log N)$; paměťová složitost je $\mathcal{O}(N)$.

```
#include <stdio.h>
#include <stdlib.h>
#include <set>
#include <algorithm>

#define MAXN 1000000

using namespace std;
```

```
int a[MAXN];
int b[MAXN];

struct SET_QUAD
{
    int c[MAXN][2];             // voda, cena
    int n;

    SET_QUAD(): n(0) {}

    void insert(int a, int b) {
        c[n][0] = a;
        c[n][1] = b;
        n++;
    }

    void increase(int d) {
        for (int i = 0; i < n; i++)
            c[i][0] += d;
    }

    int best() {
        int max = -1;
        for (int i = 0; i < n; i++) {
            if (c[i][0] >= 0 && c[i][1] > max)
                max = c[i][1];
        }
        return max;
    }

    void write() {
        for (int i = 0; i < n; i++)
            printf("(%d, %d) ", c[i][0], c[i][1]);
        printf("\n");
    }
};

struct SET_LINLOG
{
    typedef set<pair<int, int> > S;     // voda, cena
    typedef S::const_iterator IT;

    S s;
    int diff;

    SET_LINLOG(): diff(0) {}

    void insert(int a, int b) {
        IT it = s.insert(make_pair(a - diff, b)).first, it2;

        while (1) {
            // Smažeme zbytečné páry, množinu udržujeme uspořádanou podle cen.
            if (it != s.begin() && (it2 = it, it->second >= (--it2)->second)) {
                // *it je lokálně lepší než *(it - 1)
                s.erase(it2);
                continue;
            }

            if (it2 = it, (++it2 != s.end()) && (it2->second >= it->second)) {
                // *(it + 1) je lokálně lepší než *it
                s.erase(it);
```

```
                it = it2;
                continue;
            }

            break;
        }
    }

    void increase(int d) { diff += d; }

    int best() {
        IT it = s.lower_bound(make_pair(-diff, 0));
        if (it == s.end())
            return -1;
        return it->second;
    }

    void write() {
        for(IT it = s.begin(); it != s.end(); it++)
            printf("(%d, %d) ", it->first + diff, it->second);
        printf("\n");
    }
};

SET_LINLOG Set;

int main() {
    int n, best, price_sum = 0;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        scanf("%d", a + i);

        if (i != n - 1) {
            scanf("%d", b + i);
            price_sum += b[i];
        }
    }

    Set.insert(0, 0);

    for (int i = 0; i < n; i++) {
        Set.increase(a[i]);

        if(i == n - 1) {
            // Jsme na konci
            printf("%d\n", price_sum - Set.best());
        }
        else if ((best = Set.best()) >= 0) {
            // Můžeme provést řez (levá část bude OK)
            Set.insert(0, best + b[i]);
        }
    }

    return 0;
}
```

P-III-2 Vdavky

Nejjednodušší řešení používá myšlenku slévání: budeme postupně slévat všechny fronty (připomeňme, že jsou setříděné sestupně) do jedné, dokud neodebereme K prvků. V každém kroku vždy zvolíme největší prvek z prvních prvků front. Pokud budeme maximum přes všechny fronty hledat například pomocí haldy, bude celková složitost takového přístupu $\mathcal{O}(K \log N)$. S tím se ale samozřejmě nespokojíme.

Nechť K-tý největší prvek je x. Zkusme dělat větší kroky: všechny fronty si rozdělme na skupiny nějaké velikosti $S \geq 1$ (tuto velikost určíme později) a výše popsaným způsobem odeberme $\lfloor K/S \rfloor$ skupin, přičemž skupiny porovnáváme dle velikosti jejich prvního (tedy největšího) prvku. Prvky, které ve frontách zbudou, označíme jako *nepoužité*. Nyní uvažme první prvek p v poslední odebrané skupině. Zjevně všechny nepoužité prvky jsou menší než p (jinak bychom při odebírání vybrali jinou skupinu). Snadno tedy vidíme, že p je větší nebo roven x, protože počet nepoužitých prvků je alespoň $M - K$, kde M je celkový počet prvků.

Co můžeme říct o odebraných skupinách? Nechť jsme z nějaké fronty postupně odebrali skupiny $s_1, \ldots, s_t$. První prvek skupiny s_t je větší nebo roven p, proto všechny prvky ve skupinách $s_1, \ldots, s_{t-1}$ jsou větší než prvek x. Skupina s_t by ale mohla obsahovat prvky menší než x. Do každé fronty tedy vraťme poslední skupinu, kterou jsme z ní odebrali, a ostatní odebrané skupiny prohlasme za *zahozené*. Je-li počet zahozených skupin z, ve výsledných frontách tedy chceme nalézt $(K - zS)$-tý největší prvek. Nyní můžeme celý postup opakovat s $K := K - zS$ a s nějakou menší hodnotou kroku S.

V programu tedy nejprve zvolíme S jako největší mocninu dvojky menší než K, provedeme výše popsaný postup, zmenšíme S na polovinu, a toto opakujeme, dokud $S \geq 1$. V poslední iteraci, kdy $S = 1$, odebíráme jednotlivé prvky, postupujeme tedy stejně jako v prvním algoritmu. Program tedy vždy skončí a vrátí x.

Zbývá určit jeho složitost. V každé iteraci zahodíme alespoň $\max(\lfloor K/S \rfloor \cdot S - N \cdot S, 0)$ prvků, pro aktuální hodnoty K a S, proto prvků větších než x zbude do příští iterace nejvýše $(N + 1) \cdot S$. V následující iteraci tak odebereme maximálně $(N + 1) \cdot S/(S/2) = 2(N + 1)$ skupin (tento odhad platí i pro první iteraci, kde díky volbě počáteční hodnoty S odebereme nejvýše dvě skupiny). Protože odebraní jedné skupiny trvá $\mathcal{O}(\log N)$ a celkový počet iterací je $\mathcal{O}(\log K)$, získáváme tak celkovou složitost $\mathcal{O}(N \log K \log N)$.

Na závěr zmiňme, že existuje řešení pracující v čase $\mathcal{O}(N \log K)$, o kterém se navíc dá dokázat, že je optimální, svou složitostí však přesahuje rámec olympiády.

```
#include <vector>
#include <queue>

struct princ {
        int s; // skupina princů
        int i; // pozice ve skupině
};

bool operator<(const princ &prvni, const princ &druhy)
{
```

```
        return bohatsi (druhy.s+1, druhy.i+1, prvni.s+1, prvni.i+1) < 0;
}

int main(int argc, char* argv[])
{
        // načteme vstup
        int N, K;
        scanf("%d%d", &N, &K);

        std::vector<int> pocty(N);
        for (int i=0; i<N; i++) {
                scanf("%d", &pocty[i]);
        }

        // najdeme nejbližší nižší mocninu dvojky
        int S = 1;
        while (2*S <= K)
                S *= 2;

        // halda nám pomůže hledat frontu s nejbohatším princem
        std::priority_queue<princ> halda;

        std::vector<int> pozice(N, 0);
        while (S >= 1) {
                // vymažeme haldu
                halda = std::priority_queue<princ>();

                for (int i=0; i<pozice.size(); i++) {
                        // vrátíme poslední skupinu, pokud to lze
                        if (pozice[i] > 0) {
                                pozice[i] -= 2*S;
                                K += 2*S;
                        }
                        // a naplníme haldu
                        princ p = { i, pozice[i] };
                        halda.push(p);
                }

                // zkoušíme se přiblížit k hledanému princy po krocích velikosti S
                for ( ; K - S > 0; K -= S) {
                        princ p = halda.top();
                        halda.pop();

                        p.i = pozice[p.s] += S;
                        if (p.i < pocty[p.s]) {
                                halda.push(p);
                        }
                }
                S /= 2;
        }

        // hledaný princ je nyní na vrcholu haldy
        // došli jsme k němu po K-1 krocích velikosti 1
        princ hledany = halda.top();
        printf("%d %d\n", hledany.s+1, hledany.i+1);

        return 0;
}
```

P-III-3 Log-space programy

a) Nejprve si všimneme, že v logaritmickém prostoru dokážeme obejít jeden cyklus a najít jeho nejmenší prvek. Pak už stačí projít všechny prvky permutace a pro každý z nich zjistit, jestli je nejmenším prvkem cyklu, na němž leží. Takto každý cyklus započítáme právě jednou.

```
var n: integer;                          { zadaná permutace }
    P: array [1..100] of integer;
    c: integer;                          { počet nalezených cyklů }
    i, j, m: integer;                    { pomocné proměnné }
begin
  c := 0;
  for i := 1 to n do                     { i = zkoumaný prvek }
    begin
      j := i;                            { pomocí j kráčíme po cyklu }
      m := i;                            { m = nejmenší prvek cyklu }
      repeat
        if j < m then m := j;
        j := P[j];
      until j = i;
      if m = i then c := c+1;
    end;
end.
```

b) Jelikož mezi každými dvěma vrcholy u a v stromu existuje právě jedna cesta (bez opakování vrcholů), stačí zjistit, které hrany na této cestě leží, a spočítat je. Hranu na cestě z u do v přitom poznáme snadno: jejím odstraněním se u a v ocitnou v různých stromech.

Postačí tedy probírat jednu hranu za druhou a pokaždé spustit vzorové řešení úlohy P-II-4b z krajského kola, upravené tak, aby vybranou jednu hranu ignorovalo. K tomu nám jistě postačí logaritmické množství paměti.

Tím je úloha vyřešena. Popíšeme nicméně ještě jeden přístup, který nám dokonce dovolí cestu přímo sestrojit.

Opět vyjdeme z řešení úlohy P-II-4b. Připomeneme si, že jsme v něm postupným „obcházením“ hran stromu došli z vrcholu u do vrcholu v. Jinými slovy, sestrojili jsme *sled* z u do v – tak se říká posloupnosti na sebe navazujících hran, v níž se mohou vrcholy i hrany libovolně opakovat.

Pomocí skládání log-space programů, které jsme odvodili v domácím kole, zkombinujeme funkci generující sled s novou funkcí, která z libovolného sledu z u do v vyrobí cestu z u do v.

Mějme sled $u = t_1, t_2, \ldots, t_k = v$. Pokud se v něm žádný vrchol neopakuje, neopakují se ani hrany, takže máme cestu. V opačném případě najdeme nějaký vrchol w, který se opakuje. Nechť t_i je jeho první výskyt a t_j poslední. Potom $t_1, \ldots, t_{i-1}, t_j, \ldots, t_k$ je nějaký kratší sled z u do v (z původního sledu jsme „vystřihli“ uzavřený sled z w do w).

Náš algoritmus tedy bude procházet daný sled vrchol po vrcholu, pokaždé ověří, jestli se daný vrchol opakuje, a pokud ano, vystřihne všechny vrcholy od prvního výskytu do posledního. Jelikož vystřižením nemohou přibývat nová opakování, stačí sled projít jednou zleva doprava.

```
{ Vrátí i-tou hranu sledu nalezeného řešením P-II-4. Za koncem sledu vrací -1. }
function sled(i: integer): integer;

var d: integer;                          { nalezená vzdálenost }
    i, j, k: integer;                    { pomocné proměnné }

begin
  d := 0;
  i := 1;                                { i = pozice ve sledu }
  while sled(i) <> -1 do
    begin
      d := d+1;
      j := i;                            { hledáme další výskyty téhož vrcholu }
      k := sled(i);                      { k = aktuální vrchol }
      while sled(j) <> -1 do
        begin
          if sled(j) = k then
            i := j+1;
          j := j+1;
        end;
    end;

  writeln(d);
end.
```