P-II-1 Vlak

Nejdříve krátce zmíníme přímočaré řešení. Vyzkoušíme jednoduše odebrat postupně všechny možné podmnožiny vagónů a vybereme z nich minimální takovou, že zbylé vagóny tvoří palindrom. Toto řešení však vyžaduje vyzkoušení 2^N podmnožin, což je neúměrně mnoho.

Zkusme úlohu vyřešit po krocích. Předpokládejme, že umíme vyřešit případ, kdy nám magicky zmizely některé vagóny. Pokud si budeme vlak představovat jako posloupnost $V_{1\ldots N} = a_1 a_2 a_3 \ldots a_{N-1} a_N$, pak $V_{i\ldots j}$ je vlak tvořený $a_i a_{i+1} \ldots a_{j-1} a_j$.

Podívejme se, jak lze z řešení pro vlaky kratší délky vytvořit řešení pro celý vlak. Pokud $a_1 = a_N$, pak můžeme předpokládat, že se oba vagóny vyskytují v optimálním řešení pro celý vlak. Stačí tedy k řešení pro vlak $V_{2\ldots n-1}$ přidat vagóny a_1 a a_N. Pokud $a_1 \neq a_N$, pak se oba vagóny nemohou vyskytovat v řešení pro celý vlak a nejlepším řešením pro celý vlak je lepší z řešení pro vlaky $V_{1\ldots n-1}$ a $V_{2\ldots n}$.

Na této myšlence je postavený algoritmus, který běží v čase $\mathcal{O}(N^2)$. Úlohu vyřešíme dynamickým programováním přes všechny podvlaky. Na začátku víme, že vlak délky 1 je palindromický, a tedy je sám sobě optimálním řešením. Poté určíme řešení pro všechny podvlaky délky 2, pak délky 3 a nakonec nalezneme řešení pro celý vlak.

Kdyby nám stačila jen informace o největší možné délce vlaku, vystačili bychom s lineární pamětí – pamatovali bychom si jen řešení pro vlaky délek o jedna a o dvě menší. Nicméně když potřebujeme vypsat i konkrétní vynechané vagóny, vyroste nám paměť na kvadratickou.

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

#define MAX 50050

int N;
int ch = 0;
char vlak[MAX];
char novy_vlak[MAX];
struct podvlak {
    int prvni, posledni;
    int delka;
} **podvlaky;

int main(int argc, char ** argv)
{
    // přečteme vstup
    scanf("%d", &N);
    scanf("%s", vlak);

    // triviální vlak délky 1
    if (N == 1) { printf("0\n\n"); return 0; }
```

```
// inicializace -- podvlaky délky 1
podvlaky = calloc(N, sizeof(struct podvlak *));
podvlaky[0] = calloc(N, sizeof(struct podvlak));
for (int i=0;i<N;i++) {
    podvlaky[0][i].prvni = i;
    podvlaky[0][i].posledni = i;
    podvlaky[0][i].delka = 1;
}

// samotná dynamika
for (int d=1;d<N;d++) {
    podvlaky[d] = calloc(N-d, sizeof(struct podvlak));
    for (int j=0;j<N-d;j++) {
        // vyber lepší z předchozích podvlaků
        int ktery;
        if (podvlaky[d-1][j].delka > podvlaky[d-1][j+1].delka)
            ktery = j;
        else
            ktery = j+1;

        // na okrajích je stejný vagón
        if ((vlak[j] == vlak[j+d]) &&
            // zkusíme použít vlak, který obsahuje oba vagóny z okraje
            ((d == 1 ? 0 : podvlaky[d-2][j+1].delka) + 2 >
             podvlaky[d-1][ktery].delka)){
                podvlaky[d][j].delka = (d == 1 ? 0 :
                                        podvlaky[d-2][j+1].delka) + 2;
                podvlaky[d][j].prvni = j;
                podvlaky[d][j].posledni = j+d;
        }
        else {
            podvlaky[d][j].delka = podvlaky[d-1][ktery].delka;
            podvlaky[d][j].prvni = podvlaky[d-1][ktery].prvni;
            podvlaky[d][j].posledni = podvlaky[d-1][ktery].posledni;
        }
    }
}

int delka = 0;
int levy = podvlaky[N-1][0].prvni;
int pravy = podvlaky[N-1][0].posledni;

// Do nového vlaku nakopírujeme jen ty vagóny, které tvoří palindrom
while (1) {
    novy_vlak[podvlaky[pravy-levy][levy].prvni] =
        vlak[podvlaky[pravy-levy][levy].prvni];
    novy_vlak[podvlaky[pravy-levy][levy].posledni] =
        vlak[podvlaky[pravy-levy][levy].posledni];
    int lt = levy+1, rt = pravy-1;
    if ((delka += 2) >= podvlaky[N-1][0].delka) break;
    levy = podvlaky[rt-lt][lt].prvni;
    pravy = podvlaky[rt-lt][lt].posledni;
}

// Vypíšeme vagóny, které jsme vyhodili (indexovano od 1)
printf("%d\n", N - podvlaky[N-1][0].delka);
for (int d=0,i=0;i<N;i++) {
```

```
        if (novy_vlak[i] == 0) {
            printf("%d", i+1);
            if (++d + podvlaky[N-1][0].delka < N) printf(" ");
        }
    }
    printf("\n");
}
```

Poznámka: Kvadratické paměti se lze vyhnout i v obecném případě šikovným použitím metody Rozděl a panuj. Všimneme si, že když budeme vytvářet nejdelší palindrom postupným odpojováním vagónů od kraje vlaku (což přesně naše řešení dělá), nutně musíme potkat situaci, v níž bude délka aktuálního vlaku právě $\lfloor N/2 \rfloor$ nebo $\lfloor N/2 \rfloor + 1$.

Podobně jako jsme v prostoru $\mathcal{O}(N)$ dovedli spočítat, jak dlouhé je optimální řešení, můžeme v tomtéž prostoru zjistit i to, jak bude vypadat tento poloviční podvlak na cestě k optimálnímu řešení. Optimální řešení pro vlak $\alpha\beta\gamma$ (kde β je onen poloviční podvlak) se tedy musí skládat s optimálního řešení pro vlak β spojeného s optimálním řešením pro vlak $\alpha*\gamma$, kde $*$ je vagón, který není povoleno odebrat.

Jinými slovy, v čase $\Theta(N^2)$ jsme úlohu převedli na dvě úlohy poloviční velikosti, takže pro časovou složitost celého algoritmu musí platit rovnice $T(N) = \Theta(N^2) + 2T(N/2)$, která má řešení $T(N) = \Theta(N^2)$. Časová složitost je tedy stále kvadratická a prostoru jsme spotřebovali pouze lineárně.

P-II-2 Jabloňový sad

Asi nikoho moc nepřekvapí, že hledaným útvarem je konvexní obal množiny bodů v rovině – každý strom s jablky budeme pokládat za bod. Proč tomu tak je? Pokud bychom vzali jakýkoliv nekonvexní útvar, má jeho konvexní obal menší obvod a přitom ho celý obsahuje, a tedy i všechny zadané body. A konvexní obal je naopak podmnožinou všech konvexních útvarů, ve kterých leží zadané body.

Nalézt konvexní obal zadané množiny bodů v rovině je celkem známý problém, pro který existuje algoritmus pracující v čase $\mathcal{O}(N \cdot \log N)$.

Zopakujme si tedy některá tvrzení, která pro konvexní obal platí:

- Nejlevější a nejpravější bod (nejmenší a největší souřadnice x) leží na hraně obalu.
- Hrany u všech vrcholů svírají vnitřní úhel menší (nebo roven) 180°.

Konvexní obal můžeme rozdělit na horní a dolní část, začátkem obou částí bude nejlevější bod, koncem pak nejpravější. Ještě se pozastavme nad případem, že nejlevějších (resp. nejpravějších bodů) je více – příkladem je obdélník rovnoběžný s osami. Zakažme si tedy přítomnost svislých hran v konvexním obalu, protože se vyskytují pouze na levém a pravém okraji.

Za začátek horní resp. dolní části obalu (říkejme jim poloobaly) zvolíme nejhořejší resp. nejspodnější z nejlevějších bodů. Analogicky budeme postupovat pro konce těchto poloobalů. V případě obdélníku bude horním poloobalem pouze jeho horní strana rovnoběžná s osou x, dolním poloobalem pak spodní strana rovnoběžná

s osou x. Ptáte se: kam se poděly strany rovnoběžné s osou y? Tyto „opomenuté“ svislé hrany jsou vždy spojnicí začátků resp. konců horního a dolního poloobalu. Často se stane, že začátky těchto poloobalů splývají, délka této spojnice je potom nulová.

Kdybychom konvexní obal počítali jen jednou pro pevně zadanou množinu bodů, nejprve bychom si body setřídili zleva doprava a postupně jimi doplňovali horní a dolní poloobal. Fundamentální částí běžného algoritmu je řešení momentu, kdy jsme přidáním nového bodu do poloobalu způsobili, že náš předchůdce svírá vnitřní úhel větší než 180°. Tohoto předchůdce z poloobalu odebereme a tuto kontrolu zopakujeme na svém novém levém předchůdci. Takto můžeme klidně odebrat skoro všechny body, až kromě prvního, nejlevějšího.

My nahlédneme, že z tohoto postupu lze snadno vyjít. Po přidání nového bodu poopravíme daný poloobal jak směrem vlevo tak vpravo.

Zaměřme se na tuto otázku: Máme hotový konvexní obal (tedy oba poloobaly) a snažíme se přidat jeden bod. Pokud je přidaný bod uvnitř obalu, není co řešit. Pokud je vně, budeme muset obal o tento bod doplnit. Dokonce lze s jistotou říci, že bude jedním z vrcholů nového konvexního obalu.

Předpokládejme tedy, že nový bod leží nad horním poloobalem; pro spodní poloobal budeme postupovat symetricky. Protože body na poloobalu jdou za sebou přesně podle pořadí svých x-ových souřadnic, podíváme se nejprve, mezi které dva body poloobalu se nový bod začlení. Zatím předpokládejme, že nový bod není ani nejlevějším ani nejpravějším bodem ze všech, které budou v horním poloobalu. Přidáním nového bodu do posloupnosti bodů na horním poloobalu se mohly stát dvě věci: buď všechny vrcholy svírají vnitřní úhel menší (nebo roven) 180° a všechno je v pořádku, nebo se tento předpoklad u některého bodu porušil. Jediné dva body, u kterých může být nyní úhel větší než 180°, je přímý předchůdce (levý soused) a přímý následník (pravý soused) našeho nového bodu. Co se týká levého předchůdce, nápravu této chyby už známe z předchozího algoritmu – budeme prostě své přímé předchůdce zahazovat tak dlouho, dokud bude jejich vnitřní úhel nepřípustný. Pro pravého souseda budeme postupovat analogicky.

Snadno se dovtípíme, že pokud bude levý i pravý soused po aplikaci tohoto postupu svírat správný úhel, máme poloobal korektně doplněný o nový bod. Nově přidaný bod totiž z principu nemůže svírat nepřípustný úhel a u ostatních bodů se nic nezměnilo. Při každém přidání, resp. odebrání vrcholu si stačí pouze průběžně ukládat, o kolik se změnil obvod konvexního poloobalu.

Pokud byl nově přidaný bod nejlevější resp. nejpravější, můžeme prostě vynechat kontrolu vnitřního úhlu pro odpovídající (neexistující) sousedy. Jenom mějme na paměti, že jsme si zakázali svislé hrany v poloobalu, takže pokud například nový nejlevější bod má stejnou souřadnici x jako levý začátek horního poloobalu a přitom neleží výše, bez obav ho zahodíme.

Nyní se podívejme, jak tento algoritmus implementovat. Předně potřebujeme umět rychle najít své levé a pravé sousedy podle souřadnice x, dále pak chceme umět rychle odebírat a přidávat body. Pole nebo seznam zřejmě nejsou dobrou vol-

bou. Naopak binární vyhledávací strom nám plně vyhovuje. Budeme si tedy každý z poloobalů udržovat v jednom takovém vyhledávacím stromě (můžeme zvolit AVL, Červeno-černý, ... záleží pouze na časové složitosti jeho operací).

Abychom nemuseli pro horní poloobal zkoumat, co leží nad ním, a naopak pro dolní poloobal, co leží pod ním, provedeme jednoduchý trik. Dolní poloobal si otočíme podle osy x a jednoduše s ním budeme pracovat identicky jako s horním.

Ještě jedna technická poznámka: svislé hrany jsme si zakázali právě proto, aby nám nedělaly problémy při uspořádání bodů ve stromě. Všimněme si totiž, že nejlevější body by se musely třídit vzestupně podle y, naopak nejpravější body sestupně.

Celý algoritmus tedy funguje takto:

Na počátku si vezmeme první bod a vytvoříme z něho oba konvexní poloobaly, leč degenerované. (Po přidání druhého bodu budou oba poloobaly totožné úsečky, ale ani to nám nevadí.) Pak pouze přidáváme bod po bodu a doplňujeme tyto poloobaly podle potřeby. Přitom se díváme, jak se mění obvod celého obalu. Je už jen triviálním krokem, že pro body zadané na počátku vypíšeme celkový obvod konvexního obalu a pro následujících M bodů vypisujeme pouze změnu obvodu. Technickou drobností je, že přitom musíme kalkulovat se svislými spojnicemi začátků a konců horního a dolního poloobalu.

Jak efektivní náš algoritmus bude?

Každému z $N + M$ bodů vhodně naúčtujeme kroky výpočtu a uvidíme, že výsledek je překvapivě uspokojivý. Vložení nového bodu naúčtujeme právě jemu, tedy odhadem $\mathcal{O}(\log(N + M))$. Kontrolu vnitřního úhlu, která dopadla dobře, a nalezení sousedů těchto sousedů, pro které kontrola dopadla dobře, rovněž napočítáme nově vkládanému bodu – $\mathcal{O}(\log(N + M))$. Naopak neúspěšnou kontrolu a následné odebrání odpovídajícího bodu naúčtujeme odebíranému bodu – $\mathcal{O}(\log(N + M))$. Každá operace je zaúčtována, zároveň každý bod je přidán a odebrán maximálně jednou. Tedy součtem přes všechny body dostáváme $\mathcal{O}((N + M) \cdot \log(N + M))$. Zajímavé je, že stejnou složitost má i zmíněný standardní algoritmus pro konvexní obal, pokud bychom ho spustili pouze na konci na všechny body dohromady.

Paměťová efektivita je rovněž $\mathcal{O}(N + M)$, protože pro každý poloobal si držíme jeden vyhledávací strom, který může v nejhorším případě obsahovat všechny body (přesněji toto může nastat leda pro oba stromy v součtu, ale pro horní odhad nám to stačí).

Vzorové řešení je naprogramováno v jazyce C++, kde jsme si ušetřili implementaci vyhledávacích stromů použitím typu `set` z STL.

```
#include <stdio.h>
#include <math.h>
#include <set>

using namespace std;

struct Point {

    double x;
```

```
    double y;

    // překlopení podle osy x
    Point flip() const
    {
        Point pt;
        pt.x = x;
        pt.y = -y;
        return pt;
    }

    // vždy chceme třídit horizontálně podle x, dále podle y
    bool operator< (const Point& pt) const
    {
        return x < pt.x || (x == pt.x && y < pt.y);
    }

    double distanceTo(const Point& p) const
    {
        return sqrt((x - p.x) * (x - p.x) + (y - p.y) * (y - p.y));
    }
};

typedef set<Point> PointMap; // uspořádaná množina bodu

Point scanpt()
{
    Point pt;
    double x, y;
    scanf("%lf %lf\n", &x, &y);
    pt.x = x;
    pt.y = y;
    return pt;
}

// Je směrnice (a -> p) větší než (a -> b) ?
bool Above(Point p, Point a, Point b)
{
    return (p.y - a.y) * (b.x - a.x) > (b.y - a.y) * (p.x - a.x);
}

double AddPoint(PointMap& envelope, Point pt)
{
    PointMap::iterator rightNeighbour = envelope.lower_bound(pt);

    PointMap::iterator insertPoint = envelope.end();

    double delta = 0;

    if(rightNeighbour == envelope.end()) // nový pravý okraj?
    {
        PointMap::iterator leftNeighbour = envelope.end();
        leftNeighbour--;
        delta = leftNeighbour->distanceTo(pt);

        insertPoint = envelope.insert(pt).first;
    } else
    if(rightNeighbour == envelope.begin()) // nový levý okraj?
    {
        if(rightNeighbour->x == pt.x)
```

```
        // tzn. tento bod je přímo pod původním levým okrajem
        return 0; // zahodím ho

    delta = envelope.begin()->distanceTo(pt);
    insertPoint = envelope.insert(pt).first;
}
else
{
    PointMap::iterator leftNeighbour(rightNeighbour);
    leftNeighbour--;
    if(Above(pt, *leftNeighbour, *rightNeighbour))
    {
        // aha, bod nepatří dovnitř, takže zvětšíme obal
        delta = leftNeighbour->distanceTo(pt) + pt.distanceTo(*rightNeighbour)
                - leftNeighbour->distanceTo(*rightNeighbour);
        insertPoint = envelope.insert(pt).first;
    }
}

if(insertPoint != envelope.end())
{
    // Kontrola obálky směrem vlevo
    if(insertPoint != envelope.begin())
    {
        PointMap::iterator prev = insertPoint;
        prev--;
        while(prev != envelope.begin())
        {
            PointMap::iterator pprev = prev;
            pprev--;
            // body jsou za sebou v tomto pořadí: pprev -> prev -> insertPoint

            if(!Above(*prev, *pprev, *insertPoint))
            {
                delta -= pprev->distanceTo(*prev)
                         + prev->distanceTo(*insertPoint)
                         - pprev->distanceTo(*insertPoint);
                envelope.erase(prev);
                insertPoint = envelope.find(pt);
                prev = insertPoint;
                prev--; // tj. vlastně pprev se stane předchůdcem insertPointu
            } else
                break; // dál už je to v pořádku
        }
    }
    if(insertPoint != --envelope.end())
    {
        PointMap::iterator next = insertPoint;
        next++;
        while(next != --envelope.end())
        {
            PointMap::iterator nnext = next;
            nnext++;
            // body jsou za sebou v tomto pořadí: insertPoint -> next -> nnext

            if(!Above(*next, *insertPoint, *nnext))
            {
```

```
                    delta -= insertPoint->distanceTo(*next)
                             + next->distanceTo(*nnext)
                             - insertPoint->distanceTo(*nnext);
                    envelope.erase(next);
                    insertPoint = envelope.find(pt);
                    next = insertPoint;
                    next++; // tj. vlastně nnext se stane následníkem insertPointu
                } else
                    break; // dál už je to v pořádku
            }
        }
    }

    return delta;
}

double EndsDistance(PointMap& upper, PointMap& lower)
{
    PointMap::iterator lowerEnd = lower.end(), upperEnd = upper.end();
    lowerEnd--;
    upperEnd--;

    return upper.begin()->distanceTo(lower.begin()->flip())
           + upperEnd->distanceTo(lowerEnd->flip());
}

int main()
{
    int m, n;

    PointMap upper, lower; // upravovatelná horní a dolní obálka množiny bodu

    scanf("%d\n", &n);

    Point pt = scanpt();
    upper.insert(pt);
    lower.insert(pt.flip());

    double sum = 0, delta = 0;

    for(int i = 1; i < n; i++)
    {
        pt = scanpt();

        delta = -EndsDistance(upper, lower);
        delta += AddPoint(upper, pt);
        delta += AddPoint(lower, pt.flip()); // body
        delta += EndsDistance(upper, lower);

        sum += delta;
    }
    printf("%.5f\n", sum);

    scanf("%d", &m);

    for(int i = 0; i < m; i++)
    {
        pt = scanpt();

        delta = -EndsDistance(upper, lower);
        delta += AddPoint(upper, pt);
        delta += AddPoint(lower, pt.flip());
```

```
        delta += EndsDistance(upper, lower);
        printf("%.5f\n", delta);
    }
    return 0;
}
```

P-II-3 Mažoretky

Pořadí následující den

Nechť pořadí mažoretek na vstupu je $(a_1, \ldots, a_N)$. Pokud $a_1 > \cdots > a_N$, pak pořadí odpovídá poslednímu dni v roce a následující den budou mažoretky pochodovat v pořadí $(1, \ldots, N)$. Předpokládejme nyní, že pořadí neodpovídá poslednímu dni v roce, a zamysleme se, jak vypadají pořadí v jeho zbylých dnech.

Uvažme nyní dvě pořadí $(b_1, \ldots, b_N)$ a $(b'_1, \ldots, b'_N)$, která se v témže roce objeví později. Označme i nejmenší index s $a_i \neq b_i$ a i' nejmenší index s $a_i \neq b'_i$. Pokud $i < i'$, pak $a_i = b'_i < b_i$ a tedy mažoretky v pořadí $(b'_1, \ldots, b'_N)$ pochodují dříve, než v pořadí $(b_1, \ldots, b_N)$. Pro pořadí mažoretek následující hned po $(a_1, \ldots, a_N)$ proto platí, že mezi všemi pořadími, která následují v aktuálním roce po tomto pořadí, má s pořadím $(a_1, \ldots, a_N)$ nejdelší počáteční úsek.

Zaměřme se na to, jak takový úsek nalézt. Pokud pro nějaké i platí, že se mezi čísly $a_{i+2}, \ldots, a_N$ vyskytuje číslo větší než a_{i+1}, pak existuje pořadí, které následuje po $(a_1, \ldots, a_N)$ a má s tímto pořadím společný počáteční úsek délky i. Musíme tedy najít nejmenší index i takový, že $a_{i+1} > \cdots > a_N$.

Pokud takový index i nalezneme, pak a_i nahradíme nejmenším číslem mezi $a_{i+1}, \ldots, a_N$, které je větší než a_i, a následující čísla seřadíme podle velikosti od nejmenšího.

Popsaný algoritmus lze implementovat v lineárním čase, a to následovně. Od konce zadané posloupnosti nalezneme nejmenší index i takový, že $a_{i+1} > \cdots > a_N$. Pokud je $i = 0$, pak $a_1 > \cdots > a_N$ a vypíšeme posloupnost $1, \ldots, N$. Předpokládejme tedy, že i není nula. Mezi čísly $a_{i+1}, \ldots, a_N$ nalezneme nejmenší číslo větší než a_i (takové existuje, neboť $a_{i+1} > a_i$) a prohodíme ho s a_i. Povšimněme si, že úsek posloupnosti po a_i je stále sestupně seřazen. Stačí ho tedy nyní jen zrcadlově otočit a výsledné pořadí vypsat.

```
void dalsi_permutace(vector<int> &A)
{
  int x = A.size() - 2;
  while (x >= 0 && A[x] >= A[x+1])
    x--;
  if (x >= 0)                        // Není to případ poslední -> první
    {
      int y = A.size() - 1;
      while (A[y] <= A[x])
        y--;
      swap(A[x], A[y]);              // Prohodíme
    }
```

```
  reverse(A.begin()+x+1, A.end());  // Otočíme
}
```

Pořadí za půl roku

Opět označme $(a_1, \ldots, a_N)$ pořadí zadané na vstupu. Vyřešme nejdříve úlohu pro sudá N. Počet pořadí se shodným prvním číslem je $(N-1)!$. Pokud $a_1 \leq N/2$, pak hledané pořadí začíná číslem $b_1 = a_1 + N/2$; pokud $a_1 > N/2$, pak začíná číslem $b_1 = a_1 - N/2$. A pokud je mezi pořadími, která začínají a_1, pořadí $(a_1, \ldots, a_N)$ k-té, pak hledáme k-té pořadí, které začíná b_1. Takové ale snadno nalezneme – pokud a_i, $2 \leq i \leq N$, je ℓ-tý nejmenší prvek mezi $\{1, \ldots, N\} \setminus \{a_1\}$, pak b_i bude ℓ-tý nejmenší prvek mezi $\{1, \ldots, N\} \setminus \{b_1\}$.

Jestliže tedy $a_1 \leq N/2$, pak pořadí $(b_1, \ldots, b_N)$ za půl roku bude

$$b_i = \begin{cases} a_1 + N/2 & \text{pokud } i = 1, \\ a_i - 1 & \text{pokud } a_1 < a_i \leq a_1 + N/2, \\ a_i & \text{jinak.} \end{cases}$$

Analogický vzorec platí, jestliže $a_1 > N/2$.

Pro lichá N je situace komplikovanější. Potřebujeme určit pořadí za $N!/2$ dní. Nejdříve určíme pořadí za $\lfloor N/2 \rfloor (N-1)!$ dní stejně jako v případě, kdy N bylo sudé (místo $N/2$ k a_1 přičteme $\lfloor N/2 \rfloor$ a případně odečteme N).

Nyní potřebujeme určit pořadí za dalších $N!/2 - \lfloor N/2 \rfloor (N-1)! = (N-1)!/2$ dní. Tento posun však ovlivňuje posledních $N-1$ mažoretek a protože $N-1$ je sudé číslo, můžeme použít postup pro sudé N. Musíme však být opatrní, pokud je hodnota a_2 velká – během $(N-1)!/2$ dní nastane (jedna) změna i na první pozici. Například když $N = 5$ a $(a_1, \ldots, a_5) = (1, 4, 2, 5, 3)$, pak pořadí za $\lfloor N/2 \rfloor (N-1)!$ dní je $(3, 4, 1, 5, 2)$ a za dalších $(N-1)!/2$ dní je pak $(4, 1, 2, 5, 3)$. Tomuto problému se však můžeme vyhnout tak, že pokud a_2 je velké, pak k a_1 na začátku přičteme $\lceil N/2 \rceil$ a k získanému pořadí budeme hledat to o $(N-1)!/2$ dní dříve.

Implementace výše uvedeného řešení s lineární časovou a paměťovou složitostí je přímočará a kód programu následuje.

```
int cyklicky(int x, int N)
{ if (x>N) return x-N; return x; }

void prejmenuj(vector<int> &A, int stare, int nove)
{
  for (int n=1; n < int(A.size()); n++)
    {
      if (stare < nove && A[n] > stare && A[n] <= nove)
        A[n]--;
      if (stare > nove && A[n] >= nove && A[n] < stare)
        A[n]++;
    }
}

void opacna_permutace(vector<int> &A)
{
  int N = A.size();
  int old = A[0];
```

```
  A[0] = cyklicky(A[0] + N/2, N);
  prejmenuj(A, old, A[0]);
  if (N % 2 == 0)
    return;

  int velke = (N+1)/2 + (A[0] <= N/2);
  if (A[1] >= velke)
    {
      old = A[0];
      A[0] = cyklicky(A[0] + 1, N);
      prejmenuj(A, old, A[0]);
    }

  prejmenuj(A, A[0], N);
  vector<int> B(A.begin()+1, A.end());
  opacna_permutace(B);
  for (int n=1; n<N; n++)
    A[n]=B[n-1];
  prejmenuj(A, N, A[0]);
}
```

P-II-4 Grafový počítač na kliku

a) Pokusíme se opět o konstrukci zdvojováním grafu. Z úplného grafu (neboli *kliky*) velikosti $n/2$ budeme vytvářet kliku velikosti n, takže od triviální kliky dospějeme k té požadované za $\mathcal{O}(\log n)$ kroků, z nichž každý bude trvat konstantně dlouho.

Nejdříve předpokládejme, že n je mocninou dvojky. Budeme vyrábět kliky speciálního druhu, totiž takové, že $n/2$ vrcholů bude označeno značkou i a zbylých $n/2$ značkou j pro nějaké $i \neq j$. Takovým grafům budeme říkat $G_n(i,j)$. Všimněte si, že z $G_n(i,j)$ umíme operací `ReplaceV` vytvořit $G_n(i',j')$ pro libovolné i', j'. Nyní ukážeme, jak z $G_n(1,2)$ vyrobit $G_{2n}(1,2)$:

- Nejprve provedeme `Join` grafů $G_n(1,2)$ a $G_n(3,4)$. Tím vznikne graf se $2n$ vrcholy rozdělený na 4 stejně velké části se značkami 1, 2, 3, 4. Uvnitř každé části jsou vrcholy spojené každý s každým. Stejně tak každý vrchol v části 1 s každým v části 2 a podobně mezi částmi 3 a 4. Není to ovšem klika, protože mezi zbývajícími dvojicemi částí hrany chybí.
- `Join`em předchozího grafu s $G_n(1,3)$ doplníme všechny hrany mezi částmi 1 a 3. (Každý vrchol grafu $G_n(1,3)$ bude ztotožněn s nějakým vrcholem předchozího grafu, takže přibudou pouze hrany.)
- Podobně provedením `Join`ů s grafy $G_n(1,4)$, $G_n(2,3)$ a $G_n(2,4)$ doplníme všechny zbývající hrany.
- Již máme úplný graf. Pomocí `Replace` tedy přejmenujeme značku 3 na 1 a 4 na 2 a získáme kýžený $G_{2n}(1,2)$.

To dává přímočarý rekurzivní algoritmus s časovou složitostí $\mathcal{O}(\log n)$.

Zbývá tento postup upravit, aby fungoval, i když n není mocninou dvojky. Lichá n jsou problematická, protože pro ně nedává smysl ani definice $G_n(i,j)$, a tak

je ještě na chvíli odložme. Pro sudá n definice funguje, ale rekurze se bude ptát po grafu na $n/2$ vrcholech, což už může být liché číslo.

Použijeme proto malý trik: Vždy zaokrouhlíme n na nejbližší vyšší násobek čtyř (říkejme mu n'), sestrojíme $G_{n'}(1,2)$ a pak smažeme nejvýše 3 přebytečné vrcholy. Tehdy se rekurze bude odvolávat na grafy velikosti $n'/2$, což je jistě sudé. Konstrukce funguje, zbývá nahlédnout, že jsme nepokazili časovou složitost.

Především: Nemůže se algoritmus zacyklit? Je skutečně pravda, že $n'/2 < n$? Vždy platí, že $n' \leq n+3$, takže stačí ukázat, že $(n+3)/2 < n$. Kdykoliv $n > 3$, tato nerovnost platí; $n = 3$ nás nezajímá, neboť je liché. Rekurzi tedy zastavíme na $n = 2$.

Podobně ukážeme, že pro $n \geq 6$ je $n'/2 \leq (n+3)/2 \leq 3/4 \cdot n$. První rekurzivní volání tedy proběhne pro graf velikosti nejvýše $3/4 \cdot n$, další pro $(3/4)^2 \cdot n$, atd., až po $\mathcal{O}(\log n)$ krocích graf zmenšíme na méně než 6 vrcholů, pro které už kliku sestrojíme v konstantním čase. Celý algoritmus má tudíž logaritmickou časovou složitost.

Nakonec si všimneme, že spustíme-li algoritmus pro liché n, chová se přesně tak, jako by sestrojil kliku na $n+1$ vrcholech (což už je sudé) a pak jeden z vrcholů smazal. Tyto případy tedy nemusíme zvlášť ošetřovat.

Následující program pracuje přesně podle uvedeného algoritmu.

```
function klika(n: Integer): Graph;      { Sestrojí graf Gn(1,2) }
var e, f, g, h: Graph;
    nn: Integer;
begin
   if n=2 then begin                    { Ošetříme triviální případ }
      g := EmptyG;
      AddV(g, 1);
      AddV(g, 2);
      AddE(g, 1, 2, undef);
      klika := g;
      exit;
      end;
   nn := n;                             { Zaokrouhlíme na násobek 4 }
   while nn mod 4 <> 0 do inc(nn);
   g := klika(nn div 2);                { Gn(1,2) }
   h := g;
   e := g;
   ReplaceV(h, 1, 3);
   ReplaceV(h, 2, 4);                   { h = Gn(3,4) }
   g := Join(g, h, none, none);
   f := e; ReplaceV(f, 2, 3);           { f = Gn(1,3) }
   g := Join(g, f, value, any);
   f := e; ReplaceV(f, 2, 4);           { f = Gn(1,4) }
   g := Join(g, f, value, any);
   f := e; ReplaceV(f, 1, 3);           { f = Gn(2,3) }
   g := Join(g, f, value, any);
   f := e; ReplaceV(f, 1, 4);           { f = Gn(2,4) }
   g := Join(g, f, value, any);
   ReplaceV(g, 3, 1);
   ReplaceV(g, 4, 2);
   while nn > n do begin                { Smažeme přebytečné vrcholy }
```

```
      DelV(g, nn);
      dec(nn);
      end;
   klika := g;
end;
```

b) Pro libovolné k umíme ověřit, zda zadaný graf obsahuje kliku na k vrcholech: předchozím algoritmem takovou kliku sestrojíme a použijeme operaci `Find`. Stačilo by tedy postupně zkoušet $k = 1, \ldots, n$, ale rychleji to půjde binárním vyhledáváním.

Předpokládejme, že se hledaná klikovost nachází v nějakém intervalu $\langle a, b\rangle$. (Na začátku zvolíme $a = 1$, $b = n$.) Ověříme, zda se v grafu nachází klika velikosti „uprostřed intervalu“, tedy o $k = \lceil (a+b)/2 \rceil$ vrcholech. Pokud ne, víme, že hledaná klikovost leží v intervalu $\langle a, k-1\rangle$. Pokud ano, leží v $\langle k, b\rangle$. Tím jsme hledání omezili na zhruba poloviční interval, takže po zhruba logaritmickém počtu kroků dospějeme k intervalu jednotkové velikosti a klikovost nalezneme.

Během jednoho kroku přitom strávíme čas $\mathcal{O}(\log n)$ konstrukcí kliky, takže celková časová složitost algoritmu činí $\mathcal{O}(\log^2 n)$.

Než předvedeme program, měli bychom ještě uvést na pravou míru ono „zhruba poloviční“ v odhadu složitosti a spočítat kroky exaktněji. Označme ℓ velikost aktuálního intervalu, tedy $\ell = b - a + 1$. Každý z podintervalů potom bude mít nejvýše $\ell' = \lceil \ell/2 \rceil$ prvků, což můžeme shora odhadnout výrazem $(\ell + 3)/2$ a použít tytéž nerovnosti jako v podúloze a). Jen musíme rozmyslet případy s $\ell \leq 3$: Pro $\ell = 1$ se ihned zastavíme. Když je $\ell = 2$, oba podintervaly mají velikost 1 a také se zastavíme. Při $\ell = 3$ nás zachrání, že střed intervalu zaokrouhlujeme nahoru, takže oba podintervaly budou mít velikost nejvýše 2.

```
function klikovost(var g: Graph): Integer;
var a, b, k: Integer;
    h, w : Graph;
begin
   a := 1;                              { Počáteční interval }
   b := CountV(g);
   while a < b do begin                 { Binárně hledáme }
      k := (a+b+1) div 2;               { Střed intervalu }
      h := klika(k);                    { Existuje taková klika? }
      w := Find(g, h, IM_any, IM_any);
      if CountV(w)=0 then               { Ne }
         b := k-1
      else                              { Ano }
         a := k;
      end;
   klikovost := a;
end;
```

Toto řešení ovšem není optimální. Ukážeme, že testovanou kliku není potřeba vytvářet pokaždé od základu, takže algoritmus můžeme zrychlit na $\mathcal{O}(\log n)$.

Nejprve si všimneme, že jednoduchou úpravou funkce `klika` můžeme v čase $\mathcal{O}(\log n)$ sestrojit kliky všech velikostí $1, 2, 4, 8, \ldots, 2^k$, kde k je nejmenší mocnina dvojky větší než n.

```
var kliky: array [0..MaxLog] of Graph;
    pocet_klik: Integer;

procedure vyrob_kliky(limit: Integer);
var e, f, g, h: Graph;
begin
   kliky[0] := EmptyG;
   AddV(kliky[0], 1);

   kliky[1] := EmptyG;
   AddV(kliky[1], 1);
   AddV(kliky[1], 2);
   AddE(kliky[1], 1, 2, undef);

   pocet_klik := 1;
   repeat
      { Indukční krok stejný jako v předchozím řešení }
      g := kliky[pocet_klik];
      h := g;
      e := g;
      ReplaceV(h, 1, 3);
      ReplaceV(h, 2, 4);
      g := Join(g, h, none, none);
      f := e; ReplaceV(f, 2, 3);
      g := Join(g, f, value, any);
      f := e; ReplaceV(f, 2, 4);
      g := Join(g, f, value, any);
      f := e; ReplaceV(f, 1, 3);
      g := Join(g, f, value, any);
      f := e; ReplaceV(f, 1, 4);
      g := Join(g, f, value, any);
      ReplaceV(g, 3, 1);
      ReplaceV(g, 4, 2);
      inc(pocet_klik);
      kliky[pocet_klik] := g;
   until CountV(g) > limit;
end;
```

Druhá důležitá ingredience bude „odečtení“ dvou klik: pokud máme kliky K_p a K_q velikostí p a q ($p \geq q$), můžeme z nich v konstantním čase vyrobit kliku na $p-q$ vrcholech takto: Všechny vrcholy kliky K_p opatříme značkou 1, všechny vrcholy K_q značkou 2. Nyní provedeme `Join`$(K_q, K_p,$ `any, any`$)$ – tím vznikne klika velikosti p, ve které bude mít q vrcholů značku 2 a zbylých $p-q$ vrcholů značku 1. Operací `Common` pak najdeme největší společný podgraf tohoto grafu s klikou K_p, což je hledaná klika K_{p-q}.

```
function odecti_kliky(kp, kq: Graph): Graph;
var g: Graph;
begin
   SetAllV(kp, 1);
   SetAllV(kq, 2);
   g := Join(kq, kp, any, any);
   odecti_kliky := Common(g, kp, value, any);
end;
```

Nyní upravíme hledání půlením intervalu tak, aby v každém kroku pracovalo s intervalem velikosti nějaké mocniny dvojky. Začneme intervalem $\langle 0, 2^k \rangle$, v něm se řešení určitě nachází. V každém dalším kroku pak řešení leží v nějakém intervalu $\langle a, a + 2^i \rangle$. Vždy zjistíme, zda se v grafu nalézá klika velikosti $t = a + 2^{i-1}$. Pokud ano, přesuneme se do intervalu $\langle t, a + 2^i \rangle = \langle t, t + 2^{i-1} \rangle$. V opačném případě omezíme hledání na interval $\langle a, t) = \langle a, a + 2^{i-1} \rangle$.

Budeme-li si průběžně udržovat kliku velikosti $a + 2^i$, můžeme z ní vždy odečtením kliky na 2^{i-1} vrcholech sestrojit kliku velikosti t, takže jeden krok půlení provedeme v konstantním čase. Celkem tedy strávíme čas $\mathcal{O}(\log n)$ předvýpočtem klik a $\mathcal{O}(\log n)$ půlícími kroky.

```
function klikovost(g: Graph): Integer;
var a, k: Integer;
    bk, t, w: Graph;
begin
   vyrob_kliky(CountV(g));              { Připrav kliky }
   {
      V každém průchodu cyklem bude platit:
        - řesení je v intervalu <a,b), kde b=a+2^k
        - bk je klika velikosti b
   }
   a := 0;
   k := pocet_klik;
   bk := kliky[k];
   while k > 0 do begin
      dec(k);
      t := odecti_kliky(bk, kliky[k]);  { t je klika velikosti a+2^(k-1) }
      w := Find(g, t, any, any);        { nachází se v grafu g? }
      if CountV(w) <> 0 then
         a := CountV(t)                 { ano -> interval <a+2^(k-1),b) }
      else
         bk := t;                       { ne -> interval <a,a+2^(k-1)) }
      end;
   klikovost := a;
end;
```

Na závěr ještě dodejme, že paměťovou náročnost $\mathcal{O}(\log n)$ bychom ještě mohli snížit na $\mathcal{O}(1)$ tím, že bychom místo mocnin dvojky používali Fibonacciho čísla ($F_0 = 1$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$) a pracovali s intervaly tvaru $\langle a, a + F_i \rangle$. Jelikož čísla F_n rostou řádově exponenciálně rychle (indukcí snadno dokážeme, že $F_n \geq 2^{n/2}$ pro $n \geq 6$), bude kroků vyhledávání nadále $\mathcal{O}(\log n)$. A jelikož $F_{n+2} - F_{n+1} = F_n$, můžeme se po posloupnosti klik velikostí Fibonacciho čísel pohybovat tam i zpět v konstantním čase, aniž bychom si jich museli všech $\Theta(\log n)$ pamatovat.