ОСНОВНІ ПОНЯТТЯ АЛГОРИТМІСТИКИ

Етапи розв’язування задач на ЕОМ

Розв’язання будь-якої задачі на ЕОМ складається з кількох етапів, а саме: – постановка завдання; – формалізація (математична постановка задачі); – вибір (або розроблення) методу розв’язування; – розроблення алгоритму; – складання програми; – налагодження програми; – обчислення та обробка результатів.

Поняття алгоритму

Алгоритм – система формальних правил, що визначає зміст і порядок дій над вхідними даними і проміжними результатами, необхідними для отримання кінцевого результату при розв’язуванні задачі.

Кожен алгоритм є списком добре визначених інструкцій для розв'язання задачі. Починаючи з початкового стану, інструкції алгоритму описують процес обчислення, які відбуваються через послідовність станів, які, зрештою, завершуються кінцевим станом. Перехід з одного стану до наступного не обов'язково детермінований — деякі алгоритми містять елементи випадковості.

Представлення алгоритмів

У процесі розробки алгоритму можуть використовуватись різні способи його опису, які відрізняються за простотою, наочністю, компактністю, мірою формалізації, орієнтації на машинну реалізацію тощо.

Форми запису алгоритму:

словесна або вербальна (мовна, формульно-словесна); псевдокод (формальні алгоритмічні мови); схемна: (структурограми (схеми Нассі-Шнайдермана); графічна (блок-схема, виконується за вимогами стандарту).

Алгоритми мають такі властивості:

1. *Скінченність* — алгоритм має завжди завершуватись після виконання скінченної кількості кроків. Процедуру, яка має решту характеристик алгоритму, без, можливо, скінченності, називають методом обчислень.
2. *Дискретність* — процес, що визначається алгоритмом, можна розчленувати (розділити) на окремі елементарні етапи (кроки), кожен з яких називається кроком алгоритмічного процесу чи алгоритму.

3. *Визначеність* — кожен крок алгоритму має бути точно визначений. Дії, які необхідно здійснити, повинні бути чітко та недвозначно визначені для кожного можливого випадку.
4. *Масовість* — властивість алгоритму, яка полягає в тому, що алгоритм повинен забезпечувати розв'язання будь-якої задачі з класу однотипних задач за будь-якими вхідними даними, що належать до області застосування алгоритму.
5. *Результативність* — вказує на наявність таких варіантів вхідних даних, для яких обчислювальний процес, що реалізується за наданим алгоритмом, повинен через скінчену кількість етапів (кроків) зупинитись і дати шуканий результат або сигнал про те, що наданий алгоритм непридатний для розв’язання поставленої задачі.
6. *Ефективність* — алгоритм вважають ефективним, якщо всі його оператори досить прості для того, аби їх можна було точно виконати за скінченний проміжок часу з допомогою олівця та аркушу паперу.

Алгоритміка — це дисципліна, що вивчає алгоритми, та їх застосування до розв'язування задач. В практичному плані алгоритміка використовується для підготовки до участі у змаганнях зі спортивного програмування.

Алгоритміка відрізняється від теорії алгоритмів тим, що не займається пошуком доведення існування алгоритму, а займається пошуком оптимального (в основному за часом виконання), алгоритму, що розв'язує дану задачу. Якщо такий алгоритм невідомий, то робиться спроба розв'язати задачу хоча б частково.

Обчислювальна складність. Поширеним критерієм оцінки алгоритмів є час роботи та порядок зростання тривалості роботи в залежності від обсягу вхідних даних. Кожній конкретній задачі зіставляють деяке число, яке називають її розміром. Наприклад, розміром задачі обчислення добутку матриць може бути найбільший розмір матриць-множників, для задач на графах розміром може бути кількість ребер графа. Час, який витрачає алгоритм як функція від розміру задачі n, називають часовою складністю цього алгоритму $T(n)$. Асимптотику поведінки цієї функції при збільшенні розміру задачі називають асимптотичною часовою складністю, а для її позначення використовують нотації Ландау та Кнута.

Грубо кажучи, аналіз середньої асимптотичної часової складності можна поділити на два типи: аналітичний та статистичний.

АЛГОРИТМИ РОЗВ'ЯЗУВАННЯ ЗАДАЧ АРИФМЕТИКИ

Обчислення суми цифр натурального числа у системі числення із заданою основою

```
function SumOfDigits(N: Integer; const Base: Integer): Integer;
var  S: Integer;
begin
    S := 0;
    while N > 0 do
    begin
        S := S + (N mod Base);
        N := N div Base;
    end;
    SumOfDigits := S;
end;
```

Задача. Написати функцію переведення цілого числа у систему числення із заданою основою. Проаналізувати часову складність алгоритму.

Бінарний алгоритм піднесення до степеня

1. **Рекурсивна форма:**

```
function PowerRec(const a: Int64; n: UInt64): Int64;
begin
    if N > 0 then
        if Odd(n) then
            PowerRec := a*Sqr(PowerRec(a, n div 2))
        else
            PowerRec := Sqr(PowerRec(a, n div 2))
    else
        PowerRec := 1;
end;
```

2. **Ітеративна форма:**

```
function Power(const a: Int64; n: Int64): Integer;
var  P, x: Int64;
begin
    P := 1;
    x := a;
    while n > 0 do
    begin
        if Odd(n) then
            P := P * x;
        n := n div 2; // n := n shr 1;
        x := x*x;
    end;
    Power := P;
end;
```

Твердження. Якщо $\overline{1a_1a_2\ldots a_k}$ — запис натурального числа n у двійковій системі числення, то $T(n) = k + 1 + \sum_{i=1}^{k} a_i$, де $T(n)$ — кількість операцій множення у алгоритмі бінарного піднесення до степеня (Power або PowerRec).

Наслідок. $1+\lceil \log_2 n \rceil \le T(n) \le 1+2\lfloor \log_2 n \rfloor$

Алгоритм Евкліда знаходження НСД

1. **Рекурсивна форма:**

```
function gcdRecur(const a, b: Word): Word;
begin
    if b <> 0 then
        gcdRecur := gcdRecur(b, a mod b)
    else
        gcdRecur := a;
end;
```

Час роботи алгоритму Евкліда

Лема 1. *Якщо* $a > b \ge 1$ *і у результаті виклику функції* gcdRecur(*a,b*) *виконується* $k \ge 1$ *рекурсивних викликів, то* $a \ge F_{k+2}, b \ge F_{k+1}$.

Доведення. Індукція по *k*.

Наслідком леми 1 є

Теорема Ламе. *Якщо для довільного цілого* $k \ge 1$ *виконуються умови* $a > b \ge 1$ *і* $b < F_{k+1}$, *то у виклику функції* gcdRecur(*a,b*) *виконується менше ніж k рекурсивних викликів.*

Верхня границя теореми Ламе досягається у випадку $a = F_{k+1}$, $b = F_k$.

Оскільки за формулою Біне $F_k = \left(\alpha^k - \left(-\alpha^{-1}\right)^k\right) / \sqrt{5}$, де $\alpha = \left(1+\sqrt{5}\right)/2$ — золотий переріз, то кількість рекурсивних викликів у функції gcdRecur(*a,b*) рівна $O(\log b)$.

2. **Ітеративна форма:**

```
function gcdIter(a, b: Word): Word;
var
    t: Word;
begin
    while b <> 0 do
    begin
        t := b;
        b := a mod b;
        a := t;
    end;
    gcdIter := a;
end;
```

3. **Розширена версія алгоритму Евкліда:**

```
procedure ExtendedEuclid(const a,b: Word; var d, u, v: Integer);
var
    x, y: Integer;
begin
    if b = 0 then
    begin
        d := a;
        u := 1;
        v := 0;
```

```
    end
    else
    begin
        ExtendedEuclid(b, a mod b, d, x, y);
        u := y;
        v := x - (a div b)*y;
    end;
end;
```

Обчислення чисел Фібоначчі

1. Рекурсивний алгоритм:

```
function FibonacciRec(const N: Word): UInt64;
begin
    if N > 1 then
        FibonacciRec := FibonacciRec(N-1) + FibonacciRec(N-2)
    else
        FibonacciRec := N;
end;
```

Задача. Для кожного натурального k $(k < n)$ підрахувати число C_k — загальну кількість викликів функції FibonacciRec(k) у процесі обчислення FibonacciRec(n), знайти $\sum_{k=1}^{n-1} C_k$ та встановити обчислювальну складність рекурсивного алгоритму. Відповідь: $C_k = F_{n-k}, k = 1, \ldots, n-1,\ C_0 = C_2,\ \sum_{k=1}^{n-1} C_k = F_{n+1} - 1$, алгоритм має експоненційну складність.

2. Ітераційний алгоритм:

```
function FibonacciIter(const N: Word): UInt64;
var
    Prev, Curr, Temp: UInt64;
    i: Word;
begin
    Curr := 1; Prev := 1;
    for i := 3 to N do
    begin
        Temp := Prev + Curr;
        Prev := Curr;
        Curr := Temp;
    end;
    FibonacciIter := Curr;
end;
```

Задача. Дослідити складність ітеративного алгоритму.

3. Швидкий алгоритм обчислення

Нехай матриця F визначається наступним чином

$$F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Тоді справедливою є формула

$$F^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}.$$

Піднесення до степеня можна реалізувати за допомогою швидкого бінарного алгоритму.

Задача. Дослідити складність ітеративного алгоритму.

Множення довгих цілих чисел

Нехай потрібно обчислити добуток двох n-бітових цілих чисел X та Y. Звичайний алгоритм обчислення добутку пов'язаний з обчисленням n проміжних значень розмірності n і тому його час роботи рівний $\Theta\left(n^2\right)$. Один з варіантів методу декомпозиції полягає у розбитті чисел X та Y на два числа по $2^{n/2}$ бітів кожне

X:=	A	B	$X = A2^{n/2} + B$
Y:=	C	D	$Y = C2^{n/2} + D$

Добуток можна записати у вигляді

$$XY = AC2^n + (AD + BC)2^{n/2} + BD\,.$$

Для цього треба виконати 4 множення, 3 додавання і 2 зсуви. Тоді для загальної кількості операцій $T\left(n\right)$ можна записати

$$T(1) = 1\,,$$
$$T(n) = 4T(n/2) + cn\,.$$

Розв'язок останнього рекурентного рівняння є величиною $\Theta\left(n^2\right)$.

Задача. Довести останнє твердження.

Для покращення цієї оцінки потрібно так розбити задачу на підзадачі, щоб зменшити кількість підзадач. Цього можна досягти з використанням наступної формули:

$$XY = AC2^n + [(A-B)(D-C) + AC + BD]2^{n/2} + BD\,.$$

Тоді

$$T(1) = 1\,,$$
$$T(n) = 3T(n/2) + cn\,.$$

Розв'язком є $T\left(n\right) = O\left(n^{\log_2 3}\right) = O\left(n^{1,59}\right)$.

МЕТОДИ СОРТУВАННЯ МАСИВІВ

Потрібно впорядкувати числову послідовність $a_1, a_2, \ldots, a_n$ у *не спадному порядку.*

1. Сортування вичерпуванням (підрахунком)

Застосовується для сортування масивів, елементи яких приймають значення із фіксованої скінченної множини.

```
Program SortingAlgorithms;

uses
    Crt;
const
    MaxArrayDimension = 1000;
type
    TByteArray = array [1..MaxArrayDimension] of Byte;
procedure CountingSort(var A: TByteArray; n: Integer);
var
    RepArray: array [0..255] of Byte;
    i, j, k: Integer;
begin
    for i := 0 to 255 do
        RepArray[i] := 0;
    for i := 1 to n do
        Inc(RepArray[A[i]]);
    j := 1;
    for i := 0 to 255 do
        for k := 1 to RepArray[i] do
        begin
            A[j] := i;
            Inc(j);
        end;
end;
```

Задача. Встановити складність алгоритму сортування вичерпуванням ($O(n+m)$).

2. Сортування за допомогою парних порівнянь

Дерева рішень для алгоритмів сортування

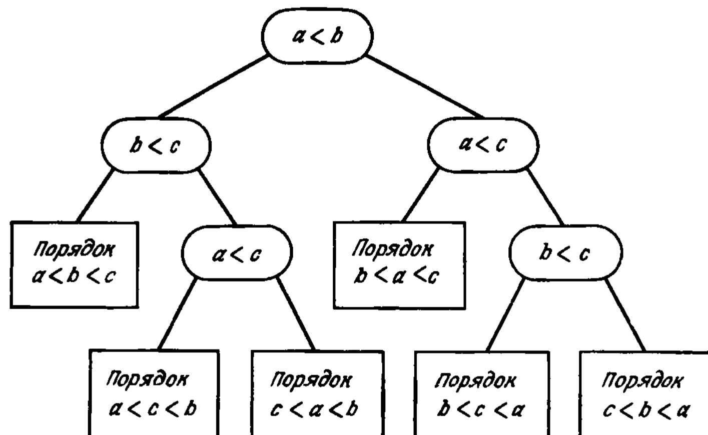


Рис 1. Дерево рішень для алгоритму впорядкування 3 чисел.

Теорема 1. *Будь-який алгоритм сортування з використанням порівнянь потребує застосування* $\Omega(n\log n)$ *операцій для деякої послідовності довжини n.*

Доведення. Доведення засноване на нерівності $n! > \left(\frac{n}{e}\right)^n$ та на наступній лемі:

Лема. *Бінарне дерево висоти h містить не більше* 2^h *листів.*

Теорема 2. *Будь-який алгоритм сортування з використанням порівнянь у середньому потребує застосування* $\Omega(n\log n)$ *операцій за умови, що на його вхід подається випадкова перестановка n-елементної послідовності.*

а) Сортування методом вставки

```
TDoubleArray = array [0..MaxArrayDimension] of Double;
procedure InsertSort(var A: TDoubleArray; n: Integer);
var
    i, j: Integer;
    t: Double;
begin
    for i := 2 to n do
    begin
        j := i - 1;
        a[0] := a[i] - 1;
        t := a[i];
        while t < a[j] do
        begin
            a[j+1] := a[j];
            j := j - 1;
        end;
        a[j+1] := t;
    end;
end;
```

Аналіз сортування методом вставки (середня складність)

б) Сортування злиттям

```
procedure MergeSort(var A: TDoubleArray; n: Integer);
var
    i, m, p: Integer;
    B, C: TDoubleArray;
begin
    if n > 1 then
    begin
        m := n div 2;
        p := n - m;
        for i := 1 to m do
            B[i] := A[i];
        for i := m+1 to n do
            C[i - m] := A[i];
        MergeSort(B, m);
        MergeSort(C, p);
        B[0] := C[1] - 1; C[0] := B[1] - 1;
        for i := n downto 1 do
        if B[m] > C[p] then
        begin
```

```
            A[i] := B[m];
            Dec(m);
        end
        else
        begin
            A[i] := C[p];
            Dec(p);
        end
    end;
end;
```

Аналіз сортування методом злиття ($T_{2n} = 2T_n + cn,\ T_n = \Theta(n\log n)$)

Швидкий алгоритм сортування

```
procedure QuickSort(var A: TDoubleArray; l, r: Integer);
var
    i, j: Integer;
    x, t: Double;
begin
    x := A[(l + r) div 2];
    i := l;
    j := r;
    while i <= j do
    begin
        while A[i] < x do
            i := i + 1;
        while A[j] > x do
            j := j - 1;
        if i <= j then
        begin
            t := A[i];
            A[i] := A[j];
            A[j] := t;
            i := i + 1;
            j := j - 1;
        end;
    end;
    if l < j then
        QuickSort(A, l, j);
    if i < r then
        QuickSort(A, i, r);
end;
```

Аналіз алгоритму швидкого сортування (найгірша та середня складність)

$$T(n) \leqslant cn + \frac{1}{n}\sum_{i=1}^{n}[T(i-1) + T(n-i)] \text{ для } n \geqslant 2.$$

$$T(n) \leqslant cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i).$$

$$T(n) \leqslant cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} ki \ln i.$$

$$\sum_{i=2}^{n-1} i \ln i \leqslant \int_{2}^{n} x \ln x\, dx \leqslant \frac{n^2 \ln n}{2} - \frac{n^2}{4}.$$

$$T(n) \leqslant cn + \frac{4b}{n} + kn \ln n - \frac{kn}{2}.$$

Порядкові статистики

Нехай на скінченній n-елементній множині задано лінійний порядок. Впорядкуємо елементи множини згідно до цього порядку. Будемо називати i-ою порядковою статистикою множини i-й елемент впорядкованої послідовності.

1. Пошук максимуму.

Твердження 1. *Для знаходження найбільшого елемента потрібно провести не менше ніж $n-1$ порівняння.*

2. Одночасний пошук максимуму та мінімуму.

Усі елементи масиву (крім перших двох) розбиваються на пари. Елементи пари спочатку порівнюються між собою, а потім більший порівнюється з поточним максимумом, менший — з мінімумом. Загальна кількість порівнянь не більша за $\left\lfloor \frac{3n}{2} \right\rfloor - 1$.

Твердження 2. *Для знаходження найбільшого та найменшого елементів серед n чисел потрібно провести не менше* $\left\lceil \frac{3n}{2} \right\rceil - 2$ *порівняння.*

3. Пошук порядкових статистик за лінійний очікуваний час.

Декомпозиційний алгоритм.

```
function Partition(var A: TDoubleArray; l, r: Integer): Integer;
var
    i, j: Integer;
    x, t: Double;
begin
    x := A[r];
    i := l - 1;
    for j := l to r - 1 do
        if A[j] <= x then
        begin
            Inc(i);
            t := A[i];
            A[i] := A[j];
            A[j] := t;
        end;
    i := i + 1;
    A[r] := A[i];
    A[i] := x;
    Partition := i;
end;

function RandomizedPartition(var A: TDoubleArray; l, r: Integer): Integer;
var
    k: Integer;
    t: Double;
begin
    k := l + Random(r - l + 1);
    t := A[k];
```

```
    A[k] := A[r];
    A[r] := t;
    RandomizedPartition := Partition(A, l, r);
end;

function Select(var A: TDoubleArray; l, r, k: Integer): Double;
var
    p, q: Integer;
begin
    if l = r then
        Select := A[l]
    else
        p := RandomizedPartition(A, l, r);
        q := p - l + 1;
        if q = k then
            Select := A[p]
        else
            if q > k then
                Select := Select(A, l, p - 1, k)
            else
                Select := Select(A, p + 1, r, k - q);
                // The last call never occurs in case p+1 > r
end;
```

Аналіз алгоритму Select.

Найгірший випадок: час роботи алгоритму рівний $\Omega\left(n^2\right)$ (навіть для пошуку мінімуму).

Нехай випадкова величина $T\left(n\right)$ — час роботи алгоритму Select на вході довжини n. Тоді має місце

Твердження 3. $M\left[T\left(n\right)\right]=\Theta\left(n\right)$.

Доведення. Визначимо індикаторну випадкову величину X_i, яка приймає значення 1, якщо в масиві $A[l...p]$ міститься рівно i елементів $\left(p-l+1=i\right)$, та значення 0 — в усіх інших випадках. Тоді

$$T\left(n\right)\le\sum_{i=1}^{n-1}X_i\max\left\{T\left(i\right),T\left(n-i-1\right)\right\}+O\left(n\right)$$

Оскільки $M\left[X_k\right]=1/n$, то

$$M\left[T\left(n\right)\right]\le\frac{1}{n}\sum_{i=0}^{n-1}\max\left\{T\left(i\right),T\left(n-i-1\right)\right\}+O\left(n\right)\le\frac{2}{n}\sum_{i=\left\lceil\frac{n-1}{2}\right\rceil}^{n-1}T\left(i\right)+O\left(n\right).$$

Припустимо, що $T\left(i\right)\le ci$, $\left(i=1,2,...,n-1\right)$ і нехай α — константа, яка відповідає члену $O(n)$. Тоді

$$T\left(n\right)\le\frac{2c}{n}\left(\frac{\left(n-1\right)n}{2}-\frac{\left(\left\lceil n/2\right\rceil-1\right)\left\lceil n/2\right\rceil}{2}\right)+\alpha n\le\frac{2c}{n}\left(\frac{\left(n-1\right)n}{2}-\frac{\left(n/2-1\right)n/2}{2}\right)+\alpha n\le$$

$$= c\left(n - 1 - n/4 + 1/2\right) + \alpha n = cn - \left(\frac{c}{4}\left(n-2\right) - \alpha n\right).$$

Таким чином, якщо $c > 4\alpha$ та $n > 2$, то $T\left(n\right) = O\left(n\right)$.

4. Алгоритм вибору з лінійним часом роботи у найгіршому випадку.

Опис кроків алгоритму LinearSelect:

1. Усі n елементів вхідного масиву розбиваються на $\lfloor n/5 \rfloor$ груп по 5 елементів у кожній та одну (можливо порожню) групу, яка містить $n \bmod 5$ елементів.
2. Кожна група сортується за допомогою методу вставки і потім знаходиться медіана кожної групи.
3. Шляхом рекурсивного застосування процедури LinearSelect визначається величина *x* — медіана множини із $\lceil n/5 \rceil$ медіан, знайдених на кроці 2.
4. За допомогою процедури Partition вхідний масив розбивається відносно елемента *x*.
5. Шукана статистика або співпадає з елементом, по якому проводилося розбиття, або шукається у верхній або нижній частинах масиву.

Твердження 4. *Алгоритм* LinearSelect *має лінійний час роботи.*

Доведення. Кількість елементів, які більші за *x* не менша за

$$3\left(\frac{1}{2}\lceil n/5 \rceil - 2\right) \geq \frac{3}{10}n - 6.$$

Така сама оцінка має місце для кількості елементів, які менші за *x*. Таким чином

$$T\left(n\right) \leq T\left(\lceil n/5 \rceil\right) + T\left(7n/10 + 6\right) + O\left(n\right).$$

Далі для доведення можна використати метод матіндукції.

АЛГОРИТМ БІНАРНОГО ПОШУК

```
program BinarySearch; // search in arranged numerical array

uses Crt;

const
    MaxArrayDimension =  1000;

type
    TByteArray = array [1..MaxArrayDimension] of Byte;
    TDoubleArray = array [0..MaxArrayDimension] of Double;

//Search in sorted array
function BinarySearch(const A: TDoubleArray; N: Integer; Value: Double;
var Found: Boolean): Integer;
var
    l, m, r: Integer;
begin
    l := 1;
    r := N;
    Found := false;
```

```
    while l <= r do
    begin
        m := (l+r) div 2;
        if A[m] > Value then
            r := m - 1
        else if A[m] < Value then
            l := m+1
        else
        begin
            Found := true;
            BinarySearch := m;
            exit;
        end;
    end;
    BinarySearch := l;
end;
```

ДИНАМІЧНЕ ПРОГРАМУВАННЯ

Динамічне програмування дає змогу знаходити розв'язок задачі, комбінуючи розв'язки допоміжних підзадач. На відміну від методу декомпозиції динамічне програмування застосовується у тому випадку, коли допоміжні підзадачі не є незалежними, тобто при розв'язуванні підзадач використовуються розв'язки попередніх підзадач. В алгоритмах динамічного програмування кожна допоміжна задача розв'язується один раз, після чого її розв'язок зберігається у таблиці.

Динамічне програмування як правило застосовується для розв'язування задач оптимізації, в яких кожному допустимому розв'язку відповідає деяке числове значення і потрібно знайти розв'язок із оптимальним значенням.

Процес розробки алгоритмів динамічного програмування можна розбити на наступні етапи:

1. Опис структури оптимального розв'язку.
2. Рекурсивне визначення значення, яке відповідає оптимальному розв'язку.
3. Обчислення значення, яке відповідає оптимальному розв'язку методом висхідного аналізу.
4. Конструювання оптимального розв'язку на основі інформації, отриманої на попередніх етапах.

Задача про ранець

Є N видів предметів, для кожного з яких відома його вага та ціна. У людини є ранець вантажопідйомністю M. Ваги предметів та M — натуральні числа. Потрібно завантажити ранець предметами таким чином, щоб їх сумарна ціна була якомога більшою. Задачу розв'язати за умови, що кількість предметів кожного виду є необмеженою.

Функція обчислення оптимального значення (ціни):

```
const
    MaxArraySize = 1000;
type
    TItem = record
        Weight: Word;
        Price: Double;
```

```
    end;
    TItemArray = array[1..MaxArraySize] of TItem;

// Non-effective exponential recursive realization
function KnapsackRecursive(M: Integer; const Items: TItemArray; N:
Integer): Double;
var
    i,j: Integer;
    t, Max: Double;
begin
    if M <= 0 then
        KnapsackRecursive := 0
    else
    begin
        Max := 0;
        for i := 1 to N do
         if Items[i].Weight <= M then
         begin
            t := KnapsackRecursive(M - Items[i].Weight, Items, N) +
Items[i].Price;
            if t > Max then
                Max := t;
         end;
         KnapsackRecursive := Max;
    end;
end;

// Bottom-up dynamic programming realization with O(MN) execution time needed
function Knapsack(M: Integer; const Items: TItemArray; N: Integer):Double;
// M - Max weight of knapsack, Items - array of thing, N - size  of thing array
var
    i,j: Integer;
    Max: Double;
    OptLoad: array[0..MaxArraySize] of Double;
begin
    for i := 0 to M do
    begin
        Max := 0;
        OptLoad[i] := 0;
        for j := 1 to N do
            if (i >= Items[j].Weight) and
               (OptLoad[i-Items[j].Weight]+Items[j].Price > Max) then
                Max := OptLoad[i-Items[j].Weight]+Items[j].Price;
        OptLoad[i] := Max;
    end;
    Knapsack := Max;
end;
```

Задача 1. *Модифікувати функцію Knapsack таким чином, щоб вона знаходила оптимальне завантаження ранцю (список усіх предметів, які кладуться у ранець при оптимальному заповненні).*

Задача 2. *Знайти таке завантаження ранця предметами, щоб їх сумарна вага була не меншою за задане число К, а їх сумарна ціна була найменшою.*

Твердження 1. *Час роботи алгоритму Knapsack рівний* $\Theta(MN)$.

Задача про обчислення добутку послідовності матриць

Нехай потрібно обчислити добуток $A_1 A_2 ..., A_n$, де A_i — матриця $p_{i-1} \times p_i$, $(i = 1, 2, ..., n)$. Будемо шукати такий порядок виконання множень матриць, при якому мінімізується кількість операцій множення елементів (при обчисленні добутку матриць розмірності $p \times q$ та $q \times r$ треба виконати pqr операцій множення).

Приклад. Нехай потрібно перемножити матриці A_1, A_2, A_3, розміри яких рівні $10 \times 100, 100 \times 5$ та 5×50. Тоді множення у порядку $(A_1 A_2) A_3$ потребує 7500 операцій множення, а множення у порядку $A_1 (A_2 A_3)$ — 75000 операцій.

Підрахунок кількості способів розстановки дужок. Нехай $P(n)$ — кількість різних способів розстановки дужок при множенні n матриць. Тоді

$$P(n) = \begin{cases} 1 & \text{при } n = 1, \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{при } n \geqslant 2. \end{cases}$$

Легко показати, що $P(n) = \Omega(2^n)$. Тому розв'язати задачу перебором для великих n неможливо.

1. **Структура оптимальної розстановки дужок.**

Нехай $A_{i...j} = A_i A_{i+1} ... A_j$. Для довільної оптимальної розстановки знайдеться такий індекс k, що $A_{i...j}$ обчислюється як добуток $A_{i...k}$ та $A_{k+1...j}$. Легко показати, що тоді розстановка в "префіксній" розстановці $A_{i...k}$ та "суфіксній" розстановці $A_{k+1...j}$ також є оптимальними.

Таким чином задачу оптимального множення матриць $A_i ... A_j$ можна звести до підзадач оптимального множення підланцюжків $A_i ... A_k$ та $A_{k+1} ... A_j$. Для знаходження оптимального розв'язку потрібно розглянути усі способи розбиття початкової задачі на підзадачі і вибрати з них найкращий.

2. **Рекурсивний розв'язок.**

Нехай $m[i, j]$ — мінімальна кількість множень, потрібна для обчислення матриці $A_{i...j}$. Тоді для знаходження повного розв'язку задачі потрібно обчислити $m[1, n]$. Визначимо величину $m[i, j]$ рекурсивно наступним чином:

$$m[i, j] = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \leqslant k < j} \{ m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \} & \text{при } i < j. \end{cases}$$

3. **Обчислення оптимальної кількості множень.**

Безпосереднє застосування рекурентного співвідношення, отриманого на 2-му етапі приводить до рекурсивної програми, час роботи якої експоненційний:

$$T(1) \geqslant 1,$$

$$T(n) \geqslant 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{при } n > 1.$$

$$T(n) \geqslant 2\sum_{i=1}^{n-1} T(i) + n.$$

Індукцією по n можна показати, що $T(n) \geq 2^{n-1}$:

$$T(n) \geqslant 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^{i} + n = 2\left(2^{n-1} - 1\right) + n = (2^n - 2) + n \geqslant 2^{n-1}.$$

Можна зауважити, що наша задача має $C_n^2 + n = \Theta(n^2)$ підзадач. Кожна з підзадач може неодноразово зустрічатися у різних гілках дерева рекурсії. Ця властивість перекриття допоміжних підзадач — друга властивість, яка вказує на можливість застосування методів динамічного програмування.

Замість використання безпосередніх рекурсивних викликів можна застосувати висхідний табличний підхід знаходження оптимального розв'язку. У процедурі використовуються допоміжна таблиця $m[i, j]$ та допоміжна таблиця $s[i, j]$, в яку записуються індекси k, при яких досягаються оптимальні значення $m[i, j]$.

```
uses
    Crt;
const
    MaxArraySize = 10;
type
    TIntVector = array[0..MaxArraySize] of Integer;
    TIntMatrix = array[1..MaxArraySize, 1..MaxArraySize] of Integer;

function MatrixChainOrder(const P: TIntVector; N: Integer; var s:
TIntMatrix): Int64;
const
    MaxInt64 = $7FFFFFFFFFFFFFFF;
var
    i,j, k, l: Integer;
    t: Int64;
    m: array[1..MaxArraySize, 1..MaxArraySize] of Int64;
begin
    for i := 1 to N do
        m[i,i] := 0;
    for l := 2 to N do
        for i := 1 to N-l+1 do
        begin
            j := i+l-1;
            m[i,j] := MaxInt64;
            for k := i to j-1 do
            begin
                t := m[i,k] + m[k+1,j] + P[i-1]*P[k]*P[j];
                if t < m[i,j] then
                begin
                    m[i,j] := t;
```

```
                    s[i,j] := k;
                end;
            end;
        end;
    MatrixChainOrder := m[1,N];
end;
```

Твердження 2. *Час роботи алгоритму MatrixChainOrder рівний* $O\left(N^3\right)$.

4. Конструювання оптимального розв'язку.

Для виведення оптимального розв'язку можна використати матрицю s.

```
procedure PrintOptimalSolution(const s: TIntMatrix; l, r: Integer);
begin
    if l = r then
        Write('A', l)
    else
    begin
        Write('(');
        PrintOptimalSolution(s, l, s[l,r]);
        PrintOptimalSolution(s, s[l,r]+1, r);
        Write(')');
    end;
end;
```

Задачі для самостійного розв'язування.

1. Задача про найдовшу спільну підпослідовність

Послідовність $Z = \left\langle z_1, z_2, ..., z_k \right\rangle$ називається підпослідовністю послідовності $\left\langle x_1, x_2, ..., x_m \right\rangle$, якщо існує строго зростаюча послідовність індексів $i_1, i_2, ..., i_k$, така що $z_j = x_{i_j}$, $j = \overline{1,k}$. Послідовність Z є спільною підпослідовністю послідовностей X та Y, якщо Z є спільною підпослідовністю як X, так і Y.

В задачі про найдовшу спільну підпослідовність дано 2 послідовності $\left\langle x_1, x_2, ..., x_m \right\rangle$ та $\left\langle y_1, y_2, ..., y_n \right\rangle$ і потрібно знайти їх спільну підпослідовність найбільшої довжини (розробити алгоритм самостійно та оцінити час його роботи).

2. Задача "про найкоротший шлях" у таблиці

В усі клітинки прямокутної таблиці записані невід'ємні числа. Під шляхом, який з'єднує клітинки A та B, будемо розуміти послідовність сусідніх клітинок таблиці, яка починається в A та завершується в B. Довжиною шляху назвемо суму чисел в усіх клітинках, через які він проходить. Написати програму знаходження шляху найменшої довжини із лівої верхньої клітинки у праву нижню.

ЖАДІБНІ АЛГОРИТМИ

Для багатьох задач оптимізації, структура задачі дає змогу розробити простіші алгоритми розв'язування, ніж алгоритми динамічного програмування. У жадібних алгоритмах завжди робиться вибір, який видається найкращим у даний момент часу, тобто проводиться локально оптимальний вибір у надії на те, що він приведе до оптимального розв'язку глобальної задачі.

Задача про вибір процесів

Припустимо, що множина $S=\{a_1,...,a_n\}$ складається з n процесів, кожний з яких характеризується півінтервалом $[s_i,f_i)$ (s_i — час початку процесу, f_i — час завершення, $i=\overline{1,n}$). Процесам потрібний певний ресурс, який одночасно може використовувати лише один процес. Процеси a_i та a_j називаються сумісними, якщо проміжки $[s_i,f_i)$ та $[s_j,f_j)$ не перетинаються. Задача про вибір процесів полягає у тому, щоб вибрати максимальну за кількістю елементів множину сумісних процесів.

Впорядкуємо процеси за часом зростання моментів їх завершення. Алгоритм знаходження розв'язку заснований на тому, що *знайдеться оптимальний розв'язок* задачі про вибір процесів, в який входить процес a_1.

```
uses
    Crt;

const
    MaxVectorSize = 255;

type
    TProcess = record
        s, f: Double;
    end;
    TProcessVector = array[1..MaxVectorSize] of TProcess;
    TSet = set of Byte;

procedure ProcessSelection(const Processes: TProcessVector; N: Integer;
var Selection: TSet);
// Array must be sorted by finish time field
var
    i, j: Integer;
begin
    Selection := [1];
    i := 1;
    for j := 2 to N do
        if Processes[j].s >= Processes[i].f then
        begin
            Include(Selection, j);
            i := j;
        end;
end;
```

Час роботи алгоритму рівний $\Theta(N)$.

Етапи процесу розробки жадібних алгоритмів:

1. Звести оптимізаційну задачу до вигляду, коли після зробленого вибору залишиться розв'язати лише одну підзадачу.
2. Довести, що завжди існує оптимальний розв'язок задачі, який можна знайти шляхом здійснення жадібного вибору.
3. Довести, що після жадібного вибору залишається підзадача, яка має таку властивість: об'єднання оптимального розв'язку підзадачі зі зробленим жадібним вибором приводить до оптимального розв'язку початкової задачі.

Задача 1. Розробити жадібний алгоритм розв'язування неперервної задачі про ранець, час роботи якого рівний $O(n\log n)$ (запас кожного товару відомий, можна брати частину кожного товару).

Приклад задачі, для якої не підходить жадібна стратегія — булева задача про ранець ((10; 60), (20; 100), (30; 120), $M = 50$).

Задача 2. Про мінімальну кількість затримок на дозаправку.

Задача 3. Нехай задані два n-вимірні вектори **a** та **b** з натуральними координатами. Розробити та обґрунтувати жадібний алгоритм знаходження найбільшого значення величини $\prod_{i=1}^{n} a_i^{b_{j_i}}$, де $(j_1,..., j_n)$ — деяка перестановка чисел $1,...,n$.

Коди Хаффмана

Нехай потрібно закодувати повідомлення, яке є послідовністю символів деякого алфавіту. У жадібному алгоритмі Хаффмана використовується таблиця, у якій містяться частоти входження символів у повідомлення.

Нехай кодується повідомлення, яке складається із 100 символів. Символи зустрічаються із частотами, наведеними у таблиці

Символ	а	б	в	г	д
Частота	55	15	12	10	8
Код фіксованої довжини	000	001	010	011	100
Код змінної довжини	0	100	101	110	111

При кодуванні з використанням ЕОМ зазвичай використовується бінарний код. Для коду фіксованої довжини знадобиться 3 біта для представлення символів і загальна довжина закодованого повідомлення становить 300 бітів. З використанням коду змінної довжини, наведеного у таблиці, можна отримати повідомлення довжиною 55 + 3(15 + 12 + 10 + 8) = 190 бітів. При використанні кодів змінної довжини зручно використовувати *префіксні коди*, для яких ніяке кодове позначення символів не співпадає з початком будь-якого іншого кодового позначення. Префіксні коди спрощують процес декодування. Можна показати, що оптимальний можливий стиск даних, який можна отримати з використанням кодів, завжди досяжний з використанням префіксних кодів. Надалі будемо розглядати лише префіксні коди.

Для реалізації процесу декодування використовуються кодові бінарні дерева, листами яких є символи, що кодуються.

Приклад кодового дерева.

Якщо C — алфавіт вхідного повідомлення, то кодове дерево має $|C|$ кінцевих вершин. Довжина закодованого повідомлення рівна

$$B(T) = \sum_{c \in C} f(c)\, d_T(c)$$

де $f(c)$ — частота входження символу c, $d_T(c)$ — довжина коду символу c.

Код називається оптимальним, якщо для нього отримується кодове повідомлення найменшої довжини.

Класичний алгоритм побудови дерева Хаффмана

1. Символи вхідного алфавіту помістити у список вільних вузлів. Кожний вузол має вагу, яка рівна частоті відповідного символу.
2. Видалити із списку вільних вузлів два вузли з найменшою вагою. Створити для них вузол-предок, вага якого рівна сумі ваг нащадків. Приписати дугам, що виходить із нового вузла 0 та 1 відповідно. Створений вузол додати у список вільних вузлів.
3. Якщо що є хоча би два вільні вузли, то повернутися на крок 2.

Класичний алгоритм використовує двійковий вихідний алфавіт. Для кодування у випадку m-символьного вихідного алфавіту Z_m потрібно модифікувати лише другий крок алгоритму. Операцій, які виконувалися у класичному алгоритмі для двох вузлів, повинні тепер виконуватися для відповідних m вузлів.

Коди Хаффмана є *префіксними*, тобто ніяке кодове позначення не співпадає з початком будь-якого іншого кодового позначення. Тому вони однозначно та легко декодуються. *Час роботи* алгоритму Хаффмана становить $O(n \log n)$, де n — кількість символів вхідного повідомлення. Недолік класичного методу Хаффмана — *потреба двох проходів* по повідомленню у процесі кодування.

Обґрунтування методу Хаффмана

Лема 1. *Нехай x, y — символи вхідного алфавіту, які мають найменшу частоту. Тоді для алфавіту C існує оптимальний префіксний код, у якому кодові позначення символів x, y мають однакову довжину та відрізняються лише у останньому біті.*

Із леми 1 випливає, що процес побудови оптимального кодового дерева можна починати із жадібного вибору.

Лема 2. *Нехай x, y — символи вхідного алфавіту, які мають найменшу частоту у алфавіті C і нехай алфавіт* C_1 *отримується із C шляхом вилучення символів x, y та додаванням символу z із частотою* $f(z) = f(x) + f(y)$. *Тоді якщо дерево T отримується із дерева оптимального кодового дерева* T_1 *алфавіту* C_1 *шляхом заміни листа z внутрішнім листом із дочірніми вузлами x та y, то T — оптимальне кодове дерево для алфавіту C.*

Із леми 2 випливає, що задача побудови оптимального кодового дерева має властивість оптимальної підструктури.

Теорема. *Дерево Хаффмана є оптимальним префіксним кодовим деревом.*

ДЕРЕВА. АЛГОРИТМИ З ВИКОРИСТАННЯМ ДЕРЕВ

Дерево — зв'язний граф без циклів. Будь яке дерево з n вершинами містить $n - 1$ ребро. *Орієнтоване дерево* — ациклічний граф, у якому тільки у одну вершину не входять дуги, а в усі інші вершини входить по одній дузі. Вершина, у яку не входять дуги, називається *коренем*, а вершини, з яких не виходять дуги — кінцевими вершинами (*листами*). *Кореневе дерево T* — дерево, у якому виділена одна вершина — корінь, а усі інші вузли входять до одного з піддерев дерева *T*. *Глибиною* або *рівнем вузла* (вершини) кореневого дерева називається довжина шляху від кореня до вузла. Максимальний рівень вузлів дерева називається *висотою дерева*. Впорядковане дерево, це дерево у якому для кожного вузла заданий порядок.

Бінарні дерева

Двійкове (бінарне) дерево — це:

1) неорієнтоване дерево, у якому степені вершин не перевищують 3;
2) орієнтоване дерево, у якому вихідні степені вузлів не перевищують 2.

Лема 1. *Висота $h(T_n)$ бінарного дерева, яке містить n вершин, задовольняє нерівності*

$$h(T_n) \geq \lfloor \log_2 n \rfloor,$$

$$h(T_n) \geq \log_2(n+1) - 1.$$

Бінарні дерева пошуку

Двійкове дерево пошуку (англ. binary search tree) — двійкове дерево, в якому кожній вершині x поставлено у відповідність певне *ключове* значення $x.key$. При цьому такі значення повинні задовольняти умові впорядкованості:

Якщо вершина y знаходиться в лівому піддереві вершини x, то $y.key \leq x.key$. Якщо вершина y знаходиться у правому піддереві x, то $y.key \geq x.key$.

Таке структурування дозволяє надрукувати усі значення у зростаючому порядку за допомогою простого алгоритму *центрованого обходу дерева*. Бінарні дерева пошуку набагато ефективніші в операціях пошуку, аніж лінійні структури, в яких витрати часу на пошук пропорційні $O(n)$, де n — розмір масиву даних, тоді як в повному бінарному дереві цей час пропорційний в середньому $O(h)$, де h — висота дерева (для збалансованих дерев h рівна $O(\log n)$).

```
program BST;
uses
    Crt;
type
    TKey = Double;
    TValue = String;
    PNode = ^TNode;
    TNode = record
        Key: TKey;
        Value: TValue;
        Left, Right: PNode;
    end;

procedure PrintTree(const Root: TNode);
    procedure PrintInPos(const NodePtr: PNode; const h, l, r: Integer);
    var
        MeanPos: Byte;
```

```
    begin
        MeanPos := (r + l)  div 2;
        GotoXY(MeanPos - Length(NodePtr^.Value) div 2, h);
        Write(NodePtr^.Value);
        if NodePtr^.Left <> nil then
            PrintInPos(NodePtr^.Left, h+2, l, MeanPos);
        if NodePtr^.Right <> nil then
            PrintInPos(NodePtr^.Right, h+2, MeanPos + 1, r);
    end;
begin
    //ClrScr;
    PrintInPos(@Root, 1, 1, 80);
end;
procedure DisposeTree(var RootPtr: PNode);
begin
    if RootPtr <> nil then
    begin
        with RootPtr^ do
        begin
            DisposeTree(Left);
            DisposeTree(Right);
        end;
        Dispose(RootPtr);
    end;
end;
function Search(RootPtr: PNode; K: TKey): PNode;
var
    p: PNode;
begin
    if (RootPtr = nil) or (RootPtr^.Key = K) then
        Search := RootPtr
    else
        if RootPtr^.Key < K then
            Search := Search(RootPtr^.Right, K)
        else
            Search := Search(RootPtr^.Left, K)
end;
procedure Insert(var RootPtr: PNode; K: TKey; V: TValue);
var
    p, q: PNode;
begin
    p := RootPtr;
    q := nil;
    while p <> nil do
    begin
        q := p;
        if p^.Key < K then
            p := p^.Right
        else
            p := p^.Left;
    end;
    new(p); p^.Left := nil; p^.Right := nil; //p^.Parent := q;
    p^.Key := K; p^.Value := V;
    if q <> nil then
        if q^.Key < K then
            q^.Right := p
```

```
        else
            q^.Left := p
    else
        RootPtr := p; // Insertion in empty tree
end;
function Min(RootPtr: PNode): PNode;
begin
    if (RootPtr = nil)  or  (RootPtr^.Left = nil) then
        Min := RootPtr
    else
        Min := Min(RootPtr^.Left);
end;
```

Середня глибина вузлів випадкового бінарного дерева пошуку

Теорема 1. *Середня глибина вершини випадкового бінарного дерева пошуку з n вузлами рівна* $O(\log n)$.

Доведення. Нехай $P(T)$ — загальна довжина шляхів бінарного дерева (сума глибин усіх вузлів). Тоді $M\left[D(x,T)\right]=\frac{1}{n}M\left[P(T)\right]$. Покажемо, що $M\left[P(T)\right]=O(n\log n)$.

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

Нехай $P(N)$ — середня довжина шляхів випадкового бінарного дерева із n вузлами. Тоді

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}\left(P(i) + P(n-i-1) + n - 1\right).$$

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1}P(k) + \Theta(n).$$

Ми отримали таке саме рекурентне співвідношення, як і при аналізі алгоритму швидкого сортування. Тому $M\left[P(T)\right]=O(n\log n)$.

Теорема 2. *Середня висота випадкового бінарного дерева пошуку з n вузлами рівна* $O(\log n)$.

```
procedure Delete(var P: PNode; K: TKey);
var
    Q: PNode;
    procedure Del(var R: PNode);
    begin
        if R^.Right <> nil then
            Del(R^.Right)
        else
        begin
            P^.Key := R^.Key; P^.Value := R^.Value;
            Q := R;
            R := R^.Left;
        end;
    end;
begin
    if P = nil then
        Exit;
    if K < P^.Key then
```

```
        Delete(P^.Left, K)
    else if K > P^.Key then
        Delete(P^.Right, K)
    else
    begin
        Q := P; // Q is variable for disposing
        if P^.Right = nil then
            P := Q^.Left
        else if P^.Left = nil then
            P := Q^.Right
        else
            del(Q^.Left);
        Dispose(Q);
    end;
end;

var
    RootPtr, NodePtr: PNode;
Begin
    Insert(RootPtr, 10, '10');  Insert(RootPtr, 20, '20');
    Insert(RootPtr, 4, '4');    Insert(RootPtr, 5, '5');
    Insert(RootPtr, 15, '15');  Insert(RootPtr, 25, '25');
    Insert(RootPtr, 0, '0');    Insert(RootPtr, 8, '8');
    Insert(RootPtr, 7, '7');    Insert(RootPtr, 5, '6');
    Insert(RootPtr, -1, '-1');  Insert(RootPtr, 1, '1');
    Insert(RootPtr, 3, '3');    Insert(RootPtr, 2, '2');
    ClrScr; PrintTree(RootPtr^, 1);
    Delete(RootPtr, 10);
    PrintTree(RootPtr^, 15);
    repeat until KeyPressed;
    DisposeTree(RootPtr);
End.
```

Кількість різних бінарних дерев

Нехай b_n — кількість різних бінарних дерев із n вузлами. Тоді $b_0 = 1$ і

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

Розв’язком попереднього рекурентного рівняння є числа Каталана. Тому $b_n = \frac{1}{n+1} C_{2n}^n$. З останньої формули з використанням формули Стірлінга можна отримати, що

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}} \left(1 + O\left(1/n\right)\right).$$

Збалансовані дерева

Бінарне дерево називається ідеально *збалансованим*, якщо для кожного його вузла кількість вузлів у лівому і правому піддеревах цього вузла відрізняються не більше ніж на 1.

Лема 2. *Висота $h(T_n)$ ідеально збалансованого бінарного дерева, яке містить n вершин, задовольняє нерівність*

$$h(T_n) \le \log_2 n.$$

Нехай $C(L'_n)$ — середня кількість операцій порівняння (середня довжина шляху + 1), потрібних для знаходження елемента у ідеально збалансованому дереві з n вершинами. Тоді $C(L'_n) \cong \log_2 n - 1$.

Якщо при побудові бінарного дерева пошуку використовувати процедуру Insert, то у випадку надходження даних у порядку зростання (спадання) ключів дерево вироджується у лінійний список. У цьому випадку для пошуку потрібно у середньому $n/2$ порівнянь і алгоритм пошуку є малоефективним. Нехай $M(L_n)$ — середня довжина шляху пошуку по усім можливим $n!$ бінарним деревам, які отримуються у результаті усіх $n!$ перестановок вхідних ключів (для кореня довжина шляху дорівнює 0). Тоді

$$M(L_n) = \frac{2(n+1)}{n} H_n - 4,$$

$$M(L_n) \cong 2(\ln n + \gamma) - 4,$$

де $\gamma = 0{,}577...$ — стала Ейлера. Тоді $C_n = M(L_n) + 1$ і

$$\lim_{n\to\infty} \frac{C(L_n)}{C(L'_n)} = \lim_{n\to\infty} \frac{2\ln n}{\log_2 n} = 2\ln 2 \approx 1{,}386.$$

Звідси випливає, що пошук з використанням ідеально збалансованих дерев дає «середній» виграш не більший за 39% від пошуку з використанням дерева, побудованого з використанням алгоритму Insert. Тому у випадку, коли відношення r між частотою звертання до елементів (пошуку) та частотою вставки у дерево є невеликим, використання алгоритму побудови випадкового дерева (Insert) є достатньо ефективним.

АВЛ-дерева

Дерево називається АВЛ-збалансованим, якщо для кожного вузла висота його піддерев відрізняється не більше ніж на 1. При реалізації АВЛ-дерев для кожного вузла зручно зберігати висоту даного вузла (тобто висоту дерева з коренем у цьому вузлі) або показник збалансованості, який обчислюється як різниця висот правого та лівого піддерев.

Найбільшу висоту серед АВЛ-дерев мають дерева Фібоначчі, які визначаються наступним чином:

1. Порожнє дерево та дерево, яке складається з одного кореня є деревами Фібоначчі.
2. Якщо T_1, T_2 — дерева Фібоначчі висоти h – 1 та h – 2 відповідно, то $<T_1, x, T_2>$ — дерево Фібоначчі висоти h.

Для кількості вузлів дерева Фібоначчі має місце рівність $n_h = n_{h-1} + n_{h-2} + 1$.

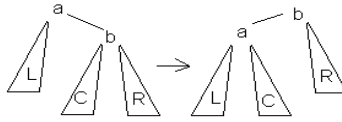
У АВЛ-дереві висоти h є не менше ніж F_{h+2} вузлів, де F_h — число Фібоначчі з індексом h.

Теорема. *Якщо* $h(T_n)$ *— висота збалансованого АВЛ-дерева, яке містить n вузлів, то*

$$\log_2(n+1) - 1 \le h(T_n) < 1{,}4405 \cdot \log_2(n+2).$$

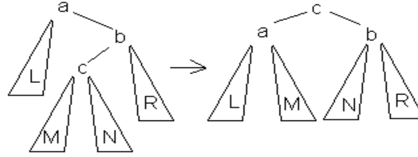
Балансування дерева. При вставці чи видаленні вершини АВЛ-дерево може стати розбалансованим, тобто висота лівого та правого піддерев деякого вузла можуть відрізнятися на 2. Для АВЛ-дерева балансуванням вершини називається операція, яка змінює його зв'язки таким чином, що різниця висот стає не більшою за 1. Балансування відбувається шляхом обертання піддерева даного вузла.

Малий лівий оберт



Застосовується тоді, коли $h(b)-h(L)=2$ та $h(C)\le h(R)$.

Великий лівий оберт



Застосовується тоді, коли $h(b)-h(L)=2$ та $h(C)>h(R)$. Аналогічно зображуються праві оберти. Можна показати, що велике обертання є комбінацією правого і лівого малих обертів.
При вставці у АВЛ-дерево достатньо провести одну операцію обертання.
Загальна схема процесу включення вузла:

1. Рухатися по шляху пошуку до тих пір, поки не виявиться, що ключа немає у дереві.
2. Вставити новий вузол та визначити новий показник збалансованості.

Виконати зворотний рух по шляху пошуку та перевірити показник збалансованості.
При реалізації АВЛ-дерев для кожної вершини потрібно додатково зберігати показник збалансованості вершини.

Література

1. Васильків Н. М., Васильків Л. О. Опорний конспект лекцій з дисципліни «Основи алгоритмізації». – Тернопіль: Економічна думка, 2005.
2. Столяр С.Е. Введение в алгоритмику. – СПб.: Издательство ЦПО "Информатизация образования", 2002.
3. Ноден П., Китте К. Алгебраическая алгоритмика, с упражнениями и решениями. – М.: Мир, 1999.
4. Вирт Н. Алгоритмы и структуры данных. – М.: ДМК, 2010.
5. Кнут Д. Искусство программирования. Т.1-3. – М.: Мир, 1999.
6. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: "Вильямс", 2005.
7. Ахо А., Хопкрофт Дж., Ульман Дж. – Структуры данных и алгоритмы – М.: "Вильямс", 2000.