Практичне заняття № 1

Наукове пізнання навколишнього світу людина здійснює за допомогою його моделювання. Кожна наукова дисципліна широко використовує різні моделі. Графічні схеми, математичні рівняння, діаграми причинно-наслідкових зв'язків, таблиці і навіть вербальні конструкції є різновидами моделювання.

Модель - це формалізоване, спрощене представлення реального об'єкту

Будь-яка наукова модель має цільовий характер, тобто вона будується для конкретних цілей і для вирішення певних задач. Вона достатньо точно відображає ті сторони модельованого явища, які мають ключове значення для вирішення поставленої задачі. Укладена в моделі помилка рано чи пізно заважає вирішувати інші задачі, пов'язані з модельованим об'єктом. Вирішити нові задачі можна або шляхом уточнення, узагальнення первинної моделі, або шляхом побудови нової моделі.

Алгоритм – фундаментальне поняття математики, йому не можна дати точне математичне визначення із залученням інших фундаментальних понять. Відсутність такого визначення не заважає інтуїтивному розумінню його змісту. Але в літературних джерелах наводяться різні варіанти розуміння сутності алгоритму. В загальному випадку під терміном "алгоритм" розуміють шлях (метод, спосіб) розв'язання задачі. Таке тлумачення не дозволяє досліджувати особливості роботи, конструювати ефективні алгоритми. Проблема розв'язується застосуванням моделей алгоритму.

В залежності від складності задачі, мети дослідження, практичного застосування моделі поділяються на декілька груп, які показані на *рис.1*.

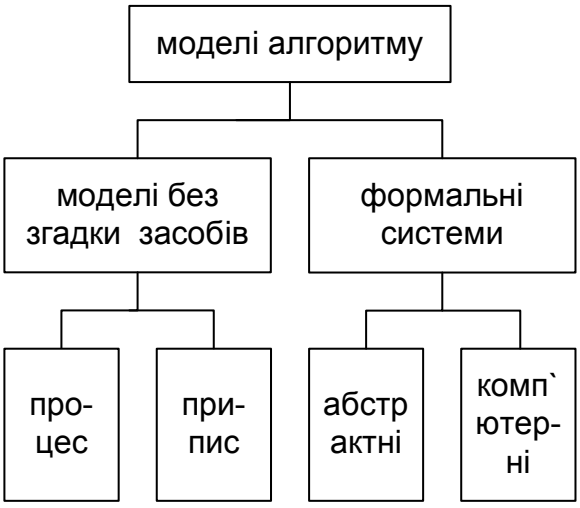


Рис. 1

Виконанню обчислювальних операцій без вказівки на засоби виконання відповідають дві моделі: "алгоритм – процес" і "алгоритм – припис"

Першу модель "**алгоритм – процес**" виконання чотирьох арифметичних операцій детально у словесній формі описав аль-Хорезмі (IX стор). Пізніше в працях Хр. Рудольфа (XVI стор.) і Г.В.Лейбніця (XVII стор.) модель аль-Хорезмі набула наступного тлумачення – *"Алгоритм означає будь-який регулярний обчислювальний процес, який за кінцеву кількість кроків розв'язує задачі визначеного класу*". Це тлумачення близьке до наведеного у сучасній фундаментальній монографії: "*Кажучи неформально, алгоритм – це будь-яка коректно сформульована обчислювальна процедура, на вхід якої подається деяка величина або набір величин, і результатом виконання якої є вихідна величина або набір значень*". Більш детальне тлумачення з описом властивостей алгоритму дано А Марковим:

“а) Алгоритм – це процес послідовної побудови величин, який проходить в дискретному часі таким чином, що в початковий момент задається початкова скінчена система величин, а в кожний наступний момент система величин отримується за певним законом (програмою) із системи величин, які були в попередній момент часу (дискретність алгоритму).
б) Система величин, які утримуються в якийсь (не початковий) момент часу, однозначно визначається системою величин, отриманих в попередні моменти часу (детермінованість алгоритму).
в) Закон отримання наступної системи величин із попередньої повинен бути простим і локальним (елементарність кроків алгоритму).
г) Якщо спосіб отримання наступної величини із якої-небудь заданої величини не дає результату, то повинно бути вказано, що потрібно рахувати результатом алгоритму (спрямованість алгоритму).
д) Початкова система величин може вибиратися із деякої потенційно нескінченної множини (масовість алгоритму)”.

Це розширене тлумачення можна знайти в ряді сучасних джерел. Порівнюючи ці три тлумачення, не можна сказати, яке з них краще або вірніше, бо це лише неформальний опис моделі алгоритмічного процесу. Для такого опису не має математичних підстав порівняння. Розглянуті тлумачення відрізняються один від одного. Наприклад, процес вимірюється часом виконання, припис вимірюється кількістю інструкцій програми або блок-схем програми, але ці відміни незначні.

Друга модель “**алгоритм – припис**” реалізується у вигляді блок-схем програм і програм на мові високого рівня. Прикладом тлумачення алгоритму на основі цієї моделі є: “*Алгоритм – це послідовність інструкцій для виконання деякого завдання*”.

Наступні варіанти тлумачення зумовлені використанням конструктивно заданих математичних моделей алгоритму – формальних алгоритмічних систем. Абстрактні формальні системи використовуються для дослідження теоретичних проблем обчислень, наприклад, проблем розв’язності. На основі абстрактних систем була створена теорія складності. До формальних алгоритмічних систем відносяться два класи математично строго визначених моделей, які містять всі основні засоби для здійснення обчислювального процесу. До першого класу відносяться абстрактні моделі, які у своєму визначенні не описують апаратні засоби, хоча їх наявність припускається. Також припускається, що операції алгоритмічного процесу здійснюються людиною без будь-яких інтелектуальних зусиль. Прикладами моделей цього класу є машина Тюрінга, нормальні алгоритми Маркова та декілька інших. Тлумаченням алгоритму, що випливає з аналізу цих моделей є: “*Алгоритм – точний припис, який задає обчислювальний процес (що називається в цьому випадку алгоритмічним), що починається з довільного початкового даного (з деякої сукупності можливих для даного алгоритму початкових даних) і спрямований на отримання результату, який повністю визначається цим початковим даним*”. Тут точний припис і алгоритмічний процес об’єднані в одному тлумаченні. Абстрактні формальні системи використовуються для дослідження теоретичних проблем обчислень, наприклад, проблем розв’язності. На основі абстрактних систем була створена теорія складності.

Другий клас формальних систем складається з моделей комп’ютерних алгоритмів. У цих моделях апаратні засоби задекларовані безпосередньо у її визначенні. Прикладом моделі другого класу є SH-модель алгоритму (SH – Software/Hardware). Теорія складності комп’ютерних алгоритмів суттєво відрізняється від абстрактної теорії. Тут крім технічних характеристик складності (часової, апаратної та ємнісної) використовуються додатково інформаційні (програмна та структурна). Модель комп’ютерного алгоритму дозволила формально визначити властивість “елементарність”, сформулювати властивість “ієрархічність”, створити модель універсального обчислювача. Відображення змісту моделі комп’ютерного алгоритму знайшло в неформальному тлумаченні.: *“Алгоритм - це*

фіксована для розв'язання деякого класу задач конфігурація апаратно-програмних засобів перетворення, передавання і зберігання даних, який задає обчислювальний процес (що називається в цьому випадку алгоритмічним), який починається з будь-яких початкових даних (з деякої потенційно нескінченної сукупності можливих для даного алгоритму початкових даних) і скерований на отримання результату, повністю визначеного цими початковими даними".

Таким чином кожне наведене тлумачення поняття "алгоритм" адекватне обраній для дослідження конкретної моделі обчислень.

Практичне заняття № 2

Те, що зараз ми розуміємо під словом алгоритм, використовувалося в глибокій давнині, наприклад, теорема про залишки в Китаї (Китайська теорема), арифметичні операції в Індії. Але праці Евкліда і аль-Хорезмі для теорії складності алгоритмів мають особливе значення.

Мухамед ібн Муса з Хорезму, за арабським ім'ям – аль-Хорезмі (походженням з середньоазіатського міста Хорезм), видатний багдадський вчений, що працював у IX столітті н.е. У своїй книжці – трактаті "Про індійський рахунок" аль-Хорезмі описав десяткову систему числення і арифметичні операції " множення і ділення, сумування, віднімання та інші". Сьогодні збереглися лише переклади трактату. Перші з них відносяться до початку XII століття.

Далі ми наводимо цитати з "Книги про індійський рахунок", переклад якого був виконаний Ю.Копелевич з середньовічного латинського тексту, що зберігається у Кембриджському університеті.

Кожний розділ трактату, а іноді навіть абзац, починався словами "Сказав Альгорізмі…". Це словосполучення використовували у своїх лекціях і професори середньовічних університетів. Поступово ім'я аль-Хорезмі набуло звучання "алгоризм", "алгоритм" і навіть перетворилися у назву нової арифметики. Пізніше термін "алгоритм" почав означати регулярний арифметичний процес (Хр. Рудольф, 1525р.). І тільки наприкінці XVII ст. в роботах Лейбніца цей термін набув змістовності, яка не заперечує сучасному тлумаченню: "Алгоритм - це будь-який регулярний обчислювальний процес, що дозволяє за кінцеву кількість кроків розв'язувати задачі визначеного класу". Зауважимо, що за довгу еволюцію слова "алгоритм" було втрачено джерело його виникнення. І тільки у 1849 році сходознавець Ж. Рейно повернув нам ім'я аль-Хорезмі .

Зауважимо, що й слово "алгебра" бере свій початок з математичного трактату аль-Хорезмі "Книга відновлення і протиставлення". Мухамеду ібн Мусі ще не були відомі від'ємні числа, тому в процесі обчислень він користувався операцією перенесення від'ємника з одної частини рівняння в іншу, де той стає доданком. Цю операцію Мухамеда ібн Муса називав "відновленням". Слову "протиставлення" відповідає зміст збирання невідомих на одну сторону рівняння. Арабською "відновлення" – аль-джебр. Звідси походить слово "алгебра". Слова "алгоритм" і "алгебра" на перших кроках розвитку математики було щільно пов'язані між собою і за змістом і за походженням. Вони виникли з одного джерела і разом пройшли багатовіковий шлях еволюції, зайняли провідні місця у сучасній науковій термінології.

Здавна найбільшу увагу приділяли дослідженням алгоритму з метою мінімізації обсягу досліджень – часовій складності розв'язання задач. Але зміст складності алгоритму не обмежується однією характеристикою. В ряді випадків не менше значення має складність логіки побудови алгоритму, різноманітність його операцій, зв'язаність їх між собою. Ця характеристика алгоритму називається програмною складністю. В теорії алгоритмів, крім часової та програмної складності, досліджуються також інші характеристики складності, наприклад, ємкістна, але найчастіше розглядають дві з них - часову і програмну. Якщо у кінцевому результаті часова складність визначає час розв'язання задачі, то програмна складність характеризує ступінь інтелектуальних зусиль, що потрібні для синтезу алгоритму. Вона впливає на витрати часу проектування алгоритму.

Вперше значення зменшення програмної складності продемонстрував аль-Хорезмі у своєму трактаті "Про індійський рахунок". У часи аль-Хорезмі для розрахунків користувалась непозиційною римською системою числення. Її вузловими числами є I, V, X, L, C, D, M, всі решта чисел утворюються сумуванням і відніманням вузлових. аль-Хорезмі, мабуть, першим звернув увагу на складність римської системи числення у порівнянні з позиційною десятковою з точки зору простоти операцій, їх послідовного виконання та засвоєння. Він писав: "…ми вирішили розтлумачити про індійський рахунок за допомогою .IX. літер, якими вони виражали будь-яке своє число для *легкості і стислості*, *полегшуючи* справу тому, хто вивчає арифметику, тобто число найбільше і найменше, і все, що є в ньому від множення і ділення, сумування, віднімання та інше.". Виокремленні слова – *легкість, стислість*, *полегшення* свідчать перш за все про те, що

мова йде про програмну складність алгоритмів арифметичних операцій з використанням двох систем числення. Мабуть, ці слова аль-Хорезмі про складність алгоритмів при їх порівнянні були першими в історії арифметики.

Алгоритми реалізації арифметичних операцій, описані аль-Хорезмі у словесній формі, були першими у позиційній десятковій системі числення. Цікаво спостерігати, як точно і послідовно описує він алгоритм сумування, користуючись арабською системою числення і кільцем (нулем). Наведемо повністю цей алгоритм.

"Сказав Алгорізмі: Якщо ти хочеш додати число до числа або відняти число від числа, постав обидва числа в два ряди, тобто одне над другим, і нехай буде розряд одиниць під розрядом одиниць і розряд десятків під розрядом десятків. Якщо захочеш скласти обидва числа, тобто додати одне до другого, то додай кожний розряд до розряду того ж роду, який над ним, тобто одиниці до одиниць, десятки до десятків. Якщо в якому-небудь із розрядів, тобто в розряді одиниць або десятків, або якому-небудь іншому набереться десять, став замість них одиницю і висувай її в верхній ряд, тобто, якщо ти маєш в першому розряді, який є розряд одиниць, десять, зроби з них одиницю і підніми її в розряд десятків, і там вона буде означати десять. Якщо від числа залишилось що-небудь, що нижче десяти, аби якщо саме число нижче десяти, залиши його в тому ж розряді. А якщо нічого не залишиться, постав кружок, щоби розряд не був пустим; але нехай буде в ньому кружок, який займе його, аби не сталося так, що якщо він буде пустим, розряди зменшаться і другий буде прийнятий за перший, і ти обманешся в своєму числі. Те ж саме ти зробиш у всіх розрядах. Подібним же чином, якщо збереться у другому розряді .X., зробиш з них одиницю і піднімеш її в третій розряд, і там вона буде означати сто, а що лишається нижче .X., залишиться тут. Якщо ж нічого в інших не залишається , ставиш тут кружок, як вище. Так ти зробиш в інших розрядах, якщо буде більше".

Можна бачити, що в цьому опису є всі параметри алгоритму. Це один з перших відомих у світі вербальних арифметичних алгоритмів.

Розглянемо логіку побудови арифметичних процедур з використанням римської та арабської систем числення з метою порівняння їх за програмною складністю. Розглянемо приклад операції сумування. Будемо користуватися алгоритмом на основі табличного методу. У арабській системі з порозрядними операціями розмір таблиці 10*10. Визначення суми чергових розрядів двох чисел, наприклад, 2 і 3 за таблицею дорівнює 5, або 7 і 9 дорівнює 16. Ці таблиці ми пам'ятаємо з дитинства.

Інша ситуація є з римською системою числення. Крім таблиці (I,II,…X)*(I,II,…X), що є еквівалентом таблиці 10*10 у арабській позиційній системі, додатково потрібно ще чотири таблиці з вузловими числами римської системи L,C,D,M та доповнення табличного методу логічними процедурами. Наприклад, для операції сумування двох чисел CMLIX + XCIV потрібні таблиці більшого об'єму, ніж 10*10 кожна. Один з варіантів рахунку полягає в представленні чисел, по-перше, розділеними на окремі цифри, по-друге, проведенням операцій віднімання і сумування окремих цифр з використанням таблиць, по-третє, об'єднанням цифр, що залишилися, в єдине число.

CMLIX + XCIV = (-C) + M +L +(-I) + X + (-X) + C +(-I) + V;

Оскільки –C + C = 0; –X + X = 0; – I – I = – II, V – II = III,

то: CMLIX + CIV = M + L + III = MLIII;

Як бачимо, для проведення розрахунків у римській системі необхідно виконувати більше типів операцій, ніж у десятковій арабській позиційній системі числення. Крім того, у римській системі потрібні додаткові логічні перетворення, що суттєво ускладнюють зв'язки між окремими операціями обчислювального процесу.

Часова складність операцій сумування і віднімання в десятковій системі визначається кількістю розрядів у взаємодіючих числах. У римській системі часова складність залежить від порядку розташування цифр у числах. У тих випадках, коли не потрібно утворювати ланцюги логічних операцій, часова складність не перевищує часову складність операцій з десятковими числами. У протилежному випадку, як у розглянутому прикладі, зростання невелике.

Таким чином, за логікою побудови і різноманітністю операцій – програмною складністю, арабська система суттєво простіша за римську, що й довів аль-Хорезмі. За часовою складністю вони майже однакові.

Величезне революційне значення мало використання десяткової систем числення у Європі у всіх сферах життя – побуті, навчанні, торгівлі, суднобудуванні, техніці, географії, астрономії та багато інших. І те, що цей процес співпав з епохою Відродження, не є випадковим.

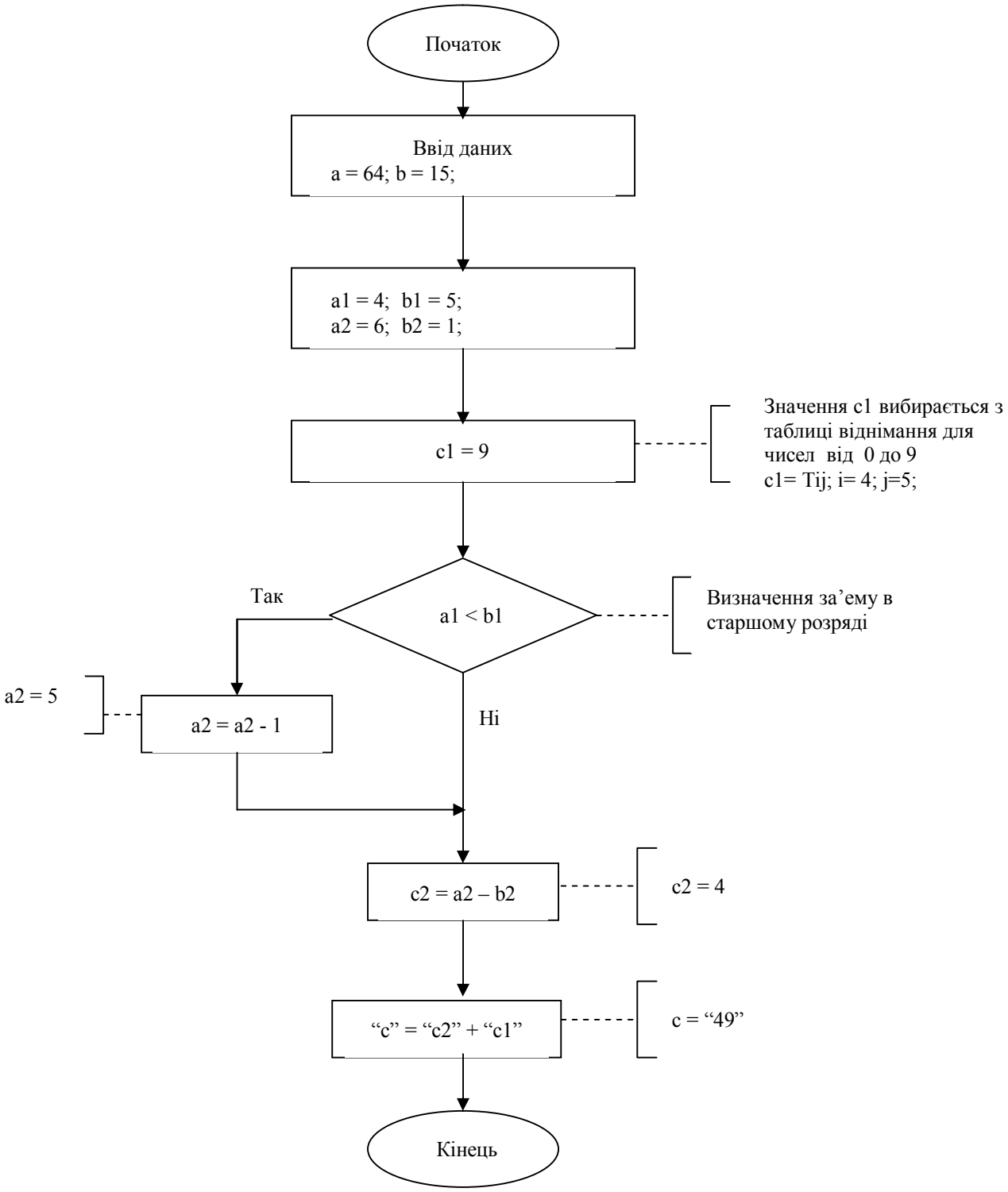
Введення десяткової системи числення, можливо, відіграло вирішальну роль прискорювача у поступових процесах Ренесансу. Значна роль в цьому належить видатному арабському вченому аль-Хорезмі.

1. *Мухаммад ибн Муса ал-Хорезми. Математические трактаты.- Ташкент, 1983.*
2. *2.Юшкевич А.П. История математики в средние века. – М1961.*

Приклад.

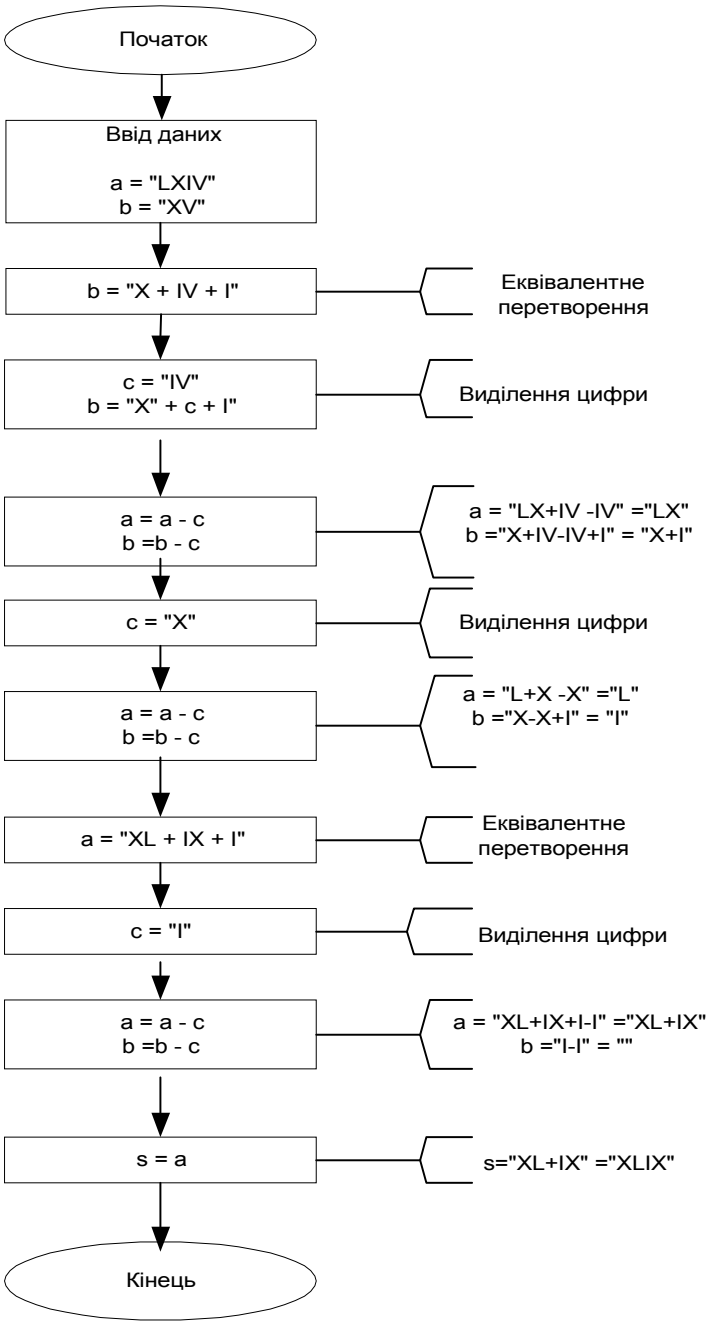
LXIV – XV 64 – 15

Блок-схема адгоритму для десаткової системи числення.



Часова складність L = 7
Програмна складність P =7

Блок-схема адгоритму для римської системи числення.



Еквівалентні перетворення здійснюються за допомогою скінченої таблиці еквівалентних перетворювань .

II=I+I; III=II+I; IV=III+I; V=IV+I; VI=V+I; VII=V+II; VIII=V+III; IX=VIII+I; X=IX+I;.....L=XL+X; ... M=CM+C;

Часова складність L = 10
Програмна складність P =10

ЛАБОРАТОРНА РОБОТА №3

Назва роботи: використання потокового графу алгоритму при проектуванні паралельних обчислень.

Мета роботи: ознайомитися з використанням потокового графу алгоритму при паралельному програмуванні.

*(детальніша інформація про **паралельні обчислення**, **паралельне програмування** та **потоковий граф алгоритму** в матеріалах лекційних занять)*

1.1. Паралельні обчислення.

Паралельні обчислення — це форма обчислень, в яких кілька дій проводяться одночасно. Є кілька різних рівнів паралельних обчислень: ***бітовий***, ***інструкцій***, ***даних*** та ***паралелізм задач***. Паралельні обчислення застосовуються вже протягом багатьох років, в основному в високопродуктивних обчисленнях, але зацікавлення ними зросло тільки недавно, через фізичні обмеження зростання частоти. Оскільки споживана потужність (і відповідно виділення тепла) комп'ютерами стало проблемою в останні роки, паралельне програмування стає домінуючою парадигмою в комп'ютерній архітектурі, в основному в формі багатоядерних процесорів.

Паралельні комп'ютери системи можуть бути грубо класифіковані згідно з рівнем, на якому апаратне забезпечення підтримує паралелізм: багатоядерні процесори; багатопроцесорні комп'ютери — комп'ютери, що мають багато обчислювальних елементів в межах одної машини, також сюди відносимо кластери; MPP та GRID — системи що використовують багато комп'ютерів для роботи над одним завданням. Також Іноді, поряд з цими традиційними комп’ютерними системами, використовуються спеціалізовані паралельні архітектури для прискорення особливих задач.

Програми для паралельних комп'ютерів писати значно складніше, ніж для послідовних, бо паралелізм додає кілька нових класів потенційних помилок, серед яких найпоширенішою є стан гонитви(конкуренції обчислювальних процесів). Комунікація, та синхронізація процесів зазвичай одна з найбільших перешкод для досягнення хорошої продуктивності паралельних програм.

1.2. Паралельні обчислення рівня машинних інструкцій.

Комп'ютерна програма, по суті, є потоком інструкцій, які виконуються процесором. Іноді ці інструкції можна перевпорядкувати, та об'єднати в групи, які потім виконувати паралельно, без зміни результату роботи програми, що відомо як паралелізм на рівні інструкцій. Такий підхід до збільшення продуктивності обчислень переважав з середини 80-тих до середини 90-тих. Сучасні процесори мають багатоетапні конвеєри команд. Кожен етап конвеєра відповідає іншій дії, що виконує процесор. Процесор що має конвеєр з N-ступенями, може одночасно обробляти N інструкцій, кожну на іншій стадії обробки. Класичним прикладом процесора з конвеєром є процесор архітектури RISC, що має п'ять етапів: завантаження інструкції, декодування, виконання, доступ до пам'яті, та запис результату. Процесор Pentium 4 має конвеєр з 35 етапами. На додачу до паралелізму на рівні інструкцій деякі процесори можуть виконувати більш ніж одну інструкцію за раз. Вони відомі як суперскалярні процесори. Інструкції групуються разом, якщо між ними не існує залежності даних. Щоб реалізувати паралелізм на рівні інструкцій використовують алгоритми Scoreboarding та Tomasulo algorithm (який аналогічний до попереднього, проте використовує перейменування регістрів).

1.3. Використання потокового графу алгоритму для паралельного програмування.

Потоковий граф алгоритму описує паралельні обчислення, в яких на заданому фрагменті виконання відсутні залежності керування за даними. Кожна вершина графу на одному ярусі виконується одночасно. Приміром на *рис.1.1* показано виконання швидкого перетворення Фур'є для 8-ми відліків.

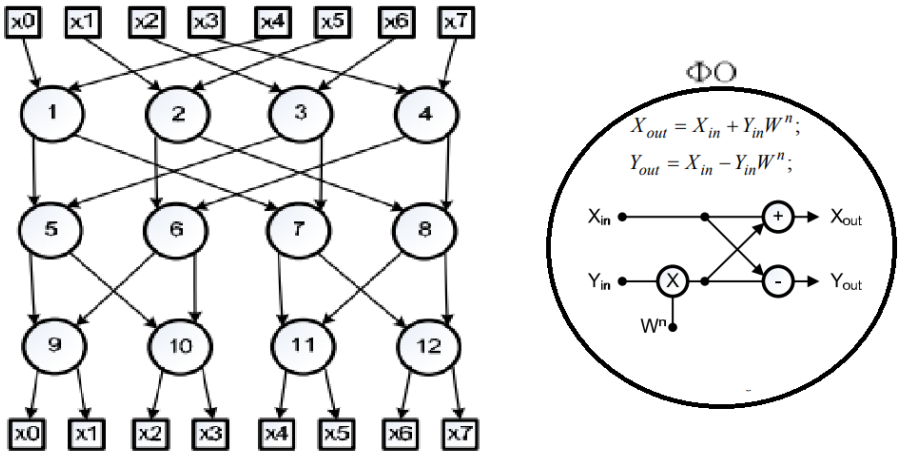
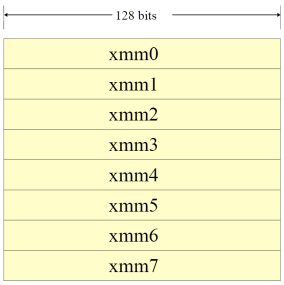


Рис.1.1. Приклад потокового графу алгоритму

1.4. Набір інструкцій SSE2.

SSE (англ. Streaming SIMD Extensions, потокове SIMD-розширення процесора) — це SIMD (англ. Single Instruction, Multiple Data, Одна інструкція — багато даних) набір інструкцій, розроблених Intel, і вперше представлених у процесорах серії Pentium III як відповідь на аналогічний набір інструкцій 3DNow! від AMD, який був представлений роком раніше. Цікаво, що початкова назва цих інструкцій була KIN, що розшифровувалася як Katmai New Instructions (Katmai — назва першої версії ядра процесора Pentium III).



SSE2 використовує вісім 128-бітних регістри (xmm0 до xmm7), що увійшли до архітектури x86 з вводом розширення SSE, кожний з яких трактується як послідовність 2 значень з плаваючою комою подвійної точності. SSE2 включає в себе набір інструкцій, який виконує дії зі скалярними і упакованими типами даних. Також SSE2 містить інструкції для потокової обробки даних з фіксованою комою у тих же 128-бітних xmm регістрах.

Лістинг 1.1

```
float a[4] = { 300.0, 4.0, 4.0, 12.0 };
float b[4] = {   1.5, 2.5, 3.5,  4.5 };
_asm {
movups xmm0, a   ; // помістити 4 змінні з рухомою комою із a в регістр xmm0
movups xmm1, b   ; // помістити 4 змінні з рухомою комою із b в регістр xmm1
mulps xmm1, xmm0 ; // перемножити пакети: xmm1=xmm1*xmm0
                 ; // xmm10 = xmm10*xmm00
                 ; // xmm11 = xmm11*xmm01
                 ; // xmm12 = xmm12*xmm02
                 ; // xmm13 = xmm13*xmm03
movups a, xmm1   ; // вивантажити результати із регістра xmm1 в a
};
```

1.5. Робата з SSE2 на рівні мов програмування високого рівня.

Для роботи з SSE, SSE2, SSE3, SSE4 та AVX в C/C++ зручно використовувати обгортки над цими інструкціями процесора. Їх реалізовують ***intrinsics-функції***, вичерпну інформацію по яких можна отримати за лінком: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ .

2. Приклад.

Математичні вирази зручно подавати у формі потокового графу алгоритму, оскільки в них обо відсутні залежності керування за даними, або їх легко трансформувати в додаткові обчислення для вибору результату. Розглянемо потоковий граф алгоритму розв'язання квадратного рівняння *(рис. 2.1.)* для двократного розпаралелення при одночасному виконанні тільки однотипних інструкцій(відповідно до концепції SSE2).

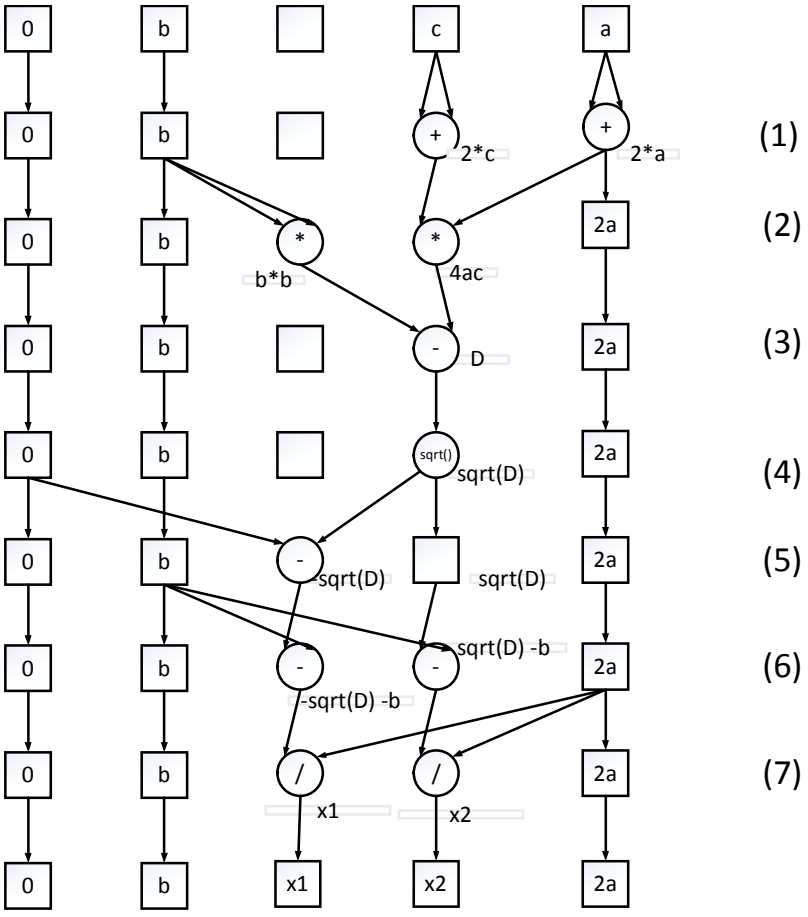


Рис. 2.1. ПГА для розв'язання квадратного рівняння

3. Приклад програми.

Лістинг 3.1

```
// don't forget to use compilation key for Linux: -lm

//-fno-tree-vectorize
//gcc -O3 -no-tree-vectorize
//gcc -O3 -ftree-vectorizer-verbose=6 -msse4.1 -ffast-math
//__attribute__((optimize("no-tree-vectorize")))
//extern "C" __attribute__ ((optimize("no-tree-vectorize")))


#include <stdio.h>
#include <stdlib.h>
//#include <x86intrin.h> // Linux
#include <intrin.h> // Windows
#include <math.h>
#include <time.h>

#define A  0.33333333
#define B  0.
#define C -3.

#pragma GCC push_options
#pragma GCC optimize ("no-unroll-loops")

#define REPEAT_COUNT 1000000
#define REPEATOR(count, code) \
for (unsigned int indexIteration = (count); indexIteration--;){ code; }
#define TWO_VALUES_SELECTOR(variable, firstValue, secondValue) \
	(variable) = indexIteration % 2 ? (firstValue) : (secondValue);

double getCurrentTime(){
	clock_t time = clock();
	if (time != (clock_t)-1) {
		return ((double)time / (double)CLOCKS_PER_SEC);
	}
	return 0.; // else
}

#pragma GCC push_options
#pragma GCC target ("no-sse2")
//__attribute__((__target__("no-sse2")))
void run_native(double * const dArr){
	double * const dAC = dArr;
	double * const dA = &dAC[0];
	double * const dC = &dAC[1];
	double * const dB = &dArr[2];
	double * const dResult = &dArr[4];
	double * const dX1 = &dResult[1];
	double * const dX2 = &dResult[0];

	REPEATOR(REPEAT_COUNT,
		TWO_VALUES_SELECTOR(*dA, 4., A);
		TWO_VALUES_SELECTOR(*dB, 3., B);
		TWO_VALUES_SELECTOR(*dC, 1., C);
		double vD = sqrt((*dB)*(*dB) - 4.*(*dA)*(*dC));
		(*dX1) = (-(*dB) + vD) / (2.*(*dA));
		(*dX2) = (-(*dB) - vD) / (2.*(*dA));
	)
}
#pragma GCC pop_options

void run_SSE2(double * const dArr){
	double * const dAC = dArr;
```

```
	double * const dA = &dAC[0];
	double * const dC = &dAC[1];
	double * const dB = &dArr[2];
	double * const dResult = &dArr[4];
	double * const dX1 = &dResult[1];
	double * const dX2 = &dResult[0];

	__m128d r__zero_zero, r__c_a, r__uORb_b, r__2cORbOR2a_2a,
		r__zero_bb, r__sqrtDiscriminant_zero, r_result;

	r__zero_zero = _mm_set_pd(0., 0.); // init

	REPEATOR(REPEAT_COUNT,
		TWO_VALUES_SELECTOR(*dA, 4., A);
		TWO_VALUES_SELECTOR(*dB, 3., B);
		TWO_VALUES_SELECTOR(*dC, 1., C);
		r__c_a = _mm_load_pd(dAC);
		// r__uORb_b = _mm_load_pd1(dB);
		r__uORb_b = _mm_load1_pd(dB);
		// b b
		r__uORb_b = _mm_unpacklo_pd(r__uORb_b, r__uORb_b);
		// (etap 1)
		r__2cORbOR2a_2a = _mm_add_pd(r__c_a, r__c_a);
		// b 2c
		r_result = _mm_unpackhi_pd(r__2cORbOR2a_2a, r__uORb_b);
		// b 2a
		r__2cORbOR2a_2a = _mm_unpacklo_pd(r__2cORbOR2a_2a, r__uORb_b);
		// bb 4ac (etap 2)
		r_result = _mm_mul_pd(r_result, r__2cORbOR2a_2a);
		r__zero_bb = _mm_unpackhi_pd(r_result, r__zero_zero);
		// zero Discriminant (etap 3)
		r_result = _mm_sub_sd(r__zero_bb, r_result);
		// zero sqrtDiscriminant (etap 4)
		r_result = _mm_sqrt_sd(r_result, r_result);
		r__sqrtDiscriminant_zero = _mm_shuffle_pd(r_result, r_result, 1);
		// sqrtDiscriminant -sqrtDiscriminant (etap 5)
		r_result = _mm_sub_sd(r__sqrtDiscriminant_zero, r_result);
		// (etap 6)
		r_result = _mm_sub_pd(r_result, r__uORb_b);
		// 2a 2a
		r__2cORbOR2a_2a = _mm_unpacklo_pd(r__2cORbOR2a_2a, r__2cORbOR2a_2a);
		// (etap 7)
		r_result = _mm_div_pd(r_result, r__2cORbOR2a_2a);
		_mm_store_pd(dResult, r_result);
	)
}

void printResult(char * const title, double * const dArr, unsigned int runTime){
	double * const dAC = dArr;
	double * const dA = &dAC[0];
	double * const dC = &dAC[1];
	double * const dB = &dArr[2];
	double * const dResult = &dArr[4];
	double * const dX1 = &dResult[1];
	double * const dX2 = &dResult[0];

	printf("%s:\r\n", title);
	printf("%fx^2 + %fx + %f = 0;\r\n", *dA, *dB, *dC);
	printf("x1 = %1.0f; x2 = %1.0f;\r\n", *dX1, *dX2);
	printf("run time: %dns\r\n\r\n", runTime);
}

int main() {
	double * const dArr = (double *)_mm_malloc(6 * sizeof(double), 16);
```

```
        double * const dAC = dArr;
        double * const dA = &dAC[0];
        double * const dC = &dAC[1];
        double * const dB = &dArr[2];
        double * const dResult = &dArr[4];
        double * const dX1 = &dResult[1];
        double * const dX2 = &dResult[0];

        double startTime, endTime;

        // native (only x86, if auto vectorization by compiler is off)
        startTime = getCurrentTime();
        run_native(dArr);
        endTime = getCurrentTime();
        printResult("x86",
               dArr,
               (unsigned int)((endTime - startTime) * (1000000000 / REPEAT_COUNT)));

        // SSE2
        startTime = getCurrentTime();
        run_SSE2(dArr);
        endTime = getCurrentTime();
        printResult("SSE2",
               dArr,
               (unsigned int)((endTime - startTime) * (1000000000 / REPEAT_COUNT)));

        _mm_free(dArr);

        printf("Press any key to continue . . .");
        getchar();
        return 0;
}

#pragma GCC pop_options
```

4. Спрощене завдання.

Скласти програму (C/C++), яка дозволяє провести порівняння двох реалізацій розв'язання квадратного рівняння(звичайний послідовний код та код на основі інструкцій SSE2, що відображає потоковий граф алгоритму) за характеристикою часової складності для таких вхідних даних:

n	**Вхідні дані**		
	c	**b**	**a**
1	111	112	107
2	145	146	141
3	165	166	161
4	185	186	181
5	205	206	201
6	225	226	221
7	245	246	241
8	265	266	261
9	285	286	281
10	305	306	301
11	325	326	321
12	345	346	341
13	365	366	361
14	385	386	381
15	405	406	401
16	425	426	421

17	445	446	441
18	465	466	461
19	485	486	481
20	505	506	501
21	525	526	521
22	545	546	541
23	565	566	561
24	585	586	581
25	605	606	601
26	625	626	621
27	645	646	641
28	665	666	661
29	685	686	681
30	705	706,	701
n – порядковий номер у журналі			

**** Для отримання 50% балів за лабораторну роботу можна використати наявний програмний код з лістингу 3.1. Для отримання 100% балів за лабораторну роботу потрібно написати власний код.***

5. Завдання базового рівня.

Скласти програму (C/C++), яка дозволяє провести порівняння двох реалізацій обчислення виразу(звичайний послідовний код та код на основі інструкцій SSE2, що відображає потоковий граф алгоритму) за характеристикою часової складності для вхідних даних, що вводяться під час роботи програми.

Варіант	**Вираз**
1	y = (a + b) + (c + d) + (e + f) + (g + h)
2	y = (a + b) - (c + d) + (e + f) - (g + h)
3	y = (a + b) * (c + d) + (e + f) * (g + h)
4	y = (a + b) / (c + d) + (e + f) / (g + h)
5	y = (a + b) + (c + d) + (e + f) + (g + h)
6	y = (a + b) - (c + d) + (e + f) - (g + h)
7	y = (a + b) * (c + d) + (e + f) * (g + h)
8	y = (a + b) / (c + d) + (e + f) / (g + h)
9	y = (a + b) + (c + d) + (e - f) + (g - h)
10	y = (a + b) - (c + d) + (e - f) - (g - h)
11	y = (a + b) * (c + d) + (e - f) * (g - h)
12	y = (a + b) / (c + d) + (e - f) / (g - h)
13	y = (a + b) + (c + d) + (e - f) + (g - h)
14	y = (a + b) - (c + d) + (e - f) - (g - h)
15	y = (a + b) * (c + d) + (e - f) * (g - h)
16	y = (a + b) / (c + d) + (e - f) / (g - h)
17	y = (a - b) + (c - d) + (e + f) + (g + h)
18	y = (a - b) - (c - d) + (e + f) - (g + h)
19	y = (a - b) * (c - d) + (e + f) * (g + h)
20	y = (a - b) / (c - d) + (e + f) / (g + h)
21	y = (a - b) + (c - d) + (e + f) + (g + h)
22	y = (a - b) - (c - d) + (e + f) - (g + h)
23	y = (a - b) * (c - d) + (e + f) * (g + h)
24	y = (a - b) / (c - d) + (e + f) / (g + h)
25	y = (a - b) + (c - d) + (e - f) + (g - h)
26	y = (a - b) - (c - d) + (e - f) - (g - h)
27	y = (a - b) * (c - d) + (e - f) * (g - h)
28	y = (a - b) / (c - d) + (e - f) / (g - h)
29	y = (a - b) + (c - d) + (e - f) + (g - h)
30	y = (a - b) - (c - d) + (e - f) - (g + h)
n – порядковий номер у журналі	

6. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.

ЛАБОРАТОРНА РОБОТА №1

Назва роботи: алгоритм; властивості, параметри та характеристики складності алгоритму.

Мета роботи: проаналізувати складність заданих алгоритмів.

1. Загальні відомості.

Здавна найбільшу увагу приділяли дослідженням алгоритму з метою мінімізації часової складності розв'язання задач. Але зміст складності алгоритму не обмежується однією характеристикою. В ряді випадків не менше значення має складність логіки побудови алгоритму, різноманітність його операцій, зв'язаність їх між собою. Ця характеристика алгоритму називається програмною складністю. В теорії алгоритмів, крім часової та програмної складності, досліджуються також інші характеристики складності, наприклад, місткісна, але найчастіше розглядають дві з них – часову і програмну. Якщо у кінцевому результаті часова складність визначає час розв'язання задачі, то програмна складність характеризує ступінь інтелектуальних зусиль, що потрібні для синтезу алгоритму. Вона впливає на витрати часу проектування алгоритму.

Вперше значення зменшення програмної складності продемонстрував аль-Хорезмі у своєму трактаті "Про індійський рахунок". Алгоритми реалізації арифметичних операцій, описані аль-Хорезмі у словесній формі, були першими у позиційній десятковій системі числення. Цікаво спостерігати, як точно і послідовно описує він алгоритм сумування, користуючись арабською системою числення і кільцем (нулем). В цьому опису є всі параметри алгоритму. Це один з перших відомих у світі вербальних арифметичних алгоритмів.

Схема розроблення будь-якого об'єкту складається з трьох операцій: синтез, аналіз та оптимізація *(Рис.1.)*.

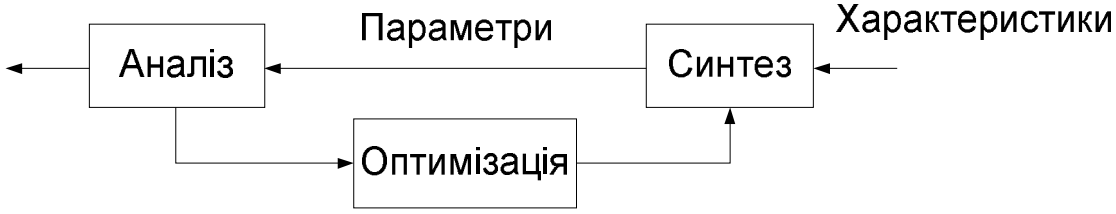


Рис. 1.

Існує два види синтезу: структурний і параметричний. Вихідними даними для структурного синтезу є параметри задачі – сформульоване намагання і набори вхідних даних. В результаті структурного синтезу отримують алгоритм, який розв'язує задачу в принципі.

Параметричний синтез змінами параметрів створює таку його структуру, яка дозволяє зменшити часову складність попередньої моделі. Існує багато способів конструювання ефективних алгоритмів на основі зміни параметрів. Розглянемо спосіб зміни правила безпосереднього перероблення на прикладі задачі знаходження найбільшого спільного дільника двох натуральних чисел..

Алгоритм знаходження найбільшого спільного дільника, яким ми користуємось для цієї цілі і донині, був запропонований Евклідом приблизно в 1150 році до н.е. у геометричній формі, в ньому порівняння величин проводилося відрізками прямих, без

використання арифметичних операцій Алгоритм розв'язку передбачав повторювання віднімання довжини коротшого відрізка від довжини довшого.

Опис алгоритму. Маючи два натуральні числа *a* та *b*, повторюємо *обчислення* для пари значень *b* та залишку від ділення *a* на *b* (тобто *a* mod *b*). Якщо *b*=0, то *a* є шуканим НСД.

Обчислення

Ітераційна версія:

```
НСД( a, b )
      поки b ≠ 0
           c = остача від ділення a на b
           a = b
           b = c
      поверни a
```

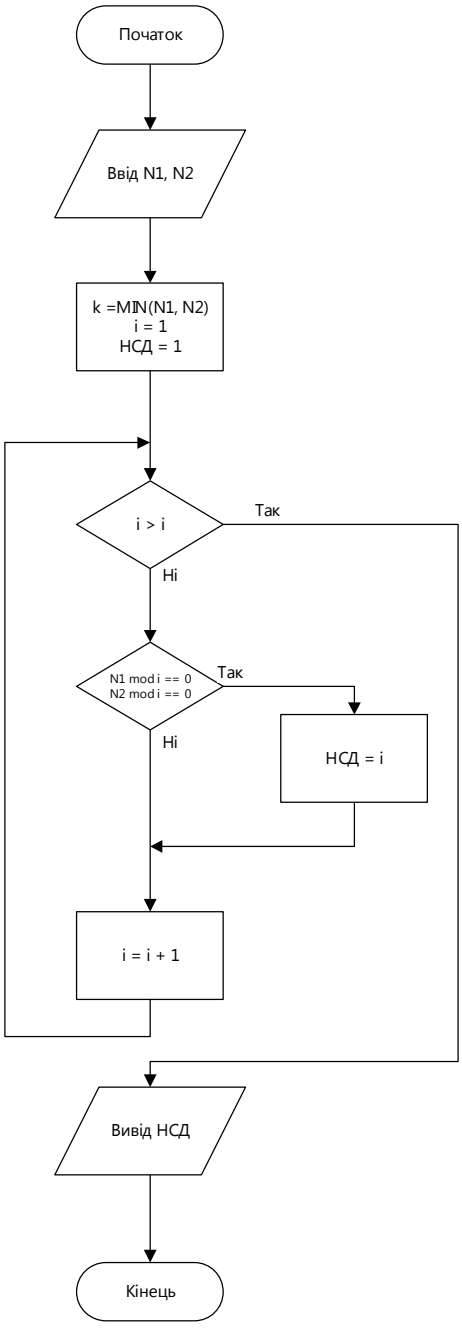
Рекурсивна версія:

```
НСД( a, b )
      якщо b == 0
         поверни a
      інакше
         поверни  НСД( b, остача від ділення a на b )
```

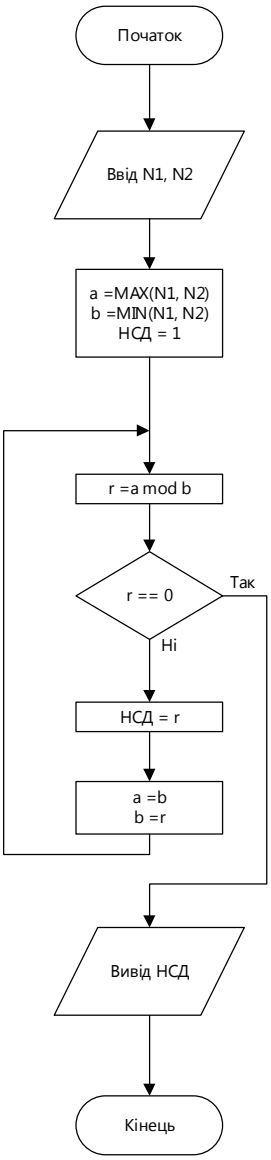
Для того, щоб довести ефективність алгоритму, потрібно порівняти його з таким, який приймається за неефективний. Прикладом такого неефективного алгоритму є процедура послідовного перебору можливих розв’язань задачі. Будемо вважати, що алгоритм перебору утворений в результаті структурного синтезу, на основі вхідних даних та намагання знайти серед всіх допустимих чисел такє, що є найбільшим дільником двох заданих чисел.

Ефективність, як правило, визначається такою характеристикою як часова складність, що вимірюється кількістю операцій, необхідних для розв’язання задачі.

Дослідимо розв’язання задачі знаходження найбільшого спільного дільника двох цілих чисел ($N_1>0$, $N_2>0$, $N_1 \geq N_2$) алгоритмом перебору і алгоритмом Евкліда. Алгоритм перебору заснований на операції інкременту змінної n від одиниці до меншого (N_2) з двох заданих чисел і перевірці, чи ця змінна є дільником заданих чисел. Якщо це так, то значення змінної запам’ятовується і операції алгоритму продовжуються. Якщо ні, то операції алгоритму продовжуються без запам’ятовування. Операції алгоритму закінчуються видачею з пам’яті знайденого останнім спільного дільника. Блок-схема алгоритму приведена на *рис.2(а)*.



а)



б)

Рис2. Блок-схема алгоритму перебору (а) і Евкліда (б).

Адаптований до сучасної арифметики алгоритм Евкліда використовує циклічну операцію ділення більшого числа на менше, знаходження остачі (r) і заміну числа, яке було більшим, на число, яке було меншим, а меншого числа на остачу. Всі перераховані операції виконуються в циклі. Операції циклу закінчуються, коли остача дорівнює нулю. Останній дільник є найбільшим спільним дільником. Блок-схема алгоритму приведена на *рис.2(б)*.

Аналіз цих двох алгоритмів показує, що часова складність алгоритму перебору значно перевищує часову складність алгоритму Евкліда. Для обох алгоритмів часова складність є функцією від вхідних даних, а не їх розміру. В таких випадках при порівнянні ефективності алгоритмів користуються порівнянням часових складностей визначених для найгіршого випадку. Часова складність для найгіршого випадку (Lmax) представляє собою максимальну часову складність серед всіх вхідних даних розміру N.

Часова складність $L_{\max}$ для алгоритму перебору:

$$L_{\max} = C \cdot N_2 \qquad (1)$$

де C – константа , яка дорівнює кількості операцій в кожній ітерації.

Для цілих чисел n $(1 \le n < r_i)$ алгоритм Евкліда знаходження найбільшого спільного дільника має найбільшу часову складність для пари чисел r_{i-1} і r_{i-2} , де 1 , 2 , 3 , … , r_{i-2}, r_{i-1} , r_i – числа Фібоначчі.

Алгоритм Евкліда є ефективним за часовою складністю у порівняні з алгоритмом перебору. Мінімізація часової складності дозволяє за всіх інших рівних умова збільшити продуктивність розв'язання задачі.

2. Приклад програми.

Лістинг 2.1

```
//to measure time it is better to run with Linux

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

#define N1  18
#define N2  27

#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))

#define REPEAT_COUNT 1000000
#define REPEATOR(count, code) \
for (unsigned int indexIteration = (count); indexIteration--;){ code; }
#define TWO_VALUES_SELECTOR(variable, firstValue, secondValue) \
       (variable) = indexIteration % 2 ? (firstValue) : (secondValue);

double getCurrentTime(){
       clock_t time = clock();
       if (time != (clock_t)-1) {
              return ((double)time / (double)CLOCKS_PER_SEC);
       }
       return 0.; // else
}

unsigned long long int f1_GCD(unsigned long long int variableN1, unsigned long long int
variableN2){
    unsigned long long int returnValue = 1;

       for(unsigned long long int i = 1, k = MIN(variableN1, variableN2); i <= k; i++){
           if(!(variableN1 % i || variableN2 % i)){
               returnValue = i;
           }
       }

       return returnValue;
}

unsigned long long int f2_GCD(unsigned long long int a, unsigned long long int b){
       for(unsigned long long int aModB;
              aModB = a % b,
              a = b,
              b = aModB;
       );
```

```
        return a;
}

unsigned long long int f3_GCD(unsigned long long int a, unsigned long long int b){
    if(!b){
            return a;
        }

        return f3_GCD(b, a % b); // else
}

int main() {
    unsigned long long int vN1 = N1, vN2 = N2, a = MAX(vN1, vN2), b = MIN(vN1, vN2),
        vN1_ = vN1, vN2_ = vN2, a_ = a, b_ = b,
        returnValue;

        double startTime, endTime;

        // f1_GCD
        startTime = getCurrentTime();
        REPEATOR(REPEAT_COUNT,
            TWO_VALUES_SELECTOR(vN1, 16, vN1_);
            TWO_VALUES_SELECTOR(vN2, 4, vN2_);
            returnValue = f1_GCD(a, b);
        )
        endTime = getCurrentTime();
        printf("f1_GCD return %d \r\nrun time: %dns\r\n\r\n",
        returnValue,
        (unsigned int)((endTime - startTime) * (double)(1000000000 / REPEAT_COUNT)));

        // f2_GCD
        startTime = getCurrentTime();
        REPEATOR(REPEAT_COUNT,
            TWO_VALUES_SELECTOR(a, 16, a_);
            TWO_VALUES_SELECTOR(b, 4, b_);
            returnValue = f2_GCD(a, b);
        )
        endTime = getCurrentTime();
        printf("f2_GCD return %d \r\nrun time: %dns\r\n\r\n",
        returnValue,
        (unsigned int)((endTime - startTime) * (double)(1000000000 / REPEAT_COUNT)));

        // f3_GCD
        startTime = getCurrentTime();
        REPEATOR(REPEAT_COUNT,
            TWO_VALUES_SELECTOR(a, 16, a_);
            TWO_VALUES_SELECTOR(b, 4, b_);
            returnValue = f3_GCD(a, b);
        )
        endTime = getCurrentTime();
        printf("f3_GCD return %d \r\nrun time: %dns\r\n\r\n",
        returnValue,
        (unsigned int)((endTime - startTime) * (double)(1000000000 / REPEAT_COUNT)));

        printf("Press any key to continue . . .");
        getchar();

        return 0;
}
```

3. Спрощене завдання.

Скласти програму (C/C++), яка дозволяє провести порівняння трьох алгоритмів(алгоритму перебору, ітераційної версії алгоритму Евкліда та рекурсивної версії алгоритму Евкліда) знаходження НСД за характеристикою часової складності для таких вхідних даних:

Варіант	**Вхідні дані**	
	N1	**N2**
1	112	107
2	146	141
3	166	161
4	186	181
5	206	201
6	226	221
7	246	241
8	266	261
9	286	281
10	306	301
11	326	321
12	346	341
13	366	361
14	386	381
15	406	401
16	426	421
17	446	441
18	466	461
19	486	481
20	506	501
21	526	521
22	546	541
23	566	561
24	586	581
25	606	601
26	626	621
27	646	641
28	666	661
29	686	681
30	706	701

**** Для отримання 50% балів за лабораторну роботу можна використати наявний програмний код** з лістингу 2.1. Для отримання 100% балів за лабораторну роботу потрібно написати власний код.*

4. Завдання базового рівня.

Скласти програму (C/C++), яка реалізовує заданий згідно варіанту алгоритм та дозволяє провести порівняння цього алгоритму з аналогічним алгоритмом прямого перебору за характеристикою часової складності для вхідних даних, що вводяться під час роботи програми.

Варіант	**Алгоритм**
1	Алгоритм Косараджу
2	Алгоритм Борувки
3	Алгоритм Крускала
4	Алгоритм Прима
5	Алгоритм Дейкстри
6	Алгоритм Флойда-Воршала
7	Алгоритм Джонсона
8	Алгоритм Беллмана-Форда
9	Алгоритм Данцига
10	Алгоритм Лі
11	Алгоритм Ахо-Корасік
12	Алгоритм Бояра-Мура
13	Алгоритм Бояра-Мура-Горспула
14	Алгоритм Кнута-Моріса-Прата
15	Алгоритм Коменц-Вальтер
16	Алгоритм Рабіна-Карпа
17	Алгоритм Нідлмана-Вунша
18	Алгоритм Барнса-Хата
19	Алгоритм Діксона
20	Алгоритм Луна
21	Алгоритм Штрассена
22	Алгоритм Копперсміта-Віноградa
23	Алгоритм Пана
24	Алгоритм Біні
25	Алгоритми Шенхаге
26	Алгоритм Малхотри-Кумара-Махешварі
27	Алгоритм Галіла-Наамада
28	Алгоритм Ахьюа-Орліна-Тар'яна
29	Алгоритм Черіяна-Хейджрапа-Мехлхорна
30	Алгоритм Келнера-Мондри-Спілман-Тена

5. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.