Даний текст є поясненням до задач “Дужки”, “Входження елемента” та “Томи”, що були запропоновані на заочний відбірковий тур міської олімпіади з інформатики (м. Черкаси, 2001–02 н.р.). З приводу будь-яких питань по цим задачам можна звернутися до автора даних пояснень за адресою `ilya@cdu.edu.ua`.

Ілля Порубльов

1 Умови задач

1.1 Задача “ДУЖКИ”

На вході задається рядок–послідовність дужок. Потрібно з’ясувати, чи утворюють вони правильний дужковий вираз.

У вхідному текстовому файлі `PARS.DAT` задані: 1-ий рядок містить число N (кількість тестів у файлі), далі N рядків, кожен з котрих містить окрему послідовність. Послідовності гарантовано не містять ніяких інших символів, крім перелічених в умові відповідної версії дужок.

Програма повинна створити текстовий файл `PARS.SOL` з N рядків, кожен рядок повинен містити 1, якщо відповідна послідовність є правильним дужковим виразом, або 0, якщо не є.

Приклад для версії A):

PARS.DAT	PARS.SOL
3	1
(())()	0
(((())))	0
))((	

Приклад для версії C):

PARS.DAT	PARS.SOL
3	1
(<>)()	0
(<>)<}{>	0
[)(]	

Версія A) Дужки можуть бути лише круглі, “(” та “)”, довжини послідовностей від 4 до 100.

Версія B) Дужки можуть бути лише круглі, “(” та “)”, довжини послідовностей від 100 до 50,000.

Версія C) Дужки можуть бути круглі “(” та “)”, квадратні “[” та “]”, фігурні “{” та “}” або кутові “<” та “>”, дужка повинна закриватися дужкою того самого типу; довжини послідовностей від 4 до 100.

Версія D) Дужки можуть бути круглі “(” та “)”, квадратні “[” та “]”, фігурні “{” та “}” або кутові “<” та “>”, дужка повинна закриватися дужкою того самого типу; довжини послідовностей від 100 до 100,000.

Версія E) Дужки можуть бути лише круглі, “(” та “)”, довжини послідовностей від 50,000 до 5,000,000.
[Кожен тестовий файл вважається зарахованим, якщо побудовано правильні відповіді на *усі* послідовності цього файла.]

1.2 Задача “ВХОДЖЕННЯ ЕЛЕМЕНТА”.

У вхідному файлі дано послідовність натуральних чисел. Потрібно з’ясувати, чи є у цій послідовності число, котре повторюється настільки часто, що кількість його повторів складає строго більше половини усієї кількості чисел у послідовності. [Наприклад, для послідовності 7,7,3,7,7,2,1 таким числом є 7, для 1,2,3,4,5 та для 0,1,0,1,0,1 таких чисел немає.]

Перший рядок вхідного текстового файла `FREQSEQ.DAT` містить N — кількість чисел у послідовності, другий — числа послідовності, розділені пропусками.

Вихідний файл `FREQSEQ.SOL` має бути текстовим файлом, котрий у випадку, якщо таких чисел нема, містить єдиний рядок з числом 0, а якщо таке число є, то у першому рядку має бути число 1, а у другому — самé це число.

Приклади:

FREQSEQ.DAT	FREQSEQ.SOL
7 7 7 3 7 7 2 1	1 7
5 1 2 3 4 5	0
6 0 1 0 1 0 1	0

Версія A) значення елементів послідовності від 1 до 100, довжина послідовності до 100 елементів.

Версія B) значення елементів послідовності від 1 до 1,000,000, довжина послідовності до 2,000 елементів.

Версія C) значення елементів послідовності від 1 до 500, довжина послідовності до 100,000 елементів.

Версія D) значення елементів послідовності від 1 до 1,000,000, але серед них не більше ніж 5,000 *різних* значень, довжина послідовності до 500,000 елементів.
[так, у послідовності 100000, 1, 100000, 1, 100000, 100000, 1, 100000, 1, 100000 значення елементів відрізняються одне від одного досить сильно, але *різних* значень лише 2]

Версія E) елементи послідовності є довільними додатними числами типу `longint` (від 1 до 2,147,483,647), довжина послідовності до 5,000,000 елементів, кількість різних чисел до 3,000,000.

1.3 Задача “ТОМИ”

Літературний твір складається з C частин. Його потрібно надрукувати у B томах ($B \le C$). Для кожної частини відомий її розмір (у сторінках). Потрібно написати програму, що знаходить таке розподілення частин по томах, щоб кількість сторінок у *найтовщому* з отриманих томів була *якнайменшою*. Кожну частину завжди починають з нової сторінки, тому товщина тому є сума довжин частин, що входять до нього. Розривати частину не можна, вона повинна повністю міститися у якомусь томі.

Якщо існує кілька різних рівноцінних оптимальних розв’язків, потрібно вивести будь-який один з них.

У версіях A), B), C) частини являють собою розділи єдиного твору і мусять бути надруковані строго по порядку; у версіях D), E), F) частини є окремими оповіданнями, котрі можна друкувати у довільному порядку. L_1, L_2 і т.д. означає кіль-ть сторінок у 1-й, 2-й і т.д. частинах, $SumL$ означає сумарну кількість сторінок ($L_1 + L_2 + \ldots + L_C$).

Вхідні дані (`VOLUMES.DAT`) $C+1$ рядок з даними про кіль-ть розділів, томів, розмір кожного розділу:

$C \quad B$
L_1
L_2
$\ldots$
L_C

Вихідні дані (`VOLUMES.SOL`) для версій A), B), C):

Товщина найтовщого тому;

порожній рядок;

C рядків, що містять номери початкового та останнього розділів кожного тому.

Приклад:

VOLUMES.DAT	VOLUMES.SOL
5 3	600
300	
300	1 2
500	3 3
300	4 5
300	

Вихідні дані (`VOLUMES.SOL`) для версій D), E), F):

Товщина найтовщого тому;

порожній рядок;

C рядків, кожен з котрих містить перелік (в довільному порядку) номерів частин, що входять до даного тому.

Приклад:

VOLUMES.DAT	VOLUMES.SOL
5 3	600
300	
500	1 3
300	2
300	4 5
300	

Версія A) частини мусять іти по порядку, кількість томів гарантовано $B = 2$, кількість частин $2 \le C \le 500$, загальна кількість сторінок $C \le SumL \le 10,000$.

Версія B) частини мусять іти по порядку, кількість томів $3 \le B \le 7$, кількість частин $B \le C \le 40$, загальна кількість сторінок $C \le SumL \le 20,000$.

Версія C) частини мусять іти по порядку, кількість томів $5 \le B \le 40$, кількість частин $B \le C \le 250$, загальна кількість сторінок $C \le SumL \le 30,000$.

Версія D) частини можуть іти в довільному порядку, кількість томів гарантовано $B = 2$, кількість частин $2 \le C \le 20$, загальна кількість сторінок $C \le SumL \le 1,000$.

Версія E) частини можуть іти в довільному порядку, кількість томів $3 \le B \le 5$, кількість частин $B \le C \le 10$, загальна кількість сторінок $C \le SumL \le 1,000$.

Версія F) частини можуть іти в довільному порядку, кількість томів $10 \le B \le 50$, кількість частин $B \le C \le 200$, загальна кількість сторінок $C \le SumL \le 20,000$.

[Для задачі "ТОМИ" максимальна кількість балів за тест буде нарахована, якщо знайдено оптимальний розв'язок, але частину балів буде нараховано також за наближені відповіді, котрі є допустимими (знайдене розбиття на томи включає кожну частину в точності в один том, кількість сторінок у найтовщому томі такого розбиття підрахована правильно) і не зовсім оптимальними (насправді існує інше правильне розбиття, при котрому товщина найтовщого тому трохи менша)]

[для версій A) та D) вхідні дані містять число B, котре в усіх тестах рівне 2.]

2 Розв'язання задачі "Дужки"

2.1 Версії A), B), E)

Вираз, що складається з самих лише круглих дужок "(" та ")" правильний тоді, коли кількість відкриваючих та кількість закриваючих дужок однакові, причому в кожній парі дужка спочатку відкривається, а потім закривається.

Заведемо лічильник `Nopen` і покладемо його значення на початку рівним нулеві. Читаючи (посимвольно) рядок із дужковим виразом, будемо на кожному прочитаному символі "(" збільшувати його на 1, на кожному ")" — зменшувати на 1. Тобто, в кожен момент часу в `Nopen` подається кількість відкритих і ще не закритих дужок. Якщо у якийсь момент `Nopen<0`, маємо закриваючу дужку без попередньої відкриваючої, і увесь рядок гарантовано не є правильним виразом (і далі його можна не перевіряти; але оскільки ми подаємо кілька тестів у одному файлі, потрібно все-таки прочитати (або пропустити) цей рядок до кінця). Якщо по закінченню рядка маємо `Nopen`$\neq$`0`, кількість дужок незбалансована і рядок не є правильним виразом. Якщо жодна з цих ситуацій не настала, то рядок є правильним дужковим виразом.

Наголосимо, що даний алгоритм є *однопрохідним*: він один раз читає вхідні дані, по мірі прочитання виконує над ними деякі дії і навіть не зберігає всіх даних у пам'яті, а отже — може обробляти вхідні дані *дуже* великих розмірів.

2.2 Версії C), D)

На жаль, нібито інтуїтивно зрозуміле поняття «правильного дужкового виразу» деякі учасники все-таки зрозуміли неадекватно.

Правильна трактовка «правильного дужкового виразу» така: потрібно, щоб дужки були збалансовані й кожна дужка 1) закривалась пізніше, ніж вона відкривається; 2) закривалась дужкою того самого типу, яким відкрилася (кругла — круглою і т.д.); 3) закривалась після того, як закриються усі дужки, відкриті всередині неї. Приклади рядків, що згідно відповідних пунктів 1)–3) не є правильними дужковими виразами: `)(`, `([)]`, `[(])`.

Прочитано	Не закриті
(	(
((	((
(([	(([
(([])	((
(([]{	(({
(([]{}	((
(([]{})	(
(([]{})<	(<
(([]{})<>	(
(([]{})<>)	

Мал. 1: Список ще не закритих дужок для версій C), D).

Отже, в кожен момент часу (поки не настав кінець рядка) у правильному дужковому виразі можна або відкривати нові дужки, або закривати саме таку, як остання з відкритих і ще не закритих.

Будемо підтримувати перелік відкритих і ще не закритих дужок. Коли нам трапляється відкриваюча дужка, її потрібно дописати у кінець цього переліку, і вона стає останньою відкритою і не закритою. Коли закриваюча — потрібно подивитися, яка була остання відкриваюча; якщо вони різних типів, то рядок не є правильним виразом, інакше потрібно вилучити цю дужку з переліку ще не закритих. По закінченню правильного дужкового виразу перелік ще не закритих дужок має бути порожнім.

Для подання такого переліку ще не закритих дужок доцільно використати стандартне подання стеку в масиві: мати масив `stack` типу, наприклад, `array[1..60000] of char`, та цілу змінну `top`. Масив міститиме елементи цього переліку, а змінна `top` — поточну кількість елементів у цьому переліку; при цьому елемент `stack[1]` є найбільш зовнішньою не закритою дужкою, а `stack[top]` — останньою не закритою. Щоб додати відкриваючу дужку, що зберігається у змінній `ch` типу `char`, потрібно написати `top:=top+1; stack[top]:=ch`, щоб вилучити останню дужку — `top:=top-1`.

Тут ми так само не потребуємо зберігати усі вхідні дані, але мусимо зберігати всі відкриті і ще не закриті дужки. Отже, "в середньому" можна обробляти досить великі вхідні файли, але у найгіршому випадку (спочатку всі дужки відкриваються, потім усі закриваються) обмеженість розміру масиву призводить до обмеження на допустимий розмір вхідних даних.

2.3 Інші можливі алгоритми

Алгоритми, наведені у пунтах 2.1 та 2.2, не мають недоліків, що робили б доцільним пошук інших алгоритмів; але все-таки наведемо один альтернативний варіант.

До версій A), B) та C) (а при великому бажанні і до D)) можна застосувати такий алгоритм: для версій A) і B) — щоразу, зустрівши в рядку пару символів `()` (тобто дужка відкрилася і тут же закрилася), вилучати цю пару символів з рядка; для C) [і D)] — те саме для будь-якої з чотирьох пар `()`, `[]`, `{}`, `<>`. Припиняється цей цикл або тоді, коли рядок стає порожнім, або тоді, коли він стає непорожнім, але таким, що нема жодного входження потрібної пари символів. У першому випадку початковий рядок був правильним дужковим виразом, у другому — не був.

Такий алгоритм правильний, але набагато програє наведеним вище у плані ефективності. Перш за все, він потребує набагато більше пам'яті, а у згаданих в пояснювальній записці DOS-середовищах розмір масиву не може перевищувати 64K. Він також потребує набагато більше часу на виконання, і на значній частині тестів для версії B) така програма не встигає завершити роботу за 60 секунд.

Ще інших правильних алгоримів розв'язання задачі «Дужки» ми не знаємо.

3 Розв'язання задачі "Входження елемента"

Будь-які вхідні дані цієї задачі можна обробити алгоритмом, що написаний для версії E); розв'язки «молодших» версій подані тут тому, що ці алгоритми більш «природні» й можуть бути корисними в багатьох інших ситуаціях, а також щоб показати, яким чином на олімпіадах з інформатики одну й ту саму задачу можна розв'язувати на зовсім різних рівнях складності (й отримувати за такі різні розв'язки різну кількість балів).

3.1 Версія A)

Дану версію можна розв'язувати хоч алгоритмом для версії B), хоч (що трохи краще) алгоритмом для версії C). Нічого ще простішого мабуть і не існує...

3.2 Версія B)

Будемо підтримувати кілька масивів однакової довжини `N` $\le$ 2000. Масив `a` міститиме самі значення елементів послідовності, масив `n` — кількості входжень (тобто, елемент `n[i]` вказує, скільки разів у послідовності зустрічаються елементи зі значенням `a[i]`. Наприклад, для послідовності з 9 елементів 7 7 3 7 7 2 1 7 2 вміст масивів має бути таким:

i	1	2	3	4	5	6	7	8	9
`a[i]`	7	7	3	7	7	2	1	7	2
`n[i]`	5	5	1	5	5	2	1	5	2

Щоб заповнити *кожен* з елементів `n[i]`, передивляємося *увесь* масив `a`; потім продивляємось масив `n`, шукаючи в ньому значення, більші за половину загальної кількості елементів.

3.3 Версія C)

Завдяки обмеженості можливих значень елементів послідовності, можна застосувати наступний дуже ефективний алгоритм: заведемо масив `num:array[1..500] of longint` (діапазон індексів масиву відповідає діапазону можливих значень елементів); на початку покладемо значення усіх його елементів рівними нулеві; щоразу, прочитавши з вхідного файлу якесь число `i`, будемо збільшувати на 1 значення елементу `num[i]`. Отже, по прочитанню входу кожен елемент `num[i]` означатиме кількість входжень елемента `i`.

3.4 Версія D)

Застосувати в чистому вигляді алгоритм версії B) або версії C) не вдасться через обмеження по пам'яті; але після ознайомлення з попередніми алгоритмами загальна ідея алгоритму для даної версії досить очевидна: потрібно підраховувати (і зберігати у пам'яті!) входження *лише* тих елементів, котрі дійсно входять до послідовності.

Одна з найпростіших реалізацій цієї ідеї така. Будемо підтримувати масив записів `mass:array[1..5000] of rec`, розміри котрого визначаються максимально можливою кількістю різних елементів у вхідній послідовності, а кожен елемент є записом, що складається з двох полів: `value` (значення елементу) та `num` (кількість входжень елемента з таким значенням). Крім само́го цього масиву потрібна ще цілочисельна змінна `used`, котра буде позначати кількість реально використаних комірок цього масиву. На початку потрібно покласти `used:=0`; прочитавши черговий елемент `el` із вхідного файлу, будемо збільшувати на 1 кількість його входжень. Для цього переглядаємо елементи масива, починаючи з 1-ого. Якщо ми натрапляємо на такий i-ий елемент, що `mass[i].value=el`, то це означає, що такий елемент вже траплявся раніше і потрібно збільшити на 1 `mass[i].num`; якщо ж передивилися всі елементи з 1-ого по `used`-ий, а на входження елементу `el` не натрапили, то його потрібно дописати в кінець масиву: `used:=used+1; mass[used].value:=el; mass[used].num:=1`.

Відмінність цього алгоритму від алгоритму версії C) у тому, що тоді потрібний елемент знаходився за одне звернення до масиву, а тепер його доводиться шукати, переглядаючи масив. Це суттєво знижує ефективність, і на вхідних даних описаного розміру програма може не встигнути обробити вхід за 60 секунд. Тестові дані до версії D) підібрані так, щоб правильна реалізація описаного алгоритму отримала близько $\frac{2}{3}$ від можливої кількості балів, а у близько $\frac{1}{3}$ випадків настав Time Overflow.

Цілком подібну ідею можна реалізувати і помітно ефективніше. Описувати ці методи докладно зараз не будемо, бо це досить громіздко і не зовсім відповідає меті даних пояснень до задач, але назвемо структури даних, докладно описані у багатьох класичних книгах, котрі дозволяють отримати набагато ефективнішу реалізацію. Це можуть бути різні модифікації впорядкованих бінарних дерев, хеш-таблиці, тощо.

3.5 Версія E)

Можна спробувати використати «зовнішню» версію алгоритма версії B): так само підраховувати кількості входжень, але замість масиву `a` використовувати сам вхідний файл, а масив `n` насправді не потрібен: якщо значення поточного елемента входить у послідовність $\leq \frac{n}{2}$ разів, то про це можна зразу забути, а якщо більше за половину разів — можна відразу виводити відповідь і не робити подальших переглядів. Такий алгоритм буде правильним, але неприпустимо повільним за часом виконання. Зате у часткових випадках, коли число, що зустрічається більше ніж половину разів, буде у вхідному файлі першим або навіть другим–третім, правильна відповідь буде отримана досить швидко.

Якщо спробувати використати алгоритм версії E), на запам'ятовування усього підряд просто не вистачить пам'яті. Можна запропонувати які-небудь евристичні правила на зразок «поки у масиві ще є місце, запам'ятовувати, а коли вже не стане, то пропускати нові значення», або ж (на перший погляд більш інтелектуальне правило) «економити місце, забуваючи значення, що зустрічалися малу кількість разів і досить давно». Звичайно, такі правила дозволяють у багатьох окремих випадках *якось* розв'язати задачу, але вони можуть і не призводити до правильного результату. Контрприклад до першого з наведених правил дуже простий: нехай спочатку зустрічаються по 1 разу всі числа від 10 до 900 000, а потім мільйон разів число 3. Щодо другого правила, то його можна (хоча і досить складно технічно) реалізувати так, щоб щойно описаний вхід був проаналізований правильно. Але можна побудувати контрприклад і до такого способу розв'язування. Наприклад, спочатку іде 3 рази число 7, потім числа від 10 до 10010 по 100 разів кожне (згідно правила, на цей момент про 3 входження числа 7 треба вже забути), а потім мільйон разів число 7. Оскільки в цій послідовності 2 000 003 елементів, і 1 000 003 з них рівні 7, то шуканий елемент існує і рівний 7; але оскільки ми про перші 3 входження забули, то можемо дійти помилкового висновку ніби жоден елемент не зустрічається строго більше ніж половину разів.

Ще одна ідея (котра може бути *дуже і дуже корисною у багатьох інших випадках*, і котра при великому бажанні може бути досить успішно застосована до версії E) замість згаданих дерев та хеш-таблиць) — відсортувати елементи[1]. Після цього знайти кількість входжень будь-якого значення буде просто, бо однакові значення йтимуть підряд. Але такий спосіб, навіть при використанні найефективнішого способу зовнішнього сортування, буде все-таки неприйнятним для версії F) за часом виконання.

Отже, маємо просто-таки скандал! Запам'ятовувати не можна, забувати теж не можна! Як же все-таки розв'язати таку задачу?

Скандал виник тому, що ми (в *усіх правильних* з наведених алгоритмів) намагалися зробити те, чого від нас насправді не вимагають. А саме, ми скрізь спиралися на загальну алгоритмічну ідею «підрахувати точні кількості входжень для усіх значень елементів».

Отже, *потрібна інша принципова алгоритмічна ідея!*

Вгадати ідею оптимального алгоритму досить важко, але можна. Наприклад, ідея стане досить очевидною, якщо почати реалізовувати розв'язок із використанням хеш-функцій.

Спробуємо підрахувати загальну кількість елементів, що закінчуються на 0, загальну кількість елементів, що закінчуються на 1, на 2, ..., на 9. Такий аналіз можна провести за один прохід по вхідному файлу, для цього потрібно завести один масив розміром 10 елементів[2].

Будь-яке значення елементу послідовності закінчується якоюсь цифрою. І якщо існує такий елемент, що його значення повторюється більше ніж $\frac{N}{2}$ разів, то і кількість елементів, що закінчуються на відповідну цифру, буде більшою за $\frac{N}{2}$.

Дослідимо докладніше, що ж звідки слідує. Наприклад, якщо числом, що повторюється більше ніж $\frac{N}{2}$ разів, є число 100, то звідси *слідує*, що чисел, котрі закінчуються на 0, більше ніж $\frac{N}{2}$. Але з того, що чисел, котрі закінчуються на 0, більше ніж $\frac{N}{2}$, *не слідує*, ніби існує якесь число, кратне 10, котре має більше ніж $\frac{N}{2}$ входжень. Можлива навіть ситуація, що всі числа різні, і при цьому всі кратні 10. Але з того, що більше ніж $\frac{N}{2}$ елементів закінчуються на 0, випливає: *якщо існує число, що має більше ніж $\frac{N}{2}$ входжень, то воно обов'язково закінчується на 0.* Якщо ж для *усіх* цифр $i = 0, 1, \ldots, 9$ кількість входжень чисел, що закінчуються цифрою i, не перевищує $\frac{N}{2}$, то шуканого числа, що має більше ніж $\frac{N}{2}$ входжень, також не існує.

Але подібний підрахунок можна робити не лише для останньої цифри. Зробимо в точності такі самі підрахун-

[1] щодо способів впорядкування послідовностей таких розмірів — див. у літературі за ключовими словами *зовнішнє сортування* та *сортування злиттям*

[2] Див. алгоритм для версії C) у пункті 3.3

ки для усіх 10 розрядів. В результаті або отримаємо, що в якомусь розряді жодна цифра не повторюється більше ніж $\frac{N}{2}$ разів (тоді шуканого числа не існує), або отримаємо, що справедливе твердження *якщо існує число, що має більше ніж $\frac{N}{2}$ входжень, то це число, котре складається з цифр $\overline{a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0}$, де a_0 — цифра, що зустрічається більше ніж $\frac{N}{2}$ разів у розряді одиниць, a_1 — цифра, що зустрічається більше ніж $\frac{N}{2}$ разів у розряді десятків, і так далі до розряду мільярдів включно.* Ще раз прочитаємо вхідний файл від початку до кінця, підраховуючи кількість входжень $\overline{a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0}$. І потім порівняємо цю кількість з $\frac{N}{2}$.

Отже, цей алгоритм *двопрохідний*: за перший прохід підраховуємо кількості входжень окремих цифр (маючи 10 масивів розміром 10 кожен), на другому перевіряємо скільки ж насправді разів зустрічається число $\overline{a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0}$.

При реалізації цієї ідеї технічно зручніше трохи модифікувати спосіб підрахунку: використовувати не значення десяткових розрядів, а значення чотирьох байтів, з котрих складається довге ціле (відповідно, використати 4 масиви на 256 елементів кожен). Але подібні дрібні модифікації є вже справою смаку програміста.

4 Розв'язання задачі "Томи"

Легко бачити, що версії A), B), C) з одного боку, та версії D), E), F) з іншого боку фактично є двома *різними* задачами, і потрібно реалізовувати *щонайменше дві програми* — відповідно для версій A), B), C) та для версій D), E), F).

Фактично ми пропонуємо будувати 5 програм — для версії A), для версій B)–C), для версії D), для версії E) і для версії F).

Алгоритми, котрі ми пропонуємо у пунктах 4.1.2 та 4.2.1 є прикладами застосування *принципу оптимальності динамічного програмування Беллмана* (або скорочено *динамічного програмування*).

Детальне пояснення принципу оптимальності та строге математичне доведення правильності його застосування до даної задачі виходять за межі пояснення розв'язків задач заочного туру; усіх зацікавлених відсилаємо до інших джерел. Оскільки під «динамічним програмуванням» у різних випадках розуміють занадто різні речі, іноді просто не пов'язані між собою, наведемо конкретні посилання. Серед книг, що вийшли останнім часом і на даний момент не є бібліографічною рідкістю, можна відзначити: Т. Кормен, Ч. Лейзерсон, Р. Ривест, «Алгоритмы: построение и анализ», Москва, МЦНМО, 2001 — класичний великий і докладний університетський підручник з аналізу та побудови алгоритмів, наведене в ньому пояснення методології динамічного програмування є найкращим з усіх відомих нам викладень; Андрей Ставровский, «Турбо Пакаль 7.0. Учебник», Киев, BHV, 2000 — підручник, що досить вдало поєднує викладення мови програмування Паскаль з викладенням загальних основ програмування і містить багато цікавих задач.

4.1 Версії A)–C)

4.1.1 Версія A)

Ця версія дуже проста і пропонувалася для того, щоб усі більш-менш уважні учасники змогли її розв'язати. Потрібно просто знайти загальну кількість сторінок в усіх розділах ($SumL$), знайти, для якого са́ме розділу сума довжин оповідань з першого по поточний приблизно рівна $\frac{SumL}{2}$, і перевірити, чи сам цей розділ краще включити до першого тому, чи до другого.

Звичайно, задачі версії A) можна розв'язувати також і алгоритмом, побудованим для версії C). Але, щоб реалізувати все в одній програмі, доводиться тримати дуже велику таблицю динамічного програмування: у версії A) дуже великі значення C, а у версії C) досить великі значення B. При програмуванні в Turbo Pascal це створює серйозні технічні незручності.

4.1.2 Версії B), C)

Замість заданої задачі будемо розглядати відразу серію однотипних задач. Кількості сторінок в i-ому розділі (L_i) будуть такими, як задано у вхідних даних, а різними задачами серії будуть такі: *«Яка мінімальна можлива товщина найтовщого тому, якщо (усі) розділи з 1-ого по j-ий потрібно видати у i томах?»*. Будемо називати кожну таку задачу (i,j)-підзадачею. Якщо задану у вхідних даних кількість розділів позначити як C, а кількість томів як B, то потрібна задача є (B,C)-підзадачею.

Виникає питання: яка користь у тому, щоб замість однієї-єдиної задачі розв'язувати цілу серію, куди потрібна задача входить частковим випадком?..

Взагалі кажучи, досить часто якраз *узагальнення* дає ключ до *розробки* ефективного *алгоритму*. А досягається це тоді, коли вдається побудувати, по-перше, нескладний спосіб переходу від «великих» задач до менших задач того самого типу, і, по-друге, простий спосіб розв'язувати всі «малі» задачі.

«Малими» будуть задачі, де кількість томів $i = 1$. Розв'язок будь-якої $(1,j)$-підзадачі тривіальний — усі частини з 1-ої по j-ту треба помістити у 1-ий том, причому «найтовщим з усіх томів» буде 1-ий том.

Розглядаючи (i,j)-підзадачу для довільних i та j з діапазону $1 < i \le B$, $i \le j \le C - B + i$, помічаємо, що у оптимальному розв'язку (i,j)-підзадачі $(i-1)$-ий том повинен закінчуватися якимось k-им розділом (чому дорівнює це k, ми поки що не знаємо), а усі розділи від $(k+1)$-ого по j-ий мають бути включені до i-ого тому.

Найтовщим з i томів може бути або якийсь від 1-ого по $(i-1)$-ий, або i-ий. Товщина найтовщого з томів від 1-ого по $(i-1)$-ий — це розв'язок $(i-1,k)$-підзадачі, а товщина i-ого тому — сума довжин розділів від $(k+1)$-ого по j-ий. Узявши з цих двох чисел максимум, отримаємо товщину найтовщого з усіх i томів. Оскільки ми використовуємо розв'язок $(i-1,k)$-підзадачі й не знаємо наперед, чому са́ме дорівнює k, потрібно *перебрати* можливі значення k. Отже, при $i \ge 2$

$$T_{ij} = \min_k \left(\max\left(T_{i-1,k}, sz_{k+1..j}\right)\right), \qquad (1)$$

де T_{ij} — товщина найтовщого тому в оптимальному розв'язку (i,j)-підзадачі, $sz_{k+1..j}$ — сума довжин розділів від $(k+1)$-ого по j-ий. Діапазон перебирання k a priori мусить бути $i-1 \le k \le j-1$, але практично його можна трохи скоротити, проводячи від $k = j-1$ і щокроку зменшуючи на 1, доки $sz_{k+1..j}$ не стане більшим за вже знайдену оцінку T_{ij}.

Формулу (1) зручно реалізувати у вигляді рекурсивної функції, але вся потужність і ефективність цієї формули буде збережена лише за умови, що це буде так звана *memoized recursion* (*рекурсія із запам'ятовуваннями*): обчислюючи T_{ij}, потрібно перш за все перевірити, чи не розв'язували ми (i,j)-підзадачу раніше; якщо не розв'язували, то розв'язати її за формулою (1) із використанням рекурсивних викликів і запам'ятати результат у (i,j)-ій комірці деякої таблиці результатів; якщо ж (i,j)-підзадачу вже розв'язували, то достатньо просто взяти відповідь з таблиці результатів.

При бажанні можна уникнути використання рекурсії, заповнюючи таблицю оцінок по рядках: спочатку повністю рядок $i = 2$, потім рядок $i = 3$ і так далі до $i = B$. Порівнюючи ці два способи реалізації формули (1), можна відзначити, що недоліки рекурсивної реалізації — ризик переповнення стеку і накладні витрати часу на виконання рекурсивних викликів, а перевага рекурсивної реалізації над ітеративним заповненням — можливість згаданого вище звуження діапазону перебирання k (тобто, уникнення витрат на розв'язування багатьох гарантовано не потрібних підзадач).

Формула (1) вказує спосіб підрахунку T_{ij}, а нам потрібно вивести не лише мінімально можливу товщину найтовщого тому, але також і розбиття розділів на томи: з якого розділу починати 2-ий, 3-ій, ..., i-ий том. Щоб мати можливість після знаходження T_{BC} ефективно знайти ці значення, можна завести ще одну таблицю (таблицю виборів), у комірках котрої запам'ятовувати, яке са́ме значення k призвело до оптимального розв'язку (i,j)-підзадачі. А використати цю таблицю виборів для виведення результату можна так: у комірці з

L_j	300	300	500	300	300
j	1	2	3	4	5
1	300	600	1100	–	–
2	–	300	600	800	–
3	–	–	500	600	600

j	1	2	3	4	5
1	–	–	–	–	–
2	–	1	2	3	–
3	–	–	2	3	3

Мал. 2: Приклад заповнених таблиць динамічного програмування (до пункту 4.1.2); вхідні дані — 5 розділів довжинами 300, 300, 500, 300, 300 потрібно розмістити у 3-ьох томах.

номером (B, C) береться число k — номер останнього розділу в $(B-1)$-ому томі, потім комірці з номером $(B-1, C-k)$ — номер останнього розділу в $(B-2)$-ому, і так далі, до 1-ого тому включно. Цей прохід називають *зворотнім ходом*. Оскільки зворотній хід будує результати від останнього тому до 1-ого, а вивести їх потрібно вд 1-ого до останнього, їх або доведеться запам'ятовувати в якомусь допоміжному масиві, або ж «перевернути» порядок обчислень у формулі (1), так щоб у зворотньому ході отримати якраз потрібний порядок.

На мал. 2 наведено приклад заповнених таблиць динамічного програмування для вхідних даних з умови.

Відмінність між версіями B) і C) полягає у тому, що для версії B) допустимими за часом виконання мають виявитися також і деякі менш ефективні перебірні методи, тоді як для версії C) описаний алгоритм є чи не єдиним правильним і допустимим за часом виконання.

4.2 Версії D)–F)

Задача у версіях D)–F) має єдину, але дуже суттєву відмінність від версій A)–C): частини дозволяється розміщувати в довільному порядку. Таке «збільшення свободи вибору» призводить до того, що загальна кількість допустимих варіантів розміщень оповідань по томах при збільшенні B та / або C зростає із шаленою швидкістю.

На сучасному етапі розвитку науки про алгоритми невідомо, як можна правильно і ефективно розв'язати версії D)–F): ця задача належить до скандально відомого і практично дуже важливого класу так званих NP-*повних задач*. Більш того, на сучасному етапі розвитку науки про алгоритми не просто невідомо *як* їх розв'язувати, а невідомо чи взагалі *існує* єдиний алгоритм, котрий розв'язує їх правильно й ефективно.

У випадку версії D) наведемо алгоритм, придатний *виключно* для розбиття на *два* томи, котрий до того ж працює *лише* для досить невеликого значення $SumL$ і обов'язково цілих значень L_i, але при дотриманні усіх цих умов є відносно ефективним.

У випадку версії E) покажемо, як можна скоротити повний перебір, так щоб він працював із допусимою швидкістю.

У випадку версії F) проголосимо без доведення, що ніякі алгоритми не можуть давати гарантовано правильного результату за досить обмежений час; оскільки краще отримати допустимий наближений результат, ніж не отримати ніякого, наведемо приклад жадібного алгоритму, що працює дуже швидко і дає розв'язки, в середньому близькі до оптимальних.

4.2.1 Версія D)

Нам потрібно розбити оповідання на два томи так, щоб у найтовщому томі була якнайменша кількість сторінок. Якщо подивитися на цю задачу з точки зору тоншого тому, маємо рівнозначну задачу: «Які з оповідань треба взяти, щоб сума їхніх довжин була якомога більшою, але не перевищувала $\frac{SumL}{2}$?»

Подібно пункту 4.1.2, розглянемо серію однотипних задач: *«Якщо брати деякі з оповідань від 1-ого по j-те так, щоб витримати вимогу, що сума їхніх довжин не перевищує L, то якої максимально можливої суми довжин можна досягти?»*. Позначимо відповідь підзадачі (самý цю максимально можливу товщину, що не перевищує L) як $T_{(j,L)}$.

Якщо кожну таку задачу позначити як (j, L)-підзадачу, то шукана задача є $(C, \frac{SumL}{2})$-підзадачею.

Якщо $j = 1$, задача тривіальна: при $L < L_1$ маємо $T_{(1,L)} = 0$, а при $L \geq L_1$ маємо $T_{(1,L)} = L_1$ (нагадаємо, що за L_i ми позначаємо довжину i-ого оповідання).

Якщо $j \geq 2$, то потрібно перевірити два варіанти: j-те оповідання можна або включати до тому, або не включати. Звідси,

$$T_{(j,L)} = \max\left(T_{(j-1,L-L_j)} + L_j,\ T_{(j-1,L)}\right). \qquad (2)$$

Справді, якщо j-те оповідання включити до тому, то найбільша можлива допустима сума довжин $T_{(j-1,L-L_j)} + L_j$, а якщо не включати — $T_{(j-1,L)}$. Звичайно, при реалізації формули (2) потрібно подбати про те, щоб коректно обробляти ситуацію $L < L_i$ (в такому випадку $T_{(j,L)} = T_{(j-1,L)}$).

Реалізовувати формулу (2) потрібно теж або через memoized recursion, або через заповнення таблиці рядок за рядком. Для відновлення ж сукупності оповідань, котрі потрібно включити до складу тоншого тому в даному випадку доцільніше не заводити окрему таблицю виборів, а в процесі зворотнього ходу дивитися, яка з ситуацій має місце: чи $T_{(j,L)} = T_{(j-1,L-L_i)} + L_i$, чи $T_{(j,L)} = T_{(j-1,L)}$.

4.2.2 Версія E)

Описаний тут метод скорочення перебору — досить універсальний і дуже корисний на практиці. На жаль, ми не маємо можливості подати його тут *дуже* докладно і зрозуміло; усіх зацікавлених відсилаємо до літератури, ключові слова — *back-tracking* (він же *бектрекінг*, він же *алгоритм із поверненнями*) і *метод гілок та меж*.

Відзначимо також, що на олімпіадах *дуже* часто зустрічаються задачі, котрі *можна* правильно розв'язати перебірними методами, але це буде дуже *неефективно*. Тому, потрібно *вміти* програмувати перебір, але перш ніж його *застосувати* на практиці, потрібно переконатися, що: 1. нема ніякого досить очевидного більш ефективного методу; 2. час роботи перебору на вхідних даних потрібного розміру складатиме секунди, а не мільярди років.

Спершу опишемо, як можна реалізувати перебір *геть усіх можливих розбиттів* оповідань на томи.

Перебір полягає у тому, щоб вибрати один варіант, щось для нього перевірити, перейти до наступного, перевірити його, перейти до наступного, і так далі. Отже, побудова повного перебору містить дві суттєві складові: 1) визначити, *що са́ме* потрібно перевіряти для можливих варіантів; 2) побудувати такий *спосіб переходу* від поточного варіанту до наступного, який жодного допустимого варіанту не пропускає і не розглядає один і той самий варіант по багато разів.

Розглянемо спочатку 1-ий том.

Кожне з C оповідань можна або включати до нього, або не включати. Будемо відзначати це у масиві `v` розміром `1..C`: якщо `i`-те оповідання включене до 1-ого тому, то `v[i]=1`, а якщо не включене, то `v[i]=0`. Уявімо собі, що ми маємо справу з C-розрядним двійковим числом, кожен розряд котрого подається у окремому елементі масива. Маємо взаємно однозначні відповідності: з одного боку, комбінації ноликів і одиничок у масиві однозначно відповідають C-розрядним двійковим числам, а з іншого боку, ці самі комбінації однозначно відповідають можливим варіантам включення / не включення оповідання до 1-ого тома. Перебрати усі числа від 0 до $2^C - 1$ просто — почати від нуля і додавати по одиничці. Отже, будемо перебирати комбінації ноликів та одиничок саме в такому порядку.

Нехай ми визначили поточну комбінацію ноликів та одиничок, що описують включення / не включення оповідань до першого тому; із тих оповідань, що *не* включені до 1-ого тому, *якісь* можуть бути включені до 2-ого. Ми не знаємо,

які слід включати, а які ні — отже, знову потрібен перебір.[3] Нам потрібно перебирати варіанти *розбиття*, тобто *кожне* оповідання має бути включене *в якийсь один* том. Будемо працювати з тим самим масивом `v` розміром `1..C`, але трохи інакше трактувати значення його елементів. Значення 0 означає, що дане оповідання ми нікуди не включили, а значення i від 1 до $B-1$ — що розглядаємо варіант, де відповідне оповідання включене до i-ого тому. Включати до 2-ого тому треба лише якісь із оповідань, не включених до 1-ого. Отже, за вибраною системою позначень, пробуємо ставити / не ставити двійки *лише* там, де не поставили одиничок. Потім йдемо далі: вибравши поточний вміст 1-ого і 2-ого томів, перебираємо можливі варіанти для 3-ого, причому пробуємо включати / не включати туди лише ті частини, що не включені ні до 1-ого, ні до 2-ого; аналогічно для усіх томів по $(B-1)$-й включно; до останнього B-ого тому потрібно включити всі оповідання, не включені до жодного з попередніх. Проаналізувавши отримане розбиття, розглядаємо наступний варіант включення оповідань до $(B-1)$-ого тому; коли варіанти розміщення оповідань у $(B-1)$-ому томі вичерпаються, розглянемо новий варіант розміщення оповідань у $(B-2)$-ому, і почнемо розгляд усіх можливих розміщень в $(B-1)$-ому спочатку. Аналогічно, вичерпавши всі можливі варіанти розміщення оповідань у $(B-2)$-ому томі, розглядаємо новий варіант розміщення у $(B-3)$-ьому й починаємо спочатку перебір варіантів $(B-2)$-ого та $(B-1)$-ого. Перебір завершиться остаточно, коли вичерпаються всі можливі варіанти розміщень оповідань у 1-ому томі.

Ми тепер знаємо, як організувати перебір *геть усіх можливих* розбиттів оповідань на томи. Дописати туди перевірку, чи є дане розбиття кращим за усі попередні, досить легко — потрібно лише мати додаткову змінну `BestThick` для зберігання кількості сторінок найтовщого тому в найкращому з досі знайдених розбиттів, та масив `v_best`, щоб зберігати власне розбиття, на котрому досягається ця мінімальна знайдена товщина найбільшого тому. Такий розв'язок буде правильним, але дуже неоптимальним, і на вхідних даних потрібного розміру не буде завершувати роботу вчасно.

Основна і універсальна ідея оптимізацій перебору така: *дуже часто можна побачити, що дана гілка неперспективна, **до** того, як будуть побудовані конкретні розбиття*. Нехай ми вже маємо якийсь допустимий розв'язок, котрий найкращий з уже знайдених (але ж не знаємо, чи є він найкращим серед усіх і тому мусимо перебирати далі). І нехай ми перейшли до розгляду нового варіанту вмісту 1-ого тома, і бачимо, що він товщий за цей найкращий з раніше знайдених розв'язків. У такому разі, *нема ніякої потреби витрачати час на перебір можливих вмістів 2-ого, 3-ого, ... томів, бо 1-ий том **уже затовстий**, і тому серед усіх цих варіантів не може бути розв'язку, кращого за вже знайдений*. Абсолютно те саме потрібно робити і для усіх томів від 2-ого по $(B-1)$-ий. Можна відтинати також варіанти, котрі включають до початкових томів настільки *мало* оповідань, що навіть якби вдалося усі наступні оповідання розкидати по усім наступним томам абсолютно рівномірно, то товщина цих томів все одно виявиться більшою за вже знайдену оптимальну.

Оцінити, наскільки сильно можна виграти від таких оптимізацій, дуже складно. По-перше, дієвість цих оптимізацій дуже сильно залежать від конкретного вигляду вхідних даних. По-друге, їх потрібно ще зуміти запрограмувати, не втративши по дорозі самої суті. Але наближено можна сказати, що в середньому виграш від таких оптимізацій на таких розмірах вхідних даних складає кілька десятків разів.

4.2.3 Версія F)

На сучасному етапі розвитку науки про алгоритми невідомо, як розв'язувати дану задачу на вхідних даних такого розміру правильно і ефективно. Причому, усі відомі правильні алгоритми *експоненційні*[4], а це означає *таку* неефективність, що її не можна і сподіватися подолати написанням програми не на Паскалі, а на Асемблері, або ж виконанням не на AMD-486, а на Pentium-III. Припустимо без точних підрахунків, що в другому випадку час виконання буде в 1000 разів менший, ніж у першому. 1000 разів — це дуже багато у переважній більшості практичних застосувань, але це — ніщо у даному випадку. Якщо має місце, що «написавши на Паскалі і запустивши на 80486DX доведеться чекати результату кілька мільярдів років», то з того, що «написавши на Асемблері і запустивши на Pentium-III — *лише* кілька мільйонів років» користі ніякої. А програма, побудована за засадах алгоритму версії E), на вхідних даних розміру $B=50$, $C=200$ мусить працювати *ще довше*.

Тобто, ми не знаємо, як розв'язувати задачу, а розв'язувати її все-таки треба... В такій ситуації важко придумати щось краще за вимушений компроміс: знаходити швидко якийсь допустимий розв'язок, не дуже далекий від оптимального.

Отже, *наведений далі алгоритм — єдиний з наведених у даному поясненні — не є гарантовано правильним.*

Покладемо, що всі томи спочатку порожні, і будемо «заповнювати» їх, включаючи до них оповідання.

Будемо намагатися розкида́ти оповідання по томах більш-менш *рівномірно*, тобто щоб розміри усіх томів завжди були приблизно однаковими. Щоб добитися рівномірного заповнення, логічно включати чергове оповідання до *того* з C томів, що *на даний момент має найменшу товщину*. Шоправда, так може виникнути ситуація, що майже всі оповідання вже розкидані по томах цілком рівномірно, і тут потрібно кудись включити дуже велике оповідання. Як наслідок, том з цим оповіданням майже напевно виявиться значно товщим за інші томи. Але цієї проблеми можна у значній мірі уникнути, якщо попередньо відсортувати оповідання за розміром, і розкида́ти по томах, починаючи з найбільших оповідань.

Дуже наближені експериментальні оцінки показали, що приблизно у 30–40% випадків такий алгоритм знаходить оптимальне розбиття, приблизно у 40–50% випадків товщина найтовщого тому, знайдена цим алгоритмом, перевищує мінімально можливу не більше ніж на 10%, і лише у приблизно 10–20% випадків отриманий результат відрізняється від оптимального більш ніж на 10%.

4.2.4 Зауваження

Звичайно ж, алгоритм для версії F) може бути застосованим також і до задач версій D) та E). Але так варто чинити в ситуації, коли треба якомога швидше отримати хоч що-небудь, а не тоді, коли бажано отримати все-таки оптимальний результат.

Звичайно, алгоритм для версії E) допускає подальші оптимізації. Одна з них — починати перебір не з тривіального припущення «включити всі оповідання до останнього тому», а з розв'язку, знайденого алгоритмом версії F): слід сподіватися, що таке покращення початкового наближення суттєво посилить відтинання і звузить таким чином область перебору.

Неважко також помітити, що якщо усі оповідання, що були у першому томі, включити до другого, а усі, що раніше були в другому — до першого, то по суті нічого не зміниться, і краще б не витрачати час на аналіз *обох* цих розбиттів. Уникнувши таких зайвих переглядів, можна сподіватися прискорити виконання програми у (майже) $B!$ разів.

Один із ефективних способів реалізувати щойно описану оптимізацію такий. Нехай перше оповідання входить обов'язково до складу першого тому; коли ми вибрали поточний варіант входжень оповідань у томи з 1-ого по $(i-1)$-ий, і починаємо вибирати сукупність оповідань, що входять до i-ого тому, нехай оповідання з мінімальним номером, ще не включене ні в який том, входить обов'язково са́ме до i-ого тому. Наприклад, із двох розбиттів `1 3  2 4` і `2 4  1 3`, котрі по суті співпадають, друге не буде розглядатися,[5] бо в

[3] Це трохи схоже (але набагато складніше) на структуру `for i:=1 to N do for j:=1 to i do ...`: спочатку беремо перше значення `i`, й перебираємо у внутрішньому циклі потрібні значення `j`, потім беремо наступне значення `i` й перебираємо можливі значення `j` спочатку, і т.д.; причому, діапазон перебору по `j` залежить від того, яке са́ме `i` зараз розглядаємо.

[4] див. примітку 6 на стор. 7

[5] Наголошуємо: не «буде відкинуте», а «взагалі не буде розглядатися»

ньому перше оповідання не входить до складу першого тому.

Після реалізації наведених та інших аналогічних оптимізацій, програму для версії E) можна успішно використовувати також і для задач версії D); втім, *якби* час виконання був дуже критичним, а об'єм пам'яті — мало критичним, то для задач версії D) кращим лишався б усе-таки псевдополіноміальний[6] алгоритм з пункту 4.2.1.

Істотно оптимізована програма для версії E) може розв'язувати за допустимий час задачі розміром приблизно до $B \le 7$, $C \le 20$ (і трохи «увійти в зону версії F)»), але, як і слід чекати від експоненційного алгоритму, при подальшому збільшенні B або C час роботи зростає так само шалено...

Що стосується підходу «розв'язати хоча б наближено», то на практиці буває корисним і ще один спосіб: написати перебір так, щоб щоразу, знайшовши результат кращий за знайдені раніше, витирати з вихідного файлу старі результати і записувати новий; коли користувачу набридне чекати далі на завершення роботи програми й він її обірве, у вихідному файлі вже буде якесь наближення до оптимального розв'язку; причому, чим довше працювала програма, тим ближчий повинен бути результат до оптимального.

Не зважаючи на корисність такого підходу на практиці, ми *категорично не рекомендуємо* використовувати його *на олімпіадах*. Фактично на Всеукраїнських учнівських олімпіадах з інформатики діє (чомусь явно не обумовлене в пам'ятці учасника...) правило, що якщо програма не завершила роботу вчасно, то вихідний файл взагалі не аналізується! Відхилення ж від цього правила на олімпіадах нижчого рівня в принципі можливі, але малоймовірні.

[6] Алгоритм називають *поліноміальним*, якщо час його виконання може бути виражений як многочлен (поліном) від *розміру* вхідних даних. Так, *усі* алгоритми, наведені в даному поясненні, крім алгоритмів для версій D) і E) даної задачі, є поліноміальними: для одного проходу по послідовності дужок (пункти 2.1 та 2.2) потрібно виконати близько N дій, де N — кількість цих дужок. Алгоритм з пункту 3.2 (названий у пункті 3.5 правильним, але дуже неоптимальним) теж поліноміальний — час його виконання складає порядку N^2, де N — кількість елементів послідовності. Алгоритм для версій B) та C) даної задачі теж поліноміальний, загальна кількість дій складає близько BC^2. А перебірний алгоритм версії E) даної задачі *експоненційний*, бо у найгіршому випадку час його виконання може сягати близько B^C: кількість оповідань C є *показником* степеню. Для алгоритма ж версії D) загальна кількість дій близька до $SumL \cdot C$. Жоден з розмірів вхідних даних не опиняється у показнику степеня, тому алгоритм не є експоненційним; але у цю оцінку входять не лише *розміри* вхідних даних (B та C), але також і $SumL$ — сума *значень* цих вхідних даних. Тому його і називають *псевдо*поліноміальним.