КОМП'ЮТЕРНІ НАУКИ

УДК 004.432;004.047 https://doi.org/10.31713/vt1202317

Шпортько О. В., к.т.н., доцент (ПВНЗ «Міжнародний економіко-гуманітарний університет імені академіка Степана Дем'янчука», м. Рівне, ITShportko@ukr.net)**, Мушин М. М., студентка** (ПВНЗ «Міжнародний економіко-гуманітарний університет імені академіка Степана Дем'янчука», м. Рівне, m.mushyn@ukr.net)**, Бомба А. Я., д.т.н., професор** (Національний університет водного господарства та природокористування, м. Рівне, abomba@ukr.net)

ПРО ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД В ПРОЦЕСІ ПРОГРАМУВАННЯ АЛГОРИТМІВ РОЗВ'ЯЗУВАННЯ ЗАДАЧ КОМБІНАТОРНОЇ ОПТИМІЗАЦІЇ

В роботі обґрунтовані переваги від застосування об'єктно-орієнтованого підходу при розв'язуванні типових задач комбінаторної оптимізації. Описано метод поступового формування множини значень цільової функції як альтернативного методам пошуку з поверненнями та врахування змін. Співставлено механізми дії алгоритмів, які застосовують ці методи для розв'язування задач комбінаторної оптимізації. Наведено фрагменти програм, які реалізують алгоритми мовою програмування C# та проаналізовано результати їх тестування у віддаленому обчислювальному середовищі. За результатами тестування показано, що реалізація методу поступового формування множини значень цільової функції кардинально зменшує час виконання програм, що вказує на доцільність його застосування при розв'язуванні типових задач комбінаторної оптимізації.

***Ключові слова*: об'єктно-орієнтоване програмування; комбінаторна оптимізація; метод поступового формування множини значень цільової функції.**

Як відомо, комбінаторика вирішує задачі вибору та розташування елементів множини відповідно до заданих правил. Вона застосовується, наприклад у криптографії, у створенні штучних нейромереж, у програмуванні чи при вирішенні задач з інших сфер.

Хоча формули комбінаторики можуть бути простими для алгоритмізації, але їх використання для вхідних даних великих обсягів може суттєво сповільнювати роботу окремих програм та в цілому обчислювальних систем. Саме тому розвиток існуючих і створення нових методів для прискорення вирішення комбінаторних задач є на сьогодні **важливим науковим завданням**, яке розв'язується в межах комбінаторної оптимізації [1].

Комбінаторна оптимізація розглядає завдання, в яких множина допустимих розв'язків дискретна або може бути зведена до дискретної. Метою таких завдань є відшукання деякого оптимального елемента з дискретної множини, для визначення якого за відведений час недостатньо загальновідомих способів пошуку з використанням повного перебору [2, С. 56–61].

Традиційно для прискорення повного перебору використовують **метод врахування змін**, який розраховує наступний варіант розв'язку не з самого початку, а лише враховує зміни відносно попереднього варіанта розв'язку (наприклад зі зміною часу, як в динамічному програмуванні [3, С. 392–447]).

Але найвідомішим методом для розв'язування задач комбінаторної оптимізації є **пошук з поверненнями** [4]. При його використанні часткове рішення розширюється і надалі аналізуються отримані неповні «кандидати на розв'язок». Якщо черговий «кандидат» недопустимий, то метод переходить до іншого «кандидата» чи повертається до іншого часткового рішення або продовжує пошук іншими шляхами. Згідно з цим методом, усі кроки пошуку оптимального рішення фіксуються, щоб у разі змін, які не підходять під обмеження розв'язку задачі, можна було повернутись до прийнятного «кандидата» [4]. Реалізації цього методу, як правило, швидші за повний перебір, адже в процесі їх виконання недопустимі «кандидати» відкидаються і не доповнюються до повного розв'язку.

На сьогодні при програмуванні розв'язування задач комбінаторної оптимізації основна увага приділяється відсіканню недопустимих «кандидатів» і недостатньо аналізується, на нашу думку, обчислювальна складність алгоритму. А даремно, адже саме показники обчислювальної складності [2, С. 13–27] дають уявлення про час виконання алгоритмів, дозволяють зрозуміти, чому один алгоритм буде працювати значно швидше від іншого, привчають майбутніх програмістів звертати увагу не лише на правильність розв'язку поставленої задачі, а й на його ефективність, не лише на

час виконання програми, а й на обсяги використовуваної пам'яті. Саме аналізуючи обчислювальну складність, ми визначатимемо у цій статті доцільність застосування запропонованих алгоритмів.

Загалом ідеї застосування множини значень використовуються в програмуванні вже тривалий час. Наприклад, для сортування цілочисельних масивів з невеликим діапазоном значень елементів за зростанням доцільно не порівнювати елементи між собою, а всього лише підрахувати частоти окремих значень і повторити в результуючому масиві кожне значення від найменшого до найбільшого стільки разів, скільки воно зустрічалося у вхідному масиві [3, С. 223–226; 5]. Зрозуміло, що це доцільно робити, коли діапазон значень співмірний з розміром масиву. Ми ж застосуємо аналіз множини значень для розв'язку задач комбінаторної оптимізації.

Ми пропонуємо застосовувати об'єктно-орієнтований підхід [6] в процесі програмування розв'язування типових задач комбінаторної оптимізації таким чином:

1. Етапи та методи розв'язування кожної типової задачі описувати в окремому класі. Екземпляри кожного з таких класів дають змогу порівнювати між собою розв'язки кожної задачі при різних початкових даних, ефективно розпаралелювати обчислення, порівнювати тривалості розв'язків кожної задачі різними методами.

2. В усіх класах типових задач комбінаторної оптимізації давати однакові назви тим самим методам розв'язання. Це дасть змогу підтримувати цим класам спільний інтерфейс і за його допомогою циклічно розв'язувати різні задачі одним і тим самим методом.

3. Однакові методи для введення початкових даних та виведення результатів для різних задач реалізовувати у спільних батьківських класах, що сприятиме зменшенню повторюваних описів.

Зрозуміло, що тестувати розроблені програми можна на локальних робочих станціях, але при цьому слід враховувати особливості обчислювальних середовищ і можливості пристосування програм до відомих тестових випадків. Тому ми вважаємо доцільним тестування розроблених програм у віддалених обчислювальних середовищах. Таке тестування – це спосіб розвитку логічного мислення та навиків оптимізації при вирішення нетипових задач. Одним із сайтів, який надає доступ до завдань та віддаленого обчислювального середовища, є https://www.eolymp.com. Ця робота

написана з використанням його ресурсів. Переваги eolymp наступні:

- значна кількість задач з різних розділів;
- групування задач за різними рівнями складності;
- можливість порівняння свої результатів з результатами інших користувачів;
- підтримка понад 15 мов програмування для подання розв'язків;
- наявність навчальних матеріалів;
- можливості порівняння затрат часу на виконання та обсягів використання оперативної пам'яті для окремого тесту при застосуванні різних підходів до розв'язування кожної задачі.

Продемонструємо застосування об'єктно-орієнтованого підходу та різних методів комбінаторної оптимізації на прикладі спрощеної *проблеми про пакування рюкзака* [7], в якій потрібно максимально заповнити рюкзак речами наперед заданого об'єму. Для наглядності конкретизуємо цю проблему на прикладі задачі «CD» (№ 1266) із сайту https://www.eolymp.com. Умову цієї задачі скорочено сформулюємо так:

У Вас є чиста магнітофонна стрічка з тривалістю звучання N хвилин. Серед S треків з заданими тривалостями звучання вам потрібно вибрати треки для запису на магнітофону стрічку таким чином, щоб невикористовуване на ній місце було мінімальним. Кожен трек може бути записаний на стрічку не більше одного разу. Визначити максимально можливу довжину запису треків на стрічку.

Обмеження на вхідні дані: кількість треків S не перевищує 100; довжина кожного треку є невід'ємним цілим числом; N також ціле (0 ≤ N ≤ 200). Кожен тест має виконуватися програмою не більше 1 с.

Для розв'язування поставленої задачі ми розробили клас з наступною структурою:

```
static public class ProblemCombiOpt
 {static int N, S, maxSum;
  static int[] track;
  static public bool ReadProblem()...
  static public void BruteForce()...
  static public void ChangeAccounting()...
  static public void SearchWithReturns()...
  static public void FormingSetValues()...
  static public void WriteResult()... },
```

де перша процедура відповідає за введення даних чергового тесту, остання – за вивід результатів, а всі інші процедури реалізують різні

методи вирішення поставленої задачі. Тоді розв'язування групи тестів, наприклад, методом пошуку з поверненнями можна реалізувати так:

```
while (ProblemCombiOpt.ReadProblem())
 {ProblemCombiOpt.SearchWithReturns();
  ProblemCombiOpt.WriteResult(); }.
```

Наведемо тепер реалізації алгоритмів різних методів розв'язування поставленої задачі та порівняємо їх ефективність.

Використання методу повного перебору. Щоб знайти максимально можливу довжину запису треків на стрічку, яка не перевищує *N*, потрібно бути готовим розглянути всі можливі варіанти їх входження, адже кращою може виявитися будь-яка комбінація треків. Достроково завершити розгляд наступних варіантів можна лише тоді, коли буде знайдено варіант, при якому довжина магнітофонної стрічки буде зайнята повністю (рівна *N*).

За умовою задачі, кількість треків рівна *S*. Кожен з них може бути лише в двох станах – врахований до суми поточного варіанта або не врахований. Отже, **кожен з варіантів входження треків можна записати як *S*-бітове двійкове число**, а всього варіантів буде 2^S. **Запишемо довжини всіх треків справа наліво**, аналогічно розміщенню бітів двійкового числа – від молодших до старших. Для перебору всіх можливих варіантів нам знадобляться усі *S*-бітові двійкові числа, крім 0, адже це значення мінімально можливої суми за замовчуванням. З метою знаходження суми довжин треків чергового варіанта будемо переводити номер варіанта у двійкову систему числення і аналізувати окремі біти. Тому обчислювальна складність алгоритму становитиме $2^S \times S$. В чергову суму будемо додавати лише довжини треків, для яких відповідні біти в номері варіанта рівні 1. Для розв'язку задачі серед сум варіантів, не більших за *N*, будемо обирати максимальну.

Врахуємо також, що якщо додавання довжини чергового треку до суми поточного варіанта робить її більшою за довжину стрічки, то це порушує обмеження задачі (оскільки довжини треків невід'ємні), і тому програма може достроково переходити до наступного варіанта і формування нової суми.

Для прикладу простежимо механізм формування максимально допустимої суми треків для стрічки довжини 10 ($N = 10$) при чотирьох треках ($S = 4$) з відповідними довжинами 2, 4, 8, 4 (таблиця).

Таблиця

Використання двійкового запису номерів варіантів для генерування комбінацій треків для $N = 10$, $S = 4$ та довжин треків {2, 4, 8, 4}

Номер варіанта в десятковій системі	**Номер варіанта в двійковій системі**	**Враховані треки**	**Сума поточного варіанта входжень треків**	**Відома наразі максимально допустима сума**
1	0001	2	2	2
2	0010	4	4	4
3	0011	4, 2	6	6
4	0100	8	8	8
5	0101	8, 2	10	10

Бачимо, наприклад, що встановлення першого біта справа в двійковому записі номера варіанта призводить до входження в поточну суму довжини першого треку. У розглянутому випадку пошук максимальної суми завершується до перебору всіх варіантів входжень треків ($2^4 = 16$), оскільки знайдена максимально допустима сума рівна довжині стрічки. Якби цього не сталося, програма продовжила б роботу до перебору всіх варіантів. Максимальне заповнення (10) знайдено за п'ять варіантів перебору.

Наведемо текст процедури для розв'язання поставленої задачі методом повного перебору мовою програмування C#, оскільки на сьогодні вона є однією з основних мов програмування прикладних додатків:

```
static public void BruteForce()
{maxSum = 0; int i, sum;
 BigInteger variantsNumber = (BigInteger)1 << S, remainder, j;
•/*1<<S те саме, що 2^S – кількість усіх можливих варіантів вибору.
Переберемо всі S-бітові числа, де і-тий біт буде вказувати, враховується і-тий трек в суму
(1) чи ні (0). Це дозволить отримати кожен з можливих варіантів вибору входжень їхніх
довжин */
 //обчислення суми треків для кожного варіанта
 for (j = 1; j < variantsNumber; j++)
  {sum = 0;
   remainder = j; //Щоб не впливати на лічильник варіантів
   for (i = 0; i < S; i++) /* Варіант вибору треків інтерпретуємо як двійкове S-бітове число,
довжина якого відповідає кількості треків */
    {if (remainder % 2 == 1) // Якщо і-ий трек до суми входить
     {sum += track[i];
```

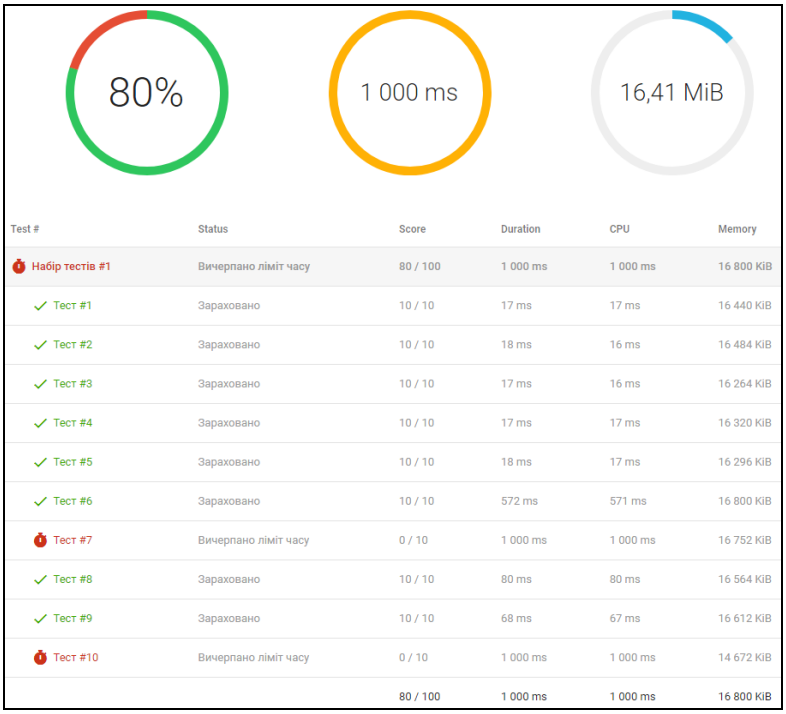
```
    if (sum > N) break; } // Перевищення недопустиме
➢  remainder /= 2; } //Переходимо до наступних треків
  if (sum <= N && sum > maxSum) // Якщо знайдена сума не перевищує
  //загальну довжину стрічки та більша за кращу з попередніх
   {maxSum = sum;
    if (maxSum == N) break; }} // Вже знайшли оптимальний варіант
•}
```

Результати тестування цієї програми на eolymp.com подані на рис. 1. Бачимо, що наведена програма проходить вісім тестів, але вичерпує ліміт часу при виконанні тестів № 7 та № 10.

Оптимізація алгоритму повного перебору. Для прискорення наведеної програми врахуємо, що цикл для переведення номеру варіанта у біти двійкового запису виконується максимум S разів для кожного варіанта. Це призводить до зайвих операцій, оскільки не всі номери варіантів є s-значними двійковими числами. Взагалі, переведення числа з однієї системи числення в іншу завершується, коли частка від ділення на основу системи числення, в яку виконують переведення, стає рівною нулю. Для нашого алгоритму це означатиме, що для чергового варіанта немає більше треків зліва, які входять в поточну суму. Для реалізації цієї ідеї доповнимо код першого варіанта після рядка, позначеного маркером ➢, такою перевіркою:

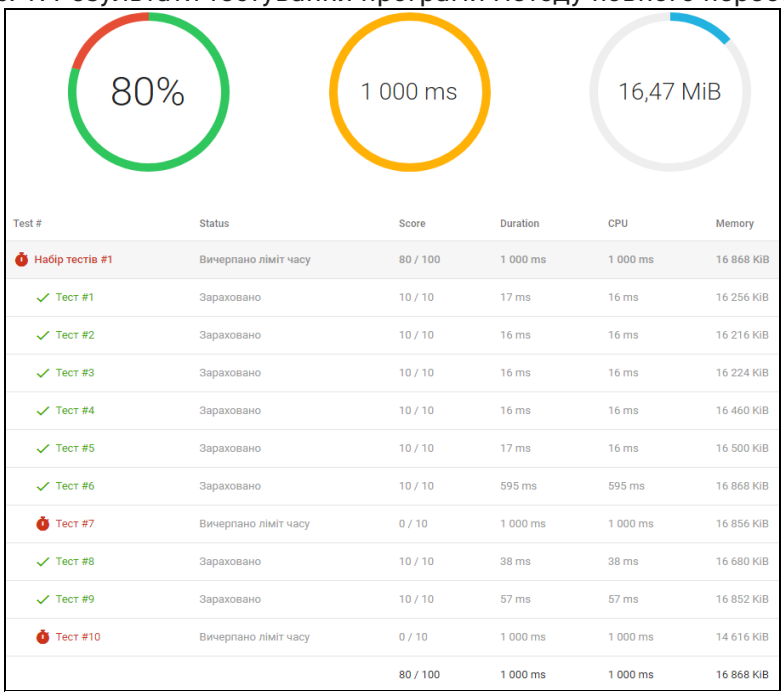
```
if(remainder == 0) break; // Більше немає треків, які входять в суму
```

Ця перевірка суттєво впливає на швидкість виконання тестів з великим S при розгляді варіантів включення треків з невеликим номером (рис. 2). Бачимо, що виконання двох тестів все ще вичерпує ліміт часу. Обчислювальна складність алгоритму тепер становить $2^{S-1}\times S$, тобто зменшилася вдвічі, але реально час виконання програми суттєво не зменшився, оскільки переривання пошуку з поверненнями не впливають на збільшення проміжних сум. Якби при виконанні наведеної перевірки вдалося ще й оминути окремі варіанти розв'язків, то це було б класичною реалізацією пошуку з поверненнями.



Test #	Status	Score	Duration	CPU	Memory
Набір тестів #1	Вичерпано ліміт часу	80 / 100	1 000 ms	1 000 ms	16 800 KiB
Тест #1	Зараховано	10 / 10	17 ms	17 ms	16 440 KiB
Тест #2	Зараховано	10 / 10	18 ms	16 ms	16 484 KiB
Тест #3	Зараховано	10 / 10	17 ms	16 ms	16 264 KiB
Тест #4	Зараховано	10 / 10	17 ms	17 ms	16 320 KiB
Тест #5	Зараховано	10 / 10	18 ms	17 ms	16 296 KiB
Тест #6	Зараховано	10 / 10	572 ms	571 ms	16 800 KiB
Тест #7	Вичерпано ліміт часу	0 / 10	1 000 ms	1 000 ms	16 752 KiB
Тест #8	Зараховано	10 / 10	80 ms	80 ms	16 564 KiB
Тест #9	Зараховано	10 / 10	68 ms	67 ms	16 612 KiB
Тест #10	Вичерпано ліміт часу	0 / 10	1 000 ms	1 000 ms	14 672 KiB
		80 / 100	1 000 ms	1 000 ms	16 800 KiB

Рис. 1. Результати тестування програми методу повного перебору



Test #	Status	Score	Duration	CPU	Memory
Набір тестів #1	Вичерпано ліміт часу	80 / 100	1 000 ms	1 000 ms	16 868 KiB
Тест #1	Зараховано	10 / 10	17 ms	16 ms	16 256 KiB
Тест #2	Зараховано	10 / 10	16 ms	16 ms	16 216 KiB
Тест #3	Зараховано	10 / 10	16 ms	16 ms	16 224 KiB
Тест #4	Зараховано	10 / 10	16 ms	16 ms	16 460 KiB
Тест #5	Зараховано	10 / 10	17 ms	16 ms	16 500 KiB
Тест #6	Зараховано	10 / 10	595 ms	595 ms	16 868 KiB
Тест #7	Вичерпано ліміт часу	0 / 10	1 000 ms	1 000 ms	16 856 KiB
Тест #8	Зараховано	10 / 10	38 ms	38 ms	16 680 KiB
Тест #9	Зараховано	10 / 10	57 ms	57 ms	16 852 KiB
Тест #10	Вичерпано ліміт часу	0 / 10	1 000 ms	1 000 ms	14 616 KiB
		80 / 100	1 000 ms	1 000 ms	16 868 KiB

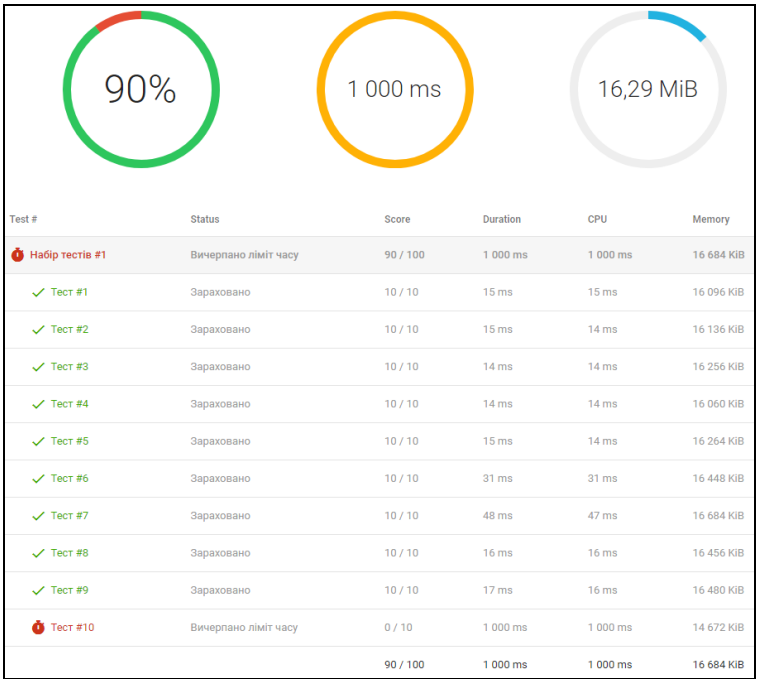
Рис. 2. Результати тестування програми методу повного перебору з урахуванням довжини номера варіанта в двійковій системі числення

Застосування методу врахування змін. Реалізуємо метод урахування змін при обчисленні суми довжин треків чергового варіанта відносно суми попереднього варіанта. Поточна сума має зменшитися відносно попередньої на довжину тих треків, в двійковому представленні, номер яких замість 1 став 0. І має збільшитися на довжину першого з треків, де замість 0 став 1. Порядок вибору треків, як і у попередній програмі, визначимо двійковими *S*-бітовими числами, але значення окремих бітів не будемо обчислювати щоразу, а збережемо в масиві *occurrence* з *S*+1 елемента. Це дасть змогу уникнути надлишкових ділень на два і розв'язувати задачу для будь-якого невід'ємного *S*. Перебір будемо завершувати, коли в додатковому елементі цього масиву з індексом *S* з'явиться 1, тобто будуть перебрані всі *S*-бітові двійкові числа. Для реалізації цього алгоритму замінимо в коді попередньої програми рядки між маркерами • наступним кодом:

```
byte[] occurrence = new byte[S+1]; /* і-тий елемент масиву вказує, враховується (1) чи ні
  (0) і-тий трек в суму. Використання цього масиву замість двійкового запису номера
  варіанта дає змогу опрацьовувати довільну кількість треків. Генерація наступного
  варіанта вибору треків по масиву occurrence відповідає принципу збільшення числа на 1
  в двійковій системі числення: додавання одиниці перетворює всі суміжні одиниці справа
  наліво в нулі, а перший справа нуль перетворює в одиницю. Відповідно, сума довжин
  треків відносно попередньої суми зменшиться на довжину треку там, де в масиві
  occurrence замість 1 став 0 і збільшиться на довжину треку там, де в масиві occurrence
  замість 0 став 1. Після створення масиву всі його елементи рівні 0, тобто перший
  варіантів вибору треків – той, при якому жоден з треків не враховується. Останній
  варіант вибору треків відповідає масиву occurrence, в якому всі елементи рівні 1, тобто
  всі треки враховуються. Завершення перебору варіантів вибору треків виконується при
  встановленні 1 в додатковій зліва позиції масиву (переповненні номера варіанта)*/
while (occurrence[S] == 0) //Поки додаткова позиція масиву не заповнена
 {j = 0; //Індекс треку, який змінюється в сумі
  while(occurrence[j] != 0)
   {occurrence[j] = 0; //Трек вже не враховується,
    //тому загальна тривалість зменшується на довжину цього треку
     sum -= track[j++]; }
  occurrence[j] = 1; //Дійшли до треку, який вже враховується,
  // тому загальна тривалість збільшується на довжину цього треку
  if (j < S) sum += track[j];
  if(sum <= N && sum > maxSum) // Знайшли кращий варіант вибору треків
   {maxSum = sum;
    if (maxSum == N) break; }}
```

Такі зміни програми прискорили її виконання для всіх тестів, але останній тест все ще не вкладається в час (рис. 3). Фактично, цей фрагмент програми в сумі треків чергового варіанта їх вибору лише враховує зміни відносно попереднього варіанта, а не перераховує цю

суму повністю, але обчислювальна складність алгоритму залишається $2^S \times S$.



Test #	Status	Score	Duration	CPU	Memory
Набір тестів #1	Вичерпано ліміт часу	90 / 100	1 000 ms	1 000 ms	16 684 KiB
Тест #1	Зараховано	10 / 10	15 ms	15 ms	16 096 KiB
Тест #2	Зараховано	10 / 10	15 ms	14 ms	16 136 KiB
Тест #3	Зараховано	10 / 10	14 ms	14 ms	16 256 KiB
Тест #4	Зараховано	10 / 10	14 ms	14 ms	16 060 KiB
Тест #5	Зараховано	10 / 10	15 ms	14 ms	16 264 KiB
Тест #6	Зараховано	10 / 10	31 ms	31 ms	16 448 KiB
Тест #7	Зараховано	10 / 10	48 ms	47 ms	16 684 KiB
Тест #8	Зараховано	10 / 10	16 ms	16 ms	16 456 KiB
Тест #9	Зараховано	10 / 10	17 ms	16 ms	16 480 KiB
Тест #10	Вичерпано ліміт часу	0 / 10	1 000 ms	1 000 ms	14 672 KiB
		90 / 100	1 000 ms	1 000 ms	16 684 KiB

Рис. 3. Результати тестування програми з реалізацією методу урахуванням змін

Для реалізації методу пошуку з поверненнями використаємо основне обмеження розв'язку задачі – сума довжин всіх треків не повинна перевищувати *N*. Відповідно, для чергового варіанта вибору треків підраховуватимемо суму треків **зліва направо** і як тільки ця сума перевищить *N* – пропустимо всі варіанти вибору з комбінацій треків, розміщених правіше, адже такі варіанти будуть недопустимими, тобто ми пропускаємо недопустимі варіанти і відразу переходимо до наступних можливих «кандидатів» на розв'язок.

Для реалізації алгоритму цього методу, як і в попередньому підрозділі, замінимо в коді наведеної програми рядки між маркерами • таким фрагментом:

```
 while(occurrence[S] == 0) //Поки додаткова позиція не заповнена
  {//Переходимо до наступного номера варіанта вибору треків
   j = 0;
   while(occurrence[j] != 0) occurrence[j++] = 0;
   occurrence[j] = 1;
   //Підраховуємо суму зліва направо, доки вона допустима
```

```
sum=0; j=S-1;
while (sum<=N && j>=0)
 {if (occurrence[j]==1) sum+=track[j];
  j--; }
if(sum <= N && sum > maxSum) //Знайшли кращий варіант вибору треків
 {maxSum = sum;
  if (maxSum == N) break; }
if (sum > N) //Досягнули обмеження суми треків –
 //пропускаємо комбінації з треків, розміщених правіше
  for (i=0;i<=j;i++) occurrence[j]=1; }          .
```

Така реалізація методу пошуку з поверненнями збільшує час проходження успішно пройдених тестів в середньому на 20% відносно тривалостей тестування, наведених на рис. 3, оскільки не враховує зміни відносно попереднього варіанта, а суму треків щоразу підраховує спочатку. При цьому останній тест не вкладається у відведений час. Цей алгоритм хоча й пропускає окремі недопустимі варіанти розв'язків, але має ту саму обчислювальну складність $2^S \times S$.

Опис та аналіз результатів застосування методу поступового формування множини значень цільової функції. Задамося питанням: як змінюється сукупність можливих сум довжин треків після розгляду чергового треку? Іншими словами: як отримати множину можливих (допустимих після сумування) довжин треків E_i, знаючи допустимі довжини сум попередніх треків E_{i-1} та довжину чергового треку $trek_i$ (i вказує на те, що розглядаються треки від початкового до i-го включно)? Якщо черговий трек $trek_i$ не враховується, то множина допустимих сум довжин треків залишаться такою самою, тобто $E'_i=E_{i-1}$. Коли ж черговий трек враховується, то з'являться нові допустимі суми довжин, утворені збільшенням попередніх сум довжин на довжину чергового треку. Тобто, $E''_i=\{e''_{i,j} : e_{i-1,j}+trek_i\}$. Очевидно, що

$$E_i=E'_i \cup E''_i. \qquad (1)$$

Якщо жоден трек не враховується, то сума довжин треків рівна 0. Коли ж розглянути початковий трек, то допустимими є суми довжин 0 (початковий трек не враховується) та $trek_0$ (якщо початковий трек враховується). Тобто $E_0=\{0; trek_0\}$, а наступні множини допустимих сум треків обчислюються згідно з (1).

Наприклад, для вищезгаданої послідовності довжин треків 2, 4, 8, 4 почергово отримаємо такі множини допустимих сум:

$E_0=\{0; 2\}$ (нульовий трек враховується або не враховується).

$E'_1=\{0; 2\}$ (перший трек не враховується), $E''_1=\{4; 6\}$ (перший трек враховується), і, згідно з (1) $E_1=E'_1 \cup E''_1=\{0; 2; 4; 6\}$ (перший трек враховується або не враховується).

E'_2={0; 2; 4; 6} (другий трек не враховується), E''_2={8; 10; 12; 14} (другий трек враховується), $E_2=E'_2 \cup E''_2$={0; 2; 4; 6; 8; 10; 12; 14}.

E'_3={0; 2; 4; 6; 8; 10; 12; 14}. (третій трек не враховується), E''_3={4; 6; 8; 10; 12; 14; 16; 18} (третій трек враховується), $E_3=E'_3 \cup E''_3$={0; 2; 4; 6; 8; 10; 12; 14; 16; 18}.

Оскільки допустимі суми щоразу зростають, то обчислювати ті з них, які більші *N,* немає сенсу (в наведеному прикладі допустимі суми понад 10 можна було б ігнорувати). Серед залишених допустимих сум потрібно вибрати максимальну після розгляду всіх треків, або рівну *N*, якщо така сума трапилася раніше (в розглянутому випадку допустима сума 10 з'являється вже після розгляду третього треку). На прикладі формування третьої множини допустимих сум бачимо також, що кількість елементів множини не обов'язково щоразу зростає вдвічі, адже можливе дублювання елементів в *E'* та *E''.*

Реалізуємо тепер програмно варіант поступового формування множин допустимих сум. Множина допустимих сум E_i після розгляду чергового елемента залежить лише від множини E_{i-1}. Тому не потрібно зберігати всі множини – достатньо ітеративно, маючи попередню множину, формувати наступну множину, а для наступної ітерації присвоїти попередній множині наступну, а наступній – попередню. Цікаво, що при цьому наступну множину можна не очищати – допустимі значення будь-якої множини завжди будуть допустимими для наступних допустимих множин. Для зберігання попередньої та наступної множин допустимих сум на кожній ітерації використаємо масиви, де значення 1 в кожному елементі буде вказувати на входження його в множину допустимих значень, а 0 – на відсутність його в цій множині. З метою швидшого переприсвоєння масивів будемо не переприсвоювати їх елементи, а переприсвоювати **посилання** на масиви. Враховуючи ці міркування, реалізація методу поступового формування множин допустимих значень для досліджуваної задачі може бути такою:

```
byte[] nextSum = new byte[N + 1], prevSum = new byte[N + 1], interim;
/*Попередні і наступні елементи множин допустимих сум,
більші N, не розглядаються */
nextSum[0] = 1; //Перед розглядом треків допустима лише сума 0
int i, j;
for (i = 0; i < S; i++) //Цикл по розглядуваних треках
 {/*Кожен трек може враховуватися, а може і не враховуватися в загальну суму.
Відповідно, можливими сумами довжин треків після розгляду чергового треку будуть всі
попередні суми (коли поточний трек не враховується) та попередні суми, збільшені на
довжину поточного треку*/
  interim = prevSum;
```

```
 prevSum = nextSum;
 nextSum = interim; /*При переході до наступного треку попередні можливі суми беруться з наявних наступних, а наступні можливі суми розраховуються. Для швидшої перестановки масивів не переприсвоюються їх значення, а міняються місцями посилання на них*/
 for (j = 0; j <= N; j++) /*Цикл по можливих сумах довжин запису на стрічку. Індекс масиву є значенням суми*/
  if (prevSum[j] == 1)
   {//Знайшли допустиму суму відносно попередніх треків
    nextSum[j] = 1;/*Якщо черговий трек не враховується, то попередня можлива сума не змінюється*/
    if (j + track[i] <= N) /*Якщо врахування чергового треку разом з можливою попередньою сумою поміститься на стрічці */
     nextSum[j + track[i]] = 1; } //Така сума треків вже можлива
  if (nextSum[N] == 1) break; }
/*Знаходимо максимальну з допустимих сум треків. Пошук починаємо з кінця, оскільки більшим індексам відповідають більші суми*/
maxSum = N;
while (nextSum[maxSum] == 0) maxSum--;
```

Результати тестування цієї підпрограми наведено на рис. 4. Бачимо, що усі тести вклалися у відведений час і при цьому використано найменше оперативної пам'яті з представлених варіантів.

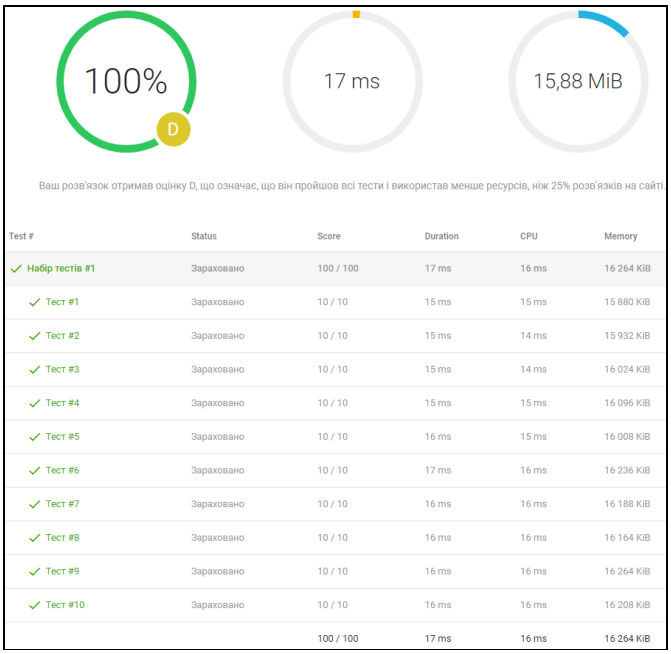


Рис. 4. Результати тестування програми з реалізацією методу поступового формування множини значень цільової функції

Прискорення розв'язання задачі у цьому випадку – не випадковість, адже обчислювальна складність алгоритму тепер становить $S \times N$, що значно менше $2^{S-1} \times S$ з врахуванням обмежень нашої задачі ($100 \times 200 << 2^{99} * 100$). На завершення зазначимо, що метод поступового формування множин допустимих значень знаходить лише розв'язок (екстремум) цільової функції цієї задачі комбінаторної оптимізації, але не визначає треки, при яких цей розв'язок отримується. Для знаходження цих треків потрібно використати інші підходи (наприклад, аналог зворотного ходу методу динамічного програмування [3, С. 392–447; 8]), але навіть знаходження екстремуму прискорює розв'язування оптимізаційних задач в класичному випадку.

З наведених результатів дослідження приходимо до таких **висновків**:

1. В процесі програмування розв'язування типових задач комбінаторної оптимізації доцільно застосовувати об'єктно-орієнтований підхід, описуючи розв'язування кожної задачі в окремому класі та даючи однакові назви тим самим методам розв'язання у різних задачах.

2. Під час визначення найефективнішого способу розв'язування задачі комбінаторної оптимізації недостатньо порівнювати час виконання на відомих тестових наборах, а доцільно намагатися попередньо проаналізувати їх обчислювальну складність.

3. Для прискорення розв'язування задач комбінаторної оптимізації недостатньо оминати деякі варіанти повного перебору, а потрібно мінімізувати ще й час обчислення цільової функції для кожного варіанта, враховуючи обмеження задачі.

4. Метод поступового формування множини значень цільової функції є дієвою альтернативою методам пошуку з поверненнями та врахування змін при розв'язуванні задач комбінаторної оптимізації, якщо область значень дискретна, а процес прийняття рішення можна розділити на окремі етапи як у методі динамічного програмування.

5. Для моделювання множин елементів в програмуванні доцільно використовувати логічні чи байтові масиви. Тоді для переприсвоєння множин достатньо переприсвоїти вказівки на масиви, а не переприсвоювати їх окремі елементи.

1. Korte B., Vygen J. Combinatorial Optimization: Theory and Algorithms, Third Edition. Berlin : Springer-Verlag, 2008. 646 p. ISBN 978-3-540-71843-7. **2.** Креневич А. П. Алгоритми і структури даних : підручник. Київ : ВПЦ «Київський Університет», 2021. 200 с. **3.** Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms, Third Edition. Kiyv : Dialektika, 2020. Vol. 1. 648 p. **4.** Knuth D. E. The Art of Computer Programmin. *Sorting and Searching*. 2-ed. Massachusetts : Addison Wesley Longman, 1997. Vol. 3. 800 p. **5.** Shportko A. V., Shportko L. V. Acceleration of large integer arrays sorting using ranges of values and frequencies of elements. *Information Extraction and Processing*. 2019. Vol. 47(123). P. 73–79. DOI:https://doi.org/10.15407/vidbir2019.47.073. **6.** Настенко Д. В., Нестерко А. Б. Об'єктно-орієнтоване програмування : навч. посіб. Частина 1. *Основи об'єктно-орієнтованого програмування на мові* C#. Київ : НТУУ «КПІ», 2016. 76 с. **7.** Hans Kellerer H, Pferschy U., Pisinger D. Knapsack Problems. Berlin : Springer, 2004. 546 p. ISBN 3-540-40286-1. **8.** Шпортько О. В., Малаш К. М. Застосування мемоізації в задачах мінімізації кількості знаків арифметичних операцій. *Вісник Національного університету водного господарства та природокористування. Сер. Технічні науки*. Рівне : НУВГП, 2020. № 2(90). С. 127–143.

REFERENCES:

1. Korte B., Vygen J. Combinatorial Optimization: Theory and Algorithms, Third Edition. Berlin : Springer-Verlag, 2008. 646 p. ISBN 978-3-540-71843-7. **2.** Krenevych A. P. Alhorytmy i struktury danykh : pidruchnyk. Kyiv : VPTs «Kyivskyi Universytet», 2021. 200 s. **3.** Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms, Third Edition. Kiyv : Dialektika, 2020. Vol. 1. 648 p. **4.** Knuth D. E. The Art of Computer Programmin. *Sorting and Searching*. 2-ed. Massachusetts : Addison Wesley Longman, 1997. Vol. 3. 800 p. **5.** Shportko A. V., Shportko L. V. Acceleration of large integer arrays sorting using ranges of values and frequencies of elements. *Information Extraction and Processing*. 2019. Vol. 47(123). P. 73–79. DOI:https://doi.org/10.15407/vidbir2019.47.073. **6.** Nastenko D. V., Nesterko A. B. Obiektno-oriientovane prohramuvannia : navch. posib. Chastyna 1. *Osnovy obiektno-oriientovanoho prohramuvannia na movi C#*. Kyiv : NTUU «KPI», 2016. 76 s. **7.** Hans Kellerer H, Pferschy U., Pisinger D. Knapsack Problems. Berlin : Springer, 2004. 546 p. ISBN 3-540-40286-1. **8.** Shportko O. V., Malash K. M. Zastosuvannia memoizatsii v zadachakh minimizatsii kilkosti znakiv aryfmetychnykh operatsii. *Visnyk Natsionalnoho universytetu vodnoho hospodarstva ta pryrodokorystuvannia. Ser. Tekhnichni nauky*. Rivne : NUVHP, 2020. № 2(90). S. 127–143.

Shportko A. V., Candidate of Economics (Ph.D.), Associate Professor (Academician Stepan Demianchuk International University of Economics Humanities, Rivne), **Mushyn M. M., Senior Student** (Academician Stepan Demianchuk International University of Economics Humanities, Rivne), **Bomba A. Ya., Doctor of Engineering, Professor** (National University of Water and Environmental Engineering, Rivne)

ABOUT THE OBJECT-ORIENTED APPROACH IN THE PROCESS OF PROGRAMMING ALGORITHMS FOR SOLVING COMBINATOR OPTIMIZATION PROBLEMS

The advantages of using an object-oriented approach for solving typical problems of combinatorial optimization are substantiated in the article. The method of gradually forming a set of values of the objective function as an alternative to the methods of search with returns and taking into account changes is described. Mechanisms of action of algorithms that use these methods to solve combinatorial optimization problems are compared. Fragments of programs that implement these algorithms in the C# programming language are presented, and the results of their testing in a remote computing environment are analyzed. According to the test results, it is shown that the implementation of the method of gradual formation of the set of values of the objective function drastically reduces the execution time of programs, which indicates the expediency of using this method in solving typical problems of combinatorial optimization.

Based on the results of the study, the following conclusions were made:

1. In the process of programming solving typical problems of combinatorial optimization, it is advisable to use an object-oriented approach. At the same time, the solution of each problem should be described in a separate class and the same names should be given to the same solution methods in different problems.

2. When determining the most efficient way to solve a combinatorial optimization problem, it is not enough to compare the execution time on known test sets, but it is advisable to try to analyze their computational complexity in advance.

3. In order to speed up the solution of combinatorial optimization problems, it is not enough to skip some variants of a complete search, but also to minimize the time of calculating the objective function for

each variant, taking into account the constraints of the problem.

4. The method of gradually forming a set of values of the objective function is an effective alternative to the methods of search with returns and taking into account changes when solving combinatorial optimization problems, if the range of values is discrete, and the decision-making process can be divided into separate stages, as in the method of dynamic programming.

5. It is advisable to use logical or byte arrays to model sets of elements in programming. Then it is sufficient to reassign pointers to arrays and not to reassign their individual elements in order to reassign sets.

***Keywords*: object-oriented programming; combinatorial optimization; method of gradual formation of a set of values of the objective function.**
