Методичні вказівки
і завдання до контрольної роботи
з дисципліни
«Теорія алгоритмів»

для студентів заочної форми навчання
освітньо-кваліфікаційного рівня бакалавр,
галузі знань 12 – «Інформаційні технології»,
спеціальності 122 – «Комп'ютерні науки
та інформаційні технології»,
спеціалізації «Комп'ютерні науки»

Навчальне видання

Методичні вказівки і завдання
до контрольної роботи
з дисципліни «Теорія алгоритмів»
для студентів заочної форми навчання
освітньо-кваліфікаційного рівня бакалавр,
галузі знань 12 – «Інформаційні технології»,
спеціальності 122 – «Комп'ютерні науки та
інформаційні технології»,
спеціалізації «Комп'ютерні науки»

Редактор Є. Плетньова
Укладачі: Арсенюк Ігор Ростиславович
Яровий Андрій Анатолійович
Майданюк Володимир Павлович

Оригінал-макет підготовлено І. Арсенюком

Підписано до друку 16.01.2018
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. 2,36
Наклад 40 (1-й запуск 1-20) пр. Зам. № 2018-020.

Видавець та виготовлювач
інформаційний редакційно-видавничий центр.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
Тел. (0432) 65-18-06.
press.vntu.edu.ua;
E-mail: kivc.vntu@gmail.com.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.

Міністерство освіти і науки України
Вінницький національний технічний університет

Методичні вказівки і завдання до контрольної роботи з дисципліни «Теорія алгоритмів»

для студентів заочної форми навчання освітньо-кваліфікаційного рівня бакалавр, галузі знань 12 – «Інформаційні технології», спеціальності 122 – «Комп'ютерні науки та інформаційні технології», спеціалізації «Комп'ютерні науки»

Вінниця
ВНТУ
2018

Рекомендовано до друку Методичною радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 6 від 16.02.2017 р.)

Рецензенти:
А. М. Пєтух, доктор технічних наук, професор
О. М. Хошаба, кандидат технічних наук, доцент

Методичні вказівки і завдання до контрольної роботи з дисципліни «Теорія алгоритмів» для студентів заочної форми навчання, освітньо-кваліфікаційного рівня бакалавр, галузі знань 12 – «Інформаційні технології», спеціальності 122 – «Комп'ютерні науки та інформаційні технології», спеціалізації «Комп'ютерні науки» / Уклад. Арсенюк І. Р., Яровий А. А., Майданюк В. П. – Вінниця : ВНТУ, 2018. – 41 с.

У методичних вказівках наведено основні рекомендації щодо вивчення дисципліни, завдання до контрольної роботи та типові розв'язування задач з урахуванням вимог до оформлення результатів.

ЗМІСТ

ВСТУП

Поняття алгоритму є центральним об'єктом спеціального розділу математики – *теорії алгоритмів*. Першим алгоритмом, в інтуїтивному розумінні, є запропонований Евклідом у III ст. до н. е. алгоритм пошуку найбільшого загального дільника двох чисел. До початку XX ст. слово «алгоритм» вживалося у словосполученні «алгоритм Евкліда», а для опису покрокового розв'язання математичних задач використовувалося слово «метод».

Точкою відліку сучасної теорії алгоритмів є робота математика К. Геделя (1931 р. – теорема про неповноту символічних логік), в якій показано, що деякі математичні проблеми не можна розв'язати алгоритмами певного класу. Це був поштовх до пошуку і аналізу різних формалізацій алгоритму.

Перші фундаментальні роботи з ТА були опубліковані незалежно у 1936 р. А. Тьюрінгом, А. Черчем та Е. Постом. Запропоновані ними машина Тьюрінга, лямбда-числення Черча та машина Поста були еквівалентними формалізаціями алгоритму. Розвитком цих робіт стало формулювання і доведення алгоритмічно нерозв'язуваних проблем. У 1950-і роки істотний внесок у ТА внесли Н. Колмогоров і А. Марков.

До 1970-х років сформувалися три напрями в ТА: 1) *класична ТА* (формулювання завдань у термінах формальних мов, поняття задачі розв'язності, введення класів складності, формулювання проблеми P=NP, відкриття класу NP-повних задач та його дослідження); 2) *теорія асимптотичного аналізу алгоритмів* (поняття складності і трудомісткості алгоритму, критерії оцінювання алгоритмів, методи отримання асимптотичних оцінок); 3) *теорія практичного аналізу алгоритмів* (отримання явних функцій трудомісткості, інтервальний аналіз функцій, практичні критерії якості алгоритмів, методика вибору раціональних алгоритмів).

Одержані в ТА результати знаходять широке практичне застосування. Завдяки ним стає можливим раціональний вибір алгоритму розв'язання цієї задачі; отримання часових оцінок розв'язання задач; отримання вірогідних оцінок неможливості розв'язання деякої задачі за певний час; розробку та вдосконалення алгоритмів на основі практичного аналізу тощо; відповісти на питання можливості алгоритмічно розв'язати певну задачу та визначити чи не належить вона до класу NP-повних завдань.

Метою дисципліни «Теорія алгоритмів» є вивчення основних підходів щодо уточнення поняття алгоритму; властивостей та форм подання алгоритмів; поняття ефективності алгоритмів та її оцінювання; деяких методів

розробки алгоритмів; різних алгоритмів сортування з аналізом їх ефективності; основ динамічного програмування; основ застосування «скупих» («жадібних») алгоритмів; поняття NP-повноти та прикладів NP-повних задач.

У методичних вказівках наведено основні теми дисципліни та їх зміст, теоретичні завдання та практичні задачі до контрольної роботи, перелік питань до іспиту, приклади розв’язування типових задач, список рекомендованої літератури та основні вимоги до оформлення контрольної роботи.

ЗМІСТ ДИСЦИПЛІНИ

ПОНЯТТЯ, ФОРМИ ПОДАННЯ ТА ВЛАСТИВОСТІ АЛГОРИТМІВ

Вступ до теорії алгоритмів. Предмет, задачі та структура дисципліни. Інтуїтивне поняття та потенційне виконання алгоритму. Аналіз основних властивостей та форм подання алгоритмів. Основні підходи щодо уточнення поняття алгоритму. Рекурсивні функції. Машина Тьюрінга. Тезис Тьюрінга. Нормальні алгоритми Маркова.

ОСНОВИ АНАЛІЗУ ЕФЕКТИВНОСТІ ТА СКЛАДНОСТІ АЛГОРИТМІВ

Швидкість зростання функцій. Асимптотичні позначення. Стандартні функції і позначення. Поняття ефективності алгоритмів. Критерії оцінювання складності алгоритмів. Часова складова і асимптотична часова складова алгоритмів. Оцінювання часу виконання програм. Про корисність швидких алгоритмів. Рекурентні співвідношення. Метод підстановки. Метод ітерацій.

АБСТРАКТНІ СТРУКТУРИ ДАНИХ

Основні абстрактні структури даних та їх застосування. Стек: організація, основні операції, приклади застосування. Черга. Реалізація черги. Основні операції над чергою. Приклади застосування черги. Дек: реалізація, основні операції, застосування. Списки: реалізація, основні операції, застосування. Двійкові дерева. Купа: властивості купи, основні операції над купою, реалізація та приклади застосування.

ДОСЛІДЖЕННЯ ТА АНАЛІЗ ДЕЯКИХ АЛГОРИТМІВ СОРТУВАННЯ

Моделі внутрішніх та зовнішніх обчислень. Внутрішнє та зовнішнє сортування. Алгоритми сортування вставками та методом злиття. Аналіз ефективності алгоритмів. Сортування за допомогою купи. Побудова, властивості та основні операції над купою. Аналіз часу виконання алгоритму. Аналіз алгоритму швидкого сортування. Алгоритми сортування за лінійний час: аналіз та оцінювання часу виконання. Сортування методом підраховування та вичерпуванням. Цифрове сортування. Порівняльний аналіз методів сортування.

ДЕЯКІ МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ

Метод «розділяй та володарюй». Динамічне програмування. Теоретичні основи «жадібних» алгоритмів. Задача про здачу. Дискретна та безперервна задачі про рюкзак. Алгоритм Дейкстри. Пошук з поверненням. Альфа-бета відсічення, метод гілок та границь. Обмеження евристичних алгоритмів. Алгоритми локального пошуку.

ВСТУП ДО NP-ПОВНИХ ЗАДАЧ

Клас NP-повних задач та їх приклади. Поліноміальний час розв'язку. Перевірка належності класу NP. NP-повнота і зведення. Деякі NP-повні задачі. Задача про виконуваність схеми. Задача про кліку. Задача про вершинне покриття. Задача про суми підмножин. Задача пошуку гамільтонового циклу у графі.

ЗАВДАННЯ ДО КОНТРОЛЬНОЇ РОБОТИ

Індивідуальне завдання до контрольної роботи з дисципліни «Теорія алгоритмів» складається з шести завдань: теоретичного питання та п'яти практичних задач, в тому числі задачі на складання та аналізу програмного коду.

Завдання № 1
(Теоретичні питання)

Дайте відповідь на питання відповідно до варіанта, визначеного викладачем.

1. Логічна та аналітична теорія алгоритмів. Основні властивості алгоритмів. Аналіз основних форм подання алгоритмів.
2. Основні підходи до уточнення поняття алгоритму. Рекурсивні функції.
3. Машина Тьюрінга: сутність, призначення, основні позначення. Тезис Тьюрінга.
4. Нормальні алгоритми Маркова: сутність, призначення, основні визначення та приклади. Тезис Тьюрінга.
5. Швидкість зростання функцій. Асимптотичні позначення. Стандартні функції і позначення.
6. Поняття ефективності алгоритмів. Критерії оцінювання складності алгоритмів. Часова складова і асимптотична часова складова алгоритмів. Оцінювання часу виконання програм. Про корисність швидких алгоритмів.
7. Рекурентні співвідношення як засіб оцінювання складності рекурсивних алгоритмів. Метод підстановки. Метод ітерацій. Загальний рецепт.
8. Динамічне програмування: сутність, область застосування, приклади.
9. Теоретичні основи «скупих» алгоритмів. Приклади «скупих» алгоритмів.
10. Клас NP-повних задач та їх приклади. Поліноміальний час розв'язку. Перевірка належності класу NP.
11. NP-повнота і зведення. Деякі NP-повні задачі. Задача про виконуваність схеми.
12. Двійкові дерева: сутність, реалізація, основні операції, приклади застосування.

13. Наведіть та охарактеризуйте основні алгоритми сортування, що мають складність $O(n \cdot log(n))$.

14. Купа: властивості купи, основні операції над купою, реалізація та приклади застосування.

15. Стек: організація, основні операції, область застосування.

16. Черга. Реалізація черги. Основні операції над чергою. Приклади застосування черги.

17. Структура даних дек: реалізація, основні операції, приклади застосування.

18. Порівняльний аналіз основних алгоритмів сортування, що мають квадратичну складність.

19. Метод «розділяй та володарюй» як засіб розробки ефективних алгоритмів. Приклади застосування методу «розділяй та володарюй».

20. Списки. Види списків, основні операції над списками та область їх застосування.

Завдання № 2
(Оцінювання швидкості зростання функцій)

Розташуйте, вказані згідно з Вашим варіантом (таблиця 1), функції у порядку збільшення їх ступеня зростання. Якщо серед наведених функцій є такі, що мають однакову швидкість зростання, вкажіть ці функції.

Таблиця 1 – Варіанти для завдання № 2

Варіант	Функції *f(n)*	Варіант	Функції *f(n)*
1	2^n; $log_2(log_2 n)$; n; $n^3+log_2 n$; 15	2	n^2; n^3; $nlog_2 n$; $(log_2 n)^2$; $n^{1/2}$
3	$n!$; n; $(3/2)^n$; $5n$; n^2	4	$n^2 log_3 n$; $(log_2 n)^2$; $n^{1/2}$; $3n$; $5n^3$
5	$10n^{1/2}$; $log_2 n$; $1.5n^2$; 0.1^n; 10	6	n^2; n^4; $n!$; 1.5^n; $n^3 log_2 n$
7	$90n^2$; $2n^3$; $log_2 n^3$; $(log_2 n)^3$; $n^{1.5}$	8	3^n; $log_3 n^2$; n; $n+log_3 n$; $nlog_2 n$
9	$n^3 log_2 n$; $log_{10} n^4$; $10n^{3/2}$; n; $n^{3/2}$	10	$2.5n^2$; 1.2^n; $8nlog_2 n^2$; $n!$; n
11	$n!$; $nlog_4 n$; 3^n; n^{100}; $100n^{20}$	12	e^n; $log_2(n^3)$; n^2; $n^3+log_2 n$;
13	$n^2 log_3 n$; $9log_8 n^3$; $n^{1/3}$; $7n$; $1/5n^3$	14	$9n^7$; $7n^9$; $log_2 n^3$; $log_3 n^2$; $33n^2$
15	$17n^3$; 0.5^n; $3n^3$; $n5$; $n+n^2$	16	1.3^n; $n^2 log_{10} n$; n; $2n^3$; $nlog_2 n$
17	$20n^2$; $3n^4$; $2n!$; e^n; $n^5 log_{10} n$	18	$100n^{1/2}$; $99log_2 n$; $1.5n^2$; 1^n; n^5
19	$n^4 log_4 n$; $5n^2+n$; $3n^3$; n; n^2	20	$10n^2$; n^3; $nlog_2 n$; $0.5n^2$; $90n^{1/2}$

Завдання № 3
(Застосування Θ- та O-позначень)

Для заданої згідно з Вашим варіантом (таблиця 2) функції *f(n)* вкажіть, що саме потрібно записати замість зірочки * для $f(n)=\Theta(*)$ та $f(n)=O(*)$.

Таблиця 2 – Варіанти для завдання № 3

Варіант	Функція *f(n)*	Варіант	Функція *f(n)*
1	$2(1/3n^3 + 5n \cdot log_2 n + 8)$	2	$n(3n^2 + 2n + log_4 n)$
3	$1/3n^6 + 3n^4 + 5n^2 - n$	4	$1/2(3n^4 + 5n^2 – 2n + log_2 n)$
5	$10n + 5n^2 + 3n^3 + 0.5n^4$	6	$1/3(n)(n-1)(n+1)$
7	$3(n^2 + 5n \cdot log_2 n + n^{1/2})$	8	$3n^3 + 10n - 2 + 7n^2 + 20n^{1/5}$
9	$2(1.6n + 3n^5 + 5n^2 - 3n)$	10	$-5n + 2n^2 + 3.3n^4 + 0.5n^6$
11	$1/2(n)(n+1)^2$	12	$3n^3 + 31n^2 + n^4 - 5n$
13	$100 + 2n^3 + 3.3n^5 + 1.3n^7$	14	$27n + 15n^2 + n^4 + 0.5n log_2 n$
15	$2(n^4 + 18n^2 - 2n + 99log_2 n)$	16	$2n^2(2n^2 + 8n log_2 n + 20log_4 n)$
17	$2(1/3n + 3n^4 + 5n^3 - n^2)$	18	$1/4n^5 + 10n^3 - 6 + 7n^2 + log_2 n$
19	$2(2n^3 + 5n^2 \cdot log_2 n – 2n)$	20	$1/2(4n^4 + 8n^2 \cdot log_2 n + 20n^{3/2})$

Завдання № 4
(Визначення доцільності застосування алгоритму із заданою складністю для заданого розміру входу)

1. У Вас є 5 алгоритмів a_1, a_2, a_3, a_4, a_5 з часовими складностями, що наведені у таблиці 3. Обґрунтуйте, які з цих алгоритмів доцільно використовувати для різних $n \geq 2$ (де n – розмір входу).

2. А) Визначте розмір входу, який може бути оброблений вищевказаними алгоритмами за 1 секунду, 1 хвилину та 1 годину, якщо одиницею часу прийняти 1 мікросекунду.

Б) Покажіть, як зміниться розмір входу для кожного з цих алгоритмів, якщо швидкодія комп’ютера зросте на порядок. Час обробки даних при цьому в обох випадках (до підвищення швидкодії комп’ютера та після неї) повинен бути однаковим. Прийміть, що час розв’язання дорівнює 1 годині.

В) Покажіть, як зміниться максимальний розмір входу, що можна обробити за 1 годину, якщо алгоритм a_5 замінити на a_4, а також, якщо алгоритм a_4 замінити на a_2.

Зробіть відповідні висновки щодо задачі.

Таблиця 3 – Варіанти для завдання № 4

Варіант	Складність алгоритму a_1	Складність алгоритму a_2	Складність алгоритму a_3	Складність алгоритму a_4	Складність алгоритму a_5
1	$50n$	$10n \cdot log_2(n)$	$3n^2$	n^3	2^n
2	$4200n^{1/2}$	$580n$	$15n^2$	$n^3/2$	$0.5 \cdot 2^n$
3	$70n$	$15n \cdot log_2(n)$	$4n^2$	$0.5n^3$	2^n
4	$300n$	$50n \cdot log_2(n)$	$14n^2$	$2n^3$	2^n
5	$600n$	$5n^2 \cdot log_2(n)$	$2n^3$	$0.1n^4$	2^n
6	$80n$	$18n \cdot log_2(n)$	$4.3n^2$	$n^3/3$	2^n
7	$1000 \cdot log_2(n)$	$300n$	$50n^2$	$5n^2 \cdot log_2(n)$	2^n

Продовження таблиці 3

8	$90n$	$19n \cdot \log_2(n)$	$4n^2$	$2n^3$	2^n
9	$100n$	$22n \cdot \log_2(n)$	$5n^2$	$3n^3/2$	2^n
10	$150n$	$30n \cdot \log_2(n)$	$10n^2$	$2n^3$	2^n
11	$200n$	$40n \cdot \log_2(n)$	$13n^2$	$n^3/10$	2^n
12	$605n$	$95n \cdot \log_2(n)$	$12n^2$	$3n^2 \cdot \log_2(n)$	$2 \cdot 2^n$
13	$50n$	$10n \cdot \log_2(n)$	$3n^2$	n^3	2^n
14	$398n$	$62n \cdot \log_2(n)$	$5n^2$	$1.05n^2 \cdot \log_2(n)$	$0.5 \cdot 2^n$
15	$300n$	$47n \cdot \log_2(n)$	$6n^2$	$1.35n^2 \cdot \log_2(n)$	2^n
16	$20000n$	$500n^2$	$19n^3$	n^4	$3 \cdot 2^n$
17	$1000n$	$155n \cdot \log_2(n)$	$30n^2$	$2n^3$	$2 \cdot 2^n$
18	$1250n$	$200n \cdot \log_2(n)$	$27n^2$	$1.5n^3$	$1.2 \cdot 2^n$
19	$9666n$	$155n^2 \cdot \log_2(n)$	$30n^3$	n^4	2^n
20	$5000n^{1/2}$	$600n$	$3.3n^2 \cdot \log_2(n)$	n^3	$1.5 \cdot 2^n$

Завдання № 5
(Оцінювання рекурентних співвідношень рекурсивних алгоритмів методами підстановки, ітерації та із застосуванням основної теореми про оцінювання рекурентних співвідношень)

Розв'яжіть задачу відповідно до варіанта, визначеного викладачем.

1. Використовуючи основну теорему про оцінювання рекурентних співвідношень, знайдіть асимптотично точні оцінки для таких співвідношень:

1) $T(n) = T(n/2) + 1$;
2) $T(n) = 4T(n/2) + n^2$;
3) $T(n) = 2T(n/2) + n$;
4) $T(n) = 4T(n/2) + n^2$;
5) $T(n) = 6T(n/3) + n^2$;
6) $T(n) = 4T(n/2) + n^3$.

2. Знайдіть асимптотичну оцінку для нижченаведених співвідношень (вважаємо, що $T(n)$ – константа при $n \le 2$), використовуючи дерево рекурсії.

7) $T(n) = 2T(n/2) + n^3$;
8) $T(n) = 4T(n/3) + 1$;
9) $T(n) = T(9n/10) + n$;
10) $T(n) = 3T(n/2) + 1$;
11) $T(n) = 4T(n/2) + n^2$;
12) $T(n) = T(n - 1) + n$;
13) $T(n) = 4T(n/2) + n$.

3. Методом ітерацій знайдіть асимптотичну оцінку для нижченаведених співвідношень (вважаємо, що $T(n)$ – константа при $n \le 2$).

14) $T(n) = 3T(n/2) + 1$;
15) $T(n) = 2T(n/2) + n^3$;
16) $T(n) = 4T(n/3) + 1$;
17) $T(n) = 2T(n/2) + n$;
18) $T(n) = T(3n/4) + n$;
19) $T(n) = 4T(n/2) + n$;
20) $T(n) = 7T(n/2) + n^2$.

Завдання № 6

Розв’яжіть задачу № 1 або № 2, де номер задачі визначає викладач.

Задача № 1
(програмування та аналіз алгоритмів сортування різної часової складності)

Наведіть графічну схему алгоритму сортування згідно з Вашим варіантом (таблиця 4) з коментарями, власний приклад роботи алгоритму та код програми (з коментарями) на обраній мові програмування.

Таблиця 4 – Варіанти для завдання № 6

Варіант	Алгоритм сортування	Критерій сортування
1	сортування вставками	за зростанням
2	сортування прямим вибором	за зростанням
3	бульбашкове сортування	за зростанням
4	модифіковане бульбашкове сортування	за зростанням
5	шейкерне сортування	за зростанням
6	сортування злиттям	за зростанням
7	швидке сортування	за зростанням
8	методом підраховування	за зростанням
9	цифрове сортування	за зростанням
10	сортування вичерпуванням	за зростанням
11	сортування вставками	за спаданням
12	сортування прямим вибором	за спаданням
13	бульбашкове сортування	за спаданням
14	модифіковане бульбашкове сортування	за спаданням
15	сортування злиттям	за спаданням
16	за допомогою купи	за спаданням
17	швидке сортування	за спаданням
18	сортування вичерпуванням	за спаданням
19	цифрове сортування	за спаданням
20	сортування методом підраховування	за спаданням

Виконайте теоретичний аналіз складності алгоритму та побудуйте графік залежності часу виконання програми від розміру входу.

Виконайте практичні дослідження Вашої програми, для чого виміряйте (програмно) час її виконання у випадку сортування лінійного масиву чисел, що складається з 10^3, 10^4, $5 \cdot 10^4$, 10^5, $5 \cdot 10^5$, 10^6 елементів у таких трьох випадках: 1) масив вже відсортовано; 2) масив не відсортовано; 3) масив відсортовано у зворотному порядку. Для кожного з цих 3-х випадків наведіть таблицю та графік залежності часу виконання програми від розміру входу.

Порівняйте практичні результати з теоретичними розрахунками.

Зробіть висновки щодо складності програмування, часової та просторової складностей, стійкості а також інших переваг, недоліків та особливостей цього алгоритму.

Задача № 2
(застосування динамічного програмування та «жадібних» алгоритмів)

І. Поясніть сутність динамічного програмування. В яких випадках доцільно використовувати динамічне програмування? Поясніть, як за допомогою динамічного програмування розв'язати нижченаведену задачу. Наведіть загальний алгоритм розв'язання задачі та програмний код його реалізації. Наведіть результати роботи Вашої програми для кількох довільних прикладів.

1. Визначте мінімальний набір монет з набору монет номіналом 1, 2, 3, 5 копійок, потрібний для видачі грошей у сумі n ($n \le 1000$) копійок.

2. Кенгуру хоче перебратися на іншу сторону річки. Він знає, що через річку є стежка з n-рівновіддалених купин ($2 \le n \le 30$). Кенгуру може стрибати на наступну купину або через одну купину. Визначте максимальну кількість маршрутів, якими кенгуру може подолати річку.

3. Обчисліть кількість n-значних (n – є парним, що не перевищує 10) щасливих квитків (квиток щасливий, якщо сума першої половини його цифр дорівнює сумі другої його половини). Наприклад, шестизначний квиток з номером 302005 щасливий, оскільки 3+0+2=0+0+5.

4. Кенгуру хоче перебратися на іншу сторону річки. Він знає, що через річку є стежка з n-рівновіддалених купин ($2 \le n \le 30$). Кенгуру може стрибати на наступну купину, через одну, а також через дві купини. Визначте максимальну кількість маршрутів, якими кенгуру може подолати річку.

5. Визначте кількість способів дати клієнтові суму грошей розміру n гривень ($n \le 100$), якщо є нескінченна кількість кожної купюри номіналом 1, 2, 3, 5 гривень. Примітка: порядок видачі купюр має значення, тобто, наприклад, здачі 1, 2, 5, або 2, 1, 5, або 5, 1, 2 тощо вважаються різними.

6. Є послідовність чисел k_i (кожне k_i є цілим, що не перевищує 10000 за модулем) довжини n ($1 \le n \le 200$). Знайдіть довжину найбільшої підпослідовності, що зростає.

7. Визначте кількість способів дати клієнтові здачу розміру n копійок ($n \le 100$), якщо є нескінченна кількість кожної монети номіналом 1, 3, 5, 10 копійок. Примітка: порядок видачі монет має значення, тобто, наприклад, здачі 1, 3, 10, або 3, 1, 10, або 10, 3, 1 тощо вважаються різними.

8. Визначте кількість послідовностей довжини n ($n \le 100$), що складаються з цифр 0 та 1, в яких ніякі дві одиниці не розташовуються поряд.

9. Є квадратна сітка (матриця) розміру *xy* (*x* горизонтальних паралельних відрізків, кожен з яких перетинається *y* вертикальними паралельними відрізками). Ліва верхня точка *A* перетину має координати (0, 0), а права нижня точка *B* – координати (*x*, *y*). Кулька може котитися по відрізках такого матричного поля (рисунок 1) лише вниз і вправо. Змінювати напрямок руху кулька може тільки у точках перетину вертикальних та горизонтальних відрізків. Визначте кількість шляхів, рухаючись якими кулька з точки *A* потрапить у точку *B*. Значення координат $2 \le x,y \le 100$.

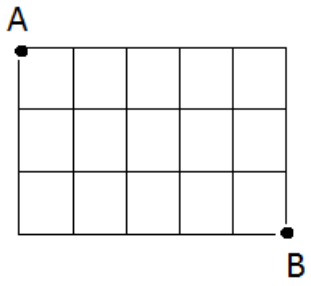


Рисунок 1 – Сітка для руху кульки

10. Визначте кількість способів дати клієнтові грошову суму розміру *n* копійок ($n \le 100$), якщо є монети номіналом 1, 2, 3, 5 копійок (кількість кожної монети дорівнює *m*) і потрібно використати рівно *m* монет. Примітка: порядок видачі монет має значення, тобто, наприклад, здачі 1, 3, 10, або 3, 1, 10, або 10, 3, 1 тощо вважаються різними.

II. Поясніть, як за допомогою «скупого» («жадібного») алгоритму розв'язати нижченаведену задачу. Наведіть загальний алгоритм розв'язання задачі та програмний код його реалізації. Наведіть результати роботи Вашої програми для кількох довільних прикладів.

1. Виконайте переведення числа *n* ($1 \le n \le 10000$) з десяткової системи числення у двійкову.
2. Є *k* ($k \le 100$ куп піску з дорогоцінних каменів). Купи мають ваги (у кілограмах) $m_1, m_2, \ldots, m_k$ ($1 \le m_i \le 100$, $i=1, 2, \ldots, k$) та вартості (у гривнях) одиниці піску $v_1, v_2, \ldots, v_k$ ($1 \le v_i \le 1000$, $i=1, 2, \ldots, k$) відповідно. У Вас є рюкзак, що може вмістити *n* кілограмів ваги. Визначте, як заповнити рюкзак піском із цих куп так, щоб вартість піску виявилася максимальною. Примітка: для спрощення вважайте, що Ви можете покласти до рюкзака будь-яку вагу піску, іншими словами не у дискретних значеннях (тобто кратну чомусь), а безперервну.
3. Знайдіть розкладання натурального числа *n* ($1 \le n \le 1000$) в суму ступенів двійки у спадному порядку. Наприклад, для $n=19$ результат 16+1+1+1.
4. Потрібно видати грошу премію у сумі *n* гривень ($1 \le v_i \le 10000$) мінімальною кількістю купюр номіналами 1, 2, 5, 10, 20, 50, 100, 200, 500 гривень.
5. Знайдіть розкладання натурального числа *n* ($1 \le n \le 1000$) в суму непарних ступенів двійки у спадному порядку. Наприклад, для $n=17$ результат 8+8+1.
6. Визначте розкладання правильного дробу на суму дробів, чисельник кожної з яких дорівнює одиниці. Наприклад, 39/50=1/2+1/4+1/34+1/1700.
7. Знайдіть розкладання натурального числа *n* ($1 \le n \le 1000$) в суму пар-

них ступенів двійки у спадному порядку. Наприклад, для n=13 результат 4+4+4+1.

8. На відрізку [0, k] (де $3 \le k \le 1000$) розташовані відрізки v_i, які з надлишком покривають відрізок. Виберіть мінімальну кількість відрізків v_i таких, щоб вони покривали відрізок [0, k].

9. Відомо, що пасажирський автомобіль може перевезти не більше M кілограмів. Є k претендентів, що хочуть проїхатися в автомобілі. Вага кожного претендента дорівнює відповідно m_1, m_2, ..., m_k. Визначте максимальну кількість претендентів, що можуть бути перевезені за один раз.

10. Потрібно видати зарплату у сумі n гривень ($1 \le v_i \le 10000$) мінімальною кількістю купюр номіналами 1, 2, 5, 10, 20, 50, 100 гривень.

ПИТАННЯ ДО ІСПИТУ
з дисципліни «Теорія алгоритмів»

1. Наведіть інтуїтивне поняття алгоритму. Потенційне виконання алгоритму.
2. Властивості алгоритмів та їх тлумачення.
3. Основні вимоги, що пред'являються до алгоритмів.
4. Форми подання алгоритмів.
5. Роль алгоритмів у науці і техніці.
6. Основні підходи до уточнення поняття «алгоритм». Універсальні алгоритмічні моделі.
7. Швидкість зростання функцій. Асимптотичні позначення.
8. Основні способи розв'язання рекурентних співвідношень, їх переваги та недоліки.
9. Розв'язання рекурентних співвідношень методом підстановки.
10. Розв'язання рекурентних співвідношень методом ітерацій.
11. Розв'язання рекурентних співвідношень із застосуванням дерева рекурсії.
12. Основна теорема про рекурентні співвідношення та приклади її застосування.
13. Ефективність алгоритмів.
14. Питання корисності швидких алгоритмів.
15. Критерії оцінювання складності алгоритмів.
16. Часова складова і асимптотична часова складова алгоритмів.
17. Оцінювання часу виконання програм.
18. Аналіз рекурсивних програм.
19. Машина Тьюрінга (МТ) як простіша модель обчислень. Основні визначення. Поняття детермінованої та недетермінованої МТ.
20. Деякі операції над машиною Тьюрінга. Тезис Тьюрінга.
21. Нормальні алгоритми Маркова та їх теоретична значимість.
22. Стисла класифікація алгоритмів сортування.

23. Алгоритм сортування вставками. Аналіз асимптотичної складової часу виконання алгоритму.

24. Бульбашковий алгоритм сортування. Аналіз часу виконання алгоритму.

25. Алгоритм сортування вибором. Аналіз асимптотичної складової часу виконання алгоритму.

26. Алгоритм сортування методом злиття. Оцінювання ефективності часу роботи алгоритму.

27. Сортування за допомогою купи (пірамідальне сортування). Побудова купи. Властивості купи.

28. Основні операції над купою. Аналіз часу виконання алгоритму.

29. Алгоритм сортування методом підраховування. Аналіз часу виконуваності алгоритму.

30. Алгоритм цифрового сортування. Аналіз часу виконання алгоритму.

31. Алгоритм сортування вичерпуванням. Переваги, недоліки та обмеження на застосування алгоритму. Аналіз часової складності алгоритму.

32. Порівняльний аналіз основних алгоритмів сортування, що мають асимтотичну часову складність $T(n)=O(n^2)$ за критеріями стійкості, об'єму пам'яті, складності програмування та обмежень щодо застосування.

33. Порівняльний аналіз основних алгоритмів сортування, що мають асимтотичну часову складність $T(n)=O(n \cdot \log n)$ за критеріями стійкості, об'єму пам'яті, складності програмування та обмежень щодо застосування.

34. Порівняльний аналіз основних алгоритмів сортування, що мають асимтотичну часову складність $T(n)=O(n)$ за критеріями стійкості, об'єму пам'яті, складності програмування та обмежень щодо застосування.

35. Основне правило застосування «жадібних» алгоритмів. Наведіть приклади, коли потрібно і коли не потрібно застосовувати «жадібні» алгоритми.

36. Розв'язання задачі про вибір заявок за допомогою «жадібного» алгоритму. Аналіз часу виконання алгоритму.

37. Алгоритм Дейкстри. Сутність, особливості застосування, обмеження.

38. Розв'язання безперервної та дискретної задачі про рюкзак за допомогою «жадібного» алгоритму.

39. Основи застосування динамічного програмування (ДП). Наведіть приклади задач, які доцільно розв'язувати із застосуванням ДП.

40. Складнісні класи задач. Поняття класу складності. Класи P та NP. Наведіть основні переваги алгоритмів класу P. Приклади задач, що належать до класу P.

41. Поняття «NP-повнота» і «зведення». Наведіть та стисло охарактеризуйте декілька прикладів NP-повних задач.

ТИПОВІ РОЗВ'ЯЗУВАННЯ

ПЕРШИЙ ТИП ЗАДАЧ

1. Нехай є 5 алгоритмів, що призначені для розв'язання деякої задачі. Часові складності алгоритмів a_1, a_2, a_3, a_4 і a_5 дорівнюють відповідно $1000n$, $l00n \cdot log\ n$, $10n^2$, n^3 і 2^n. Визначте, який алгоритм доцільніше використовувати при заданому розмірі входу $n \geq 2$ [1, 8].

2. А) Визначте розмір входу, який може бути оброблений вищевказаними алгоритмами за 1 секунду, 1 хвилину та 1 годину, якщо одиницею часу прийняти 1 мікросекунду.

Б) Покажіть, як зміниться розмір входу для кожного з цих алгоритмів, якщо швидкодія комп'ютера зросте на порядок. Час обробки даних при цьому в обох випадках (до підвищення швидкодії комп'ютера та після неї) повинен бути однаковим. Прийміть, що час розв'язання дорівнює 1 годині.

В) Покажіть, як зміниться максимальний розмір входу, що можна обробити за 1 годину, якщо алгоритм a_4 замінити на a_2.

Розв'язання

1. Ідея розв'язання полягає в тому, щоб замість n підставляти у вирази складностей ($1000n$, $l00n \cdot log\ n$, $10n^2$, n^3 і 2^n) конкретні значення, що може набувати n, тобто 1, 2, 3, ... та порівнювати, для кожного такого значення, отримані результати. Той алгоритм, який дає найменший результат на конкретному n і буде кращим для цього значення n. Реалізувавши вказану ідею, робимо висновок, що алгоритм a_5 буде найкращим для задач розміром $2 \leq n \leq 9$, a_3 – для задач розміром $10 \leq n \leq 58$, a_2 – при $59 \leq n \leq 1024$, а a_1 – при $n > 1024$.

Проте, не потрібно забувати, що в більшості випадків програмістів цікавить той алгоритм, асимптотика зростання часової складності якого буде найменша. В цьому випадку цей алгоритм a_1.

2. А) Розмір входу, який може бути оброблений вищевказаними алгоритмами за 1 секунду, 1 хвилину та 1 годину наведено нижче у таблиці 5.

Таблиця 5 – Максимальний розмір входу для алгоритмів $a_1 - a_5$

Алгоритм	Часова складність	Максимальний розмір входу задачі		
		1 секунда	1 хвилина	1 година
a_1	$1000n$	1000	60000	36000000
a_2	$100n\ log\ n$	1002	4895	1736782
a_3	$10n^2$	316	2449	18973
a_4	n^3	100	391	1532
a_5	2^n	19	25	31

Б) Якщо швидкодія комп'ютера зросте на порядок, то максимальний

розмір задачі для алгоритму a_1 зросте у 10 разів, для a_2 – у 8,7 раза, для a_3 – у 3,16 раза, для a_4 – у 2,15 раза, а для a_5 – лише на 3,3.

В) Якщо алгоритм a_4 замінити на a_2, то максимальний розмір входу, що можна обробити за час, як видно з таблиці 3, зміниться більше ніж у 1133 рази.

ДРУГИЙ ТИП ЗАДАЧ

1. Оцініть рекурентне співвідношення $T(n) = 2T(\lfloor n/2 \rfloor) + n$ методом підстановки, де позначення $\lfloor n/2 \rfloor$ – найбільше ціле, що менше або дорівнює *n/2* [1, 5, 8].

Розв'язання

Зауважимо, що співвідношення $T(n) = 2T(\lfloor n/2 \rfloor) + n$ має місце у випадку, коли алгоритм розбиває задачу розміром *n* на 2 підзадачі розміром *n/2*. Ці підзадачі розв'язуються рекурсивно (кожна за час *T(n/2)*) і результати об'єднуються. При цьому витрати на розбивання й об'єднання дорівнюють *n*.

Припустимо, що $T(n)=O(n \cdot \log n)$, тобто $T(n) \le cn \cdot \log n$ для відповідного $c>0$. Доведемо це за індукцією. Нехай ця оцінка правильна для $\lceil n/2 \rceil$, тобто $T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$. Підставивши її у співвідношення, одержимо

$$T(n) \le 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \le cn\log(n/2) + n =$$

$$= cn\log n - cn\log 2 + n = cn\log n - cn + n \le cn\log n.$$

Останній перехід законний при $c \ge 1$.

Залишається перевірити базис індукції, тобто довести оцінку для початкового значення *n*. Отут ми зіштовхуємося з тим, що при $n=1$ права частина нерівності перетворюється в нуль, яким би не взяти *c* (оскільки $\log 1 = 0$). У цьому випадку потрібно пам'ятати, що асимптотичну оцінку досить довести для всіх *n*, починаючи з деякого. Підберемо *c* так, щоб оцінка $T(n) \le cn \cdot \log n$ була правильна при $n=2$ і $n=3$. Тоді для великих *n* можна міркувати за індукцією, і небезпечний випадок $n=1$ нам не зустрінеться (оскільки $\lfloor n/2 \ge 2 \rfloor$ при $n>3$).

Тепер виникає питання: як вгадати відповідь? Для цього, як вже згадувалося вище, потрібні певні навички і досвід. Наведемо приклади, які наштовхують на думку.

Аналогія. Розглянемо для прикладу співвідношення

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n,$$

яке відрізняється від $T(n) = 2T(\lfloor n/2 \rfloor) + n$ додатковим доданком *17* у правій частині. Можна очікувати, однак, що при великих *n* різниця між

$\lfloor n/2 \rfloor + 17$ і $\lfloor n/2 \rfloor$ навряд чи така вже істотна. Природно припустити, що оцінка $T(n)=O(n \cdot \log n)$ залишається в силі, а потім довести це за індукцією.

2. Оцініть методом ітерації співвідношення $T(n) = 3T(\lfloor n/4 \rfloor) + n$.

Розв'язання

Підставляючи це співвідношення само у себе, одержимо:

$$T(n) = n + 3T(\lfloor n/4 \rfloor) = n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) = n + 3(\lfloor n/4 \rfloor +$$

$$+3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) = n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor).$$

Тут ми скористалися тим, що $\lfloor \lfloor n/4 \rfloor / 4 \rfloor = \lfloor n/16 \rfloor$ і $\lfloor \lfloor n/16 \rfloor / 4 \rfloor = \lfloor n/64 \rfloor$. Визначимо, скільки кроків треба зробити, щоб дійти до початкової умови. Оскільки після i-ї ітерації праворуч виявиться $T(\lfloor n/4^i \rfloor)$, ми дійдемо до $T(1)$, коли $\lfloor n/4^i \rfloor = 1$, тобто коли $i \geq \log_4 n$. Враховуючи, що $\lfloor n/4^i \rfloor \leq n/4^i$, ми можемо оцінити наш ряд спадною геометричною прогресією (плюс останній член, що відповідає $3^{\log_4 n}$ задачам обмеженого розміру):

$$T(n) \leq n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4 n}\Theta(1) \leq$$

$$\leq n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) = 4n + o(n) = O(n).$$

(Ми замінили кінцеву суму з не більш ніж $\log_4 n+1$ членів на суму нескінченного ряду, а також переписали $3^{\log_4 n}$ як $n^{\log_4 3}$, що є $o(n)$, оскільки $\log_4 3<1$.)

Процес підстановки співвідношення в себе можна зобразити у вигляді дерева рекурсії. Як це робиться на прикладі співвідношення

$$T(n) = 2T(n/2) + n^2$$

показано на рисунку 2. Для зручності припустимо, що n степінь двійки. Рухаючись від (а) до (г), ми поступово розвертаємо вираз для $T(n)$, використовуючи вираз для $T(n)$, $T(n/2)$, $T(n/4)$ і т. д. Тепер ми можемо обчислити $T(n)$ *шляхом знаходження суми* значень вершин з кожного рівня. На верхньому рівні маємо n^2, на другому: $(n/2)^2+(n/2)^2=n^2/2$, на третьому: $(n/4)^2+(n/4)^2+(n/4)^2+(n/4)^2=n^2/4$. Виходить спадна геометрична прогресія, сума якої відрізняється від її першого члена не більше ніж на постійний множник. Отже, $T(n)=\Theta(n^2)$.

На рисунку 3 показано складніший приклад – дерево для співвідношення

$$T(n) = T(n/3) + T(2n/3) + n.$$

Для простоти ми знову ігноруємо округлення. Тут сума значень на кожному рівні дорівнює n. Дерево обривається, коли значення аргументу

стають порівнянними з *1*. Для різних галузей це відбувається на різних рівнях, і найдовший шлях $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$ потребує біля $k=log_{3/2}n$ кроків (при такому k ми маємо $(2/3)^k n=1$). Тому $T(n)$ можна оцінити як $O(n \cdot logn)$.

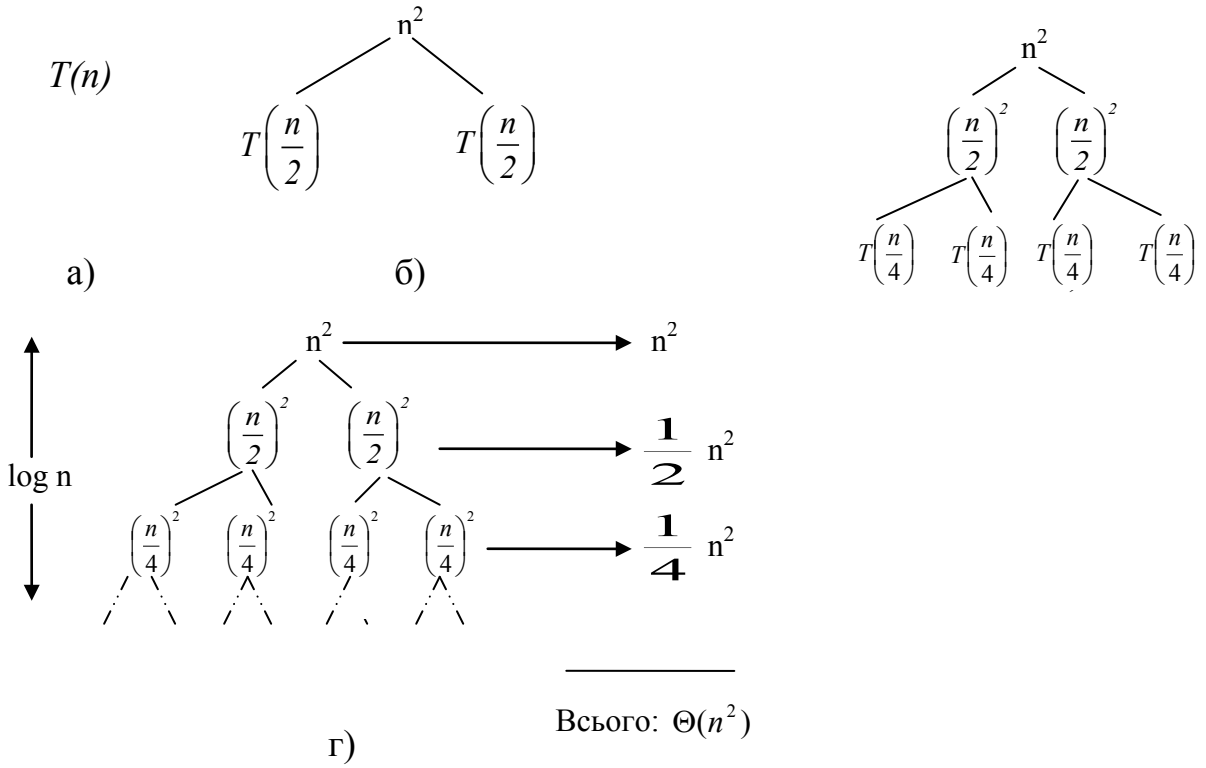


Рисунок 2 – Дерево рекурсії для співвідношення $T(n) = 2T(n/2) + n^2$

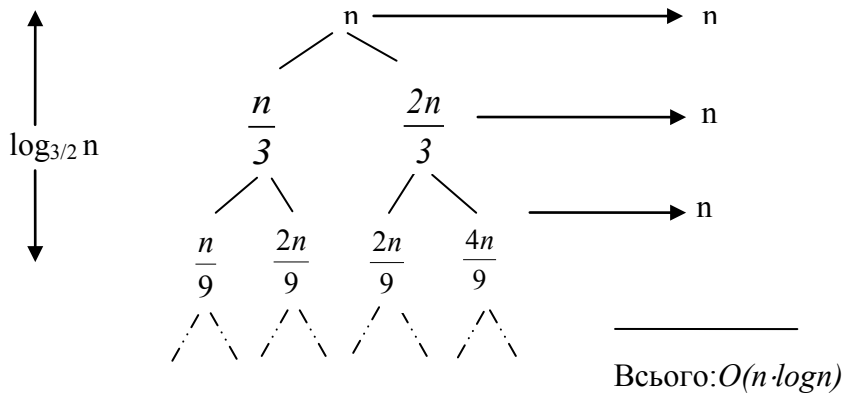


Рисунок 3 – Дерево рекурсії для співвідношення $T(n)=T(n/3)+T(2n/3)+n$

3. Оцініть рекурентні співвідношення

$T(n) = 9T(n/3) + n$, $T(n) = 3T(n/4) + n \cdot \log n$, $T(n) = T(2n/3) + 1$

за допомогою основної теореми про оцінювання рекурентних співвідношень.

Розв'язання

Перш ніж розв'язувати задачу, наведемо основну теорему про рекурентне оцінювання.

Основна теорема про рекурентне оцінювання [1, 5, 8].

Нехай $a \geq 1$ і $b>1$ – константи, $f(n)$ – функція, $T(n)$ визначено при невід'ємних n формулою

$$T(n) = aT(n/b) + f(n), \qquad (1)$$

де під n/b розуміється або $\lceil n/b \rceil$, або $\lfloor n/b \rfloor$. Тоді маємо три такі випадки:

1) якщо $f(n) = O(n^{\log_b a - \varepsilon})$ для деякого $\varepsilon>0$, то $T(n) = \Theta(n^{\log_b a})$;

2) якщо $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;

3) якщо $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для деякого $\varepsilon>0$ і якщо $af(n/b) \leq cf(n)$ для деякої константи $c<1$ і досить великих n, то $T(n) = \Theta(f(n))$.

Суть цієї теореми така. У кожному з трьох випадків ми порівнюємо $f(n)$ з $n^{\log_b a}$ і якщо одна з цих функцій зростає швидше інших, то вона і визначає порядок росту $T(n)$ (випадки 1 і 3). Якщо обидві функції одного порядку (випадок 2), то з'являється додатковий логарифмічний множник і відповіддю є формула $\Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.

Відзначимо важливі технічні деталі. У першому випадку недостатньо, щоб $f(n)$ була просто менша, ніж $n^{\log_b a}$. Нам потрібен «зазор» розміром n^{ε} для деякого $\varepsilon>0$. Точно так само в третьому випадку $f(n)$ повинна бути більше $n^{\log_b a}$ з запасом і до того ж задовольняти умову регулярності: $af(n/b) \leq cf(n)$.

Зауважимо, що три зазначених випадки не вичерпують усіх можливостей: може виявитися, наприклад, що функція $f(n)$ менша, ніж $n^{\log_b a}$, але зазор недостатньо великий для того, щоб скористатися першим твердженням теореми. Аналогічна «щілина» є і між випадками 2 і 3. Нарешті, функція може не мати властивості регулярності [1].

1. Оцінимо співвідношення $T(n) = 9T(n/3) + n$.

У цьому випадку $a=9$, $b=3$, $f(n)=n$, а $n^{\log_b a} = \Theta(n^2)$. Оскільки $f(n) = O(n^{\log_3 9 - \varepsilon})$ для $\varepsilon=1$, ми застосовуємо перше твердження теореми й робимо висновок, що $T(n) = \Theta(n^2)$.

2. Оцінимо співвідношення $T(n) = T(2n/3) + 1$.

Тут $a=1$, $b=3/2$, $f(n)=1$ і $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$, а отже, підходить випадок 2, оскільки $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, і ми одержуємо, що $T(n) = \Theta(\log n)$.

3. Для співвідношення $T(n) = 3T(n/4) + n \cdot \log n$ маємо $a=3$, $b=4$, $f(n) = n \cdot \log n$; при цьому $n^{\log_b a} = n^{\log_3 4} = O(n^{0,793})$. Зазор ($\varepsilon \approx 0,2$) є, залишається перевірити умову регулярності. Для досить великого n маємо $af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n \cdot \log n = cf(n)$ для $c=3/4$. Тим самим за третім твердженням теореми $T(n) = \Theta(n \cdot \log n)$.

Наведемо приклад, коли теорему застосувати не вдається. Нехай

$$T(n) = 2T(n/2) + n \cdot \log n .$$

Тут $a=2$, $b=2$, $f(n)=n \cdot \log n$, $n^{\log_b a} = n$. Видно, що $f(n)=n \cdot \log n$ асимптотично більша, ніж $n^{\log_b a}$, але зазор недостатній. Співвідношення

$$f(n) = (n \cdot \log n)/n = \log n$$

не оцінюється знизу величиною n^{ε} ні для якого $\varepsilon > 0$. Це співвідношення потрапляє у проміжок між випадками 2 та 3 і для нього можна одержати відповідь за формулою $T(n) = \Theta\left(\log^2 n\right)$.

І взагалі, якщо $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$, де $k \geq 0$, то співвідношення (1) приводить до $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.

Примітка. Під час оцінювання асимптотичної часової складності алгоритмів доцільно пам'ятати ряд формул. Наведемо деякі з них [1].

$b^{\log_b a} = a$; $e^{\ln(x)} = x$; $\log_b a^c = c \cdot \log_b a$; $a^{\log_b c} = c^{\log_b a}$;

$\log_b a = \dfrac{1}{\log_a b}$; $\log_b a = \dfrac{\log_c a}{\log_c b}$.

$\sum\limits_{k=1}^{n} k = \dfrac{1}{2} n(n+1) = \Theta(n^2)$; $\sum\limits_{k=0}^{n} k^2 = \dfrac{n(n+1)(2n+1)}{6}$;

$\sum\limits_{k=0}^{n} k^3 = \dfrac{x^{n+1}-1}{x-1}$, де $x \neq 1$; $\sum\limits_{k=1}^{n} \dfrac{1}{k} = \ln(n) + O(1)$.

Суму членів нескінченної геометричної прогресії, що спадає, можна знайти за формулою $\sum\limits_{k=1}^{n} \dfrac{1}{m^k} = \dfrac{A}{1-q}$, де $m>1$, A – перший член прогресії, q – основа прогресії.

Суму членів скінченної геометричної прогресії, що зростає $a+ar+ar^2+ +ar^{k-1}$ (де $r>1$) можна знайти за формулою $\sum\limits_{u=0}^{k-1} a \cdot r^u = \dfrac{a(r^k-1)}{r-1}$.

ТРЕТІЙ ТИП ЗАДАЧ

Наведіть графічну схему (або псевдокод) алгоритму сортування методом купи з коментарями; власний приклад роботи алгоритму та код програми (з коментарями) на обраній мові програмування.

Виконайте теоретичний аналіз складності алгоритму.

Виконайте практичні дослідження Вашої програми, для чого виміряйте (програмно) час її виконання у випадку сортування лінійного масиву чисел, що складається з 10^3, 10^4, $5 \cdot 10^4$, 10^5, $5 \cdot 10^5$, 10^6 елементів у таких трьох випадках: 1) масив вже відсортовано; 2) масив не відсортовано; 3) масив відсортовано у зворотному порядку. Для кожного з цих 3-х випадків наведіть таблицю та графік залежності часу виконання програми від розміру входу.

Порівняйте практичні результати з теоретичними розрахунками.

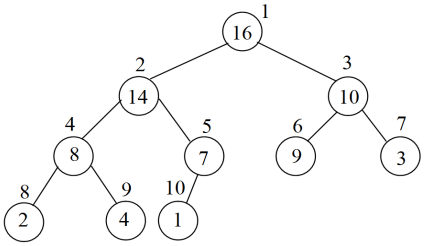
Зробіть висновки щодо складності програмування, часової та просторової складностей, стійкості, а також інших переваг, недоліків та особливостей цього алгоритму.

Розв'язання

Алгоритм сортування за допомогою купи потребує часу $\Theta\left(n \cdot \log n\right)$ для сортування n об'єктів та потребує додаткову пам'ять розміром $\Theta\left(1\right)$.

Структура даних, яку використовує алгоритм (вона називається двійковою купою) буває корисною і в інших ситуаціях. Особливо ефективно на її основі можна організовувати чергу з пріоритетами.

Двійковою купою називається масив з визначеними властивостями впорядкованості. Щоб сформулювати ці властивості, будемо розглядати масив як двійкове дерево (рисунок 4) [1].



а)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

б)

Рисунок 4

Кожна вершина дерева відповідає елементу масиву. Якщо вершина має індекс i, то її батько має індекс $\lfloor i/2 \rfloor$, вершина з індексом 1 є коренем, а її дочірні вершини мають індекси $2i$ та $2i+1$. Будемо вважати, що купа може не займати всього масиву, і тому будемо зберігати не тільки масив A і його довжину *lengthA*, а й спеціальний параметр *heapSizeA* (розмір купи), причому *обидві змінні повинні бути об'явлені глобально та heapSizeA* $\leq$ $\leq$ *lengthA*. Купа складається з елементів *A[1], ..., A[heapSizeA]*.

Додаткова інформація: замість запам'ятовування довжини масиву та розміру купи можна використовувати STL контейнер – vector, який замінить масив A. Прочитати про нього можна у джерелах [15, 16].

Рух по дереву здійснюється за допомогою функцій:

```
parent(i)                left(i)                right(i)
    return ⌊i / 2⌋;          return 2i;             return 2i + 1.
```

Купу можна розглядати як дерево (рисунок 4, а) або масив (рисунок 4, б). Усередині кожної вершини наведено її значення. Біля вершини наведено її індекс у масиві. Елемент *A[1]* є коренем дерева.

Додаткова інформація: у більшості комп'ютерів для виконання функцій left(i) та parent(i) можна використовувати команди лівого і правого зсуву, відповідно. Наприклад, new=left(i) можна замінити на new=i<<1. Процедура right(i) потребує лівого зсуву, після якого у молодший розряд записується одиниця, наприклад, new=((i<<1)|1).

Елементи, що зберігаються в купі, повинні характеризуватися **головною властивістю купи**: для кожної вершини *i*, крім кореня (при *2≤i≤heapSizeA),*

$$A[i] \le A[parent(i)]. \qquad (2)$$

Звідси випливає, що **значення нащадка не перевищує значення батька**. Таким чином, найбільший елемент дерева (або будь-якого піддерева) знаходиться у кореневій вершині цього дерева (або піддерева).

Висотою вершини дерева є висота піддерева з коренем у цій вершині (число ребер в найдовшому шляху збігається з початком у цій вершині вниз по дереву до листка). Висота дерева, таким чином, збігається з висотою його кореня. У дереві, що утворює купу, всі рівні (крім останнього) заповнені повністю. Тому висота цього дерева дорівнює $\Theta(\log n)$, де n – число елементів купи. Як побачимо нижче, час роботи основних операцій над купою пропорційний висоті дерева і, відповідно, становить $\Theta(\log n)$.

Перерахуємо основні операції над купою [1]:

- Процедура heapify (vector <int> &A, int cur) дозволяє підтримувати головні властивості купи. Час її роботи становить $\Theta(\log n)$.
- Процедура buildHeap (vector <int> &A) будує купу з вихідного (невідсортованого) масиву. Час її роботи $\Theta(n)$.
- Процедура heapSort (vector <int> &A) сортує масив, не використовуючи допоміжної пам'яті. Час її роботи $\Theta(n \cdot \log n)$.

Додаткова інформація: vector <int> &A – передавання масиву в функцію за посиланням. Це дозволяє змінювати переданий масив. Прочитати про це детальніше можна у джерелах [17, 18].

Збереження головної властивості купи

Процедура heapify – важливий метод роботи з купою. Вважається, що піддерева з коренями left(*i*) і right(*i*) вже мають головні властивості. Процедура переміщує елементи піддерева з вершиною *i*, після чого воно буде мати головну властивість. Ідея така: якщо головна властивість не виконана для вершини *i*, то її потрібно поміняти із старшим із її дочірніх вершин і т. д., доти, доки елемент A[*i*] не «завантажиться» до потрібного місця.

Процедура heapify наведена у прикладі 1. У рядках 3 – 7 змінна *largest* набуває значення, що дорівнює індексу найбільшого з елементів $A[i]$, $A[\text{left}(i)]$, $A[\text{right}(i)]$. Якщо $largest=i$, то елемент $A[i]$ вже «завантажився» до потрібного місця, і робота процедури закінчена. Інакше процедура міняє місцями $A[i]$ та $A[largest]$ (що забезпечує виконання властивості (2) у вершині i, але, можливо, порушує цю властивість у вершині *largest*) і рекурсивно викликає себе для вершини *largest* (рядок 10), щоб виправити можливі порушення.

Приклад 1. Процедура збереження головної властивості купи (псевдокод).

```
heapify(A, i)
1. l:= left(i)
2. r:= right(i)
3. if (l ≤ heapSizeA) and  (A[l] > A[i])
4.   then largest:= l
5.   else largest:= i
6. if (r ≤ heapSizeA) and (A[r] > A[largest])
7.   then largest:= r
8. if largest ≠ i
9.   then виконати обмін A[i] ↔ A[largest]
10.       heapify(A, largest)
```

Приклад виконання процедури наведено на рисунку 5. Робота процедури heapify (A, 2) при $heapSizeA$=10 (а). У вершині i=2 головна властивість порушена. Щоб відновити її, необхідно поміняти місцями $A[2]$ і $A[4]$. Після цього (б) головна властивість порушується у вершині з індексом 4. Рекурсивний виклик процедури heapify (A, 4) відновлює головну властивість у вершині з індексом 4 шляхом перестановки $A[4] \leftrightarrow A[9]$ (в). Після цього головна властивість виконана для усіх вершин, тому процедура heapify (A, 9) вже нічого не виконує.

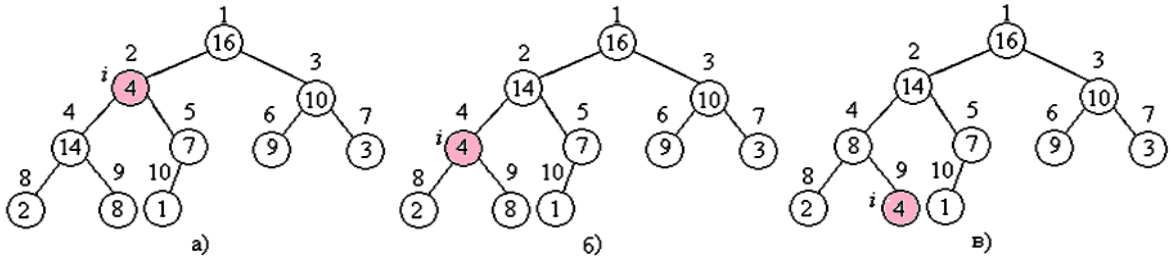


Рисунок 5 – Приклад виконання процедури heapify

Оцінимо час роботи процедури heapify. На кожному кроці потрібно провести $\Theta(1)$ дій, не враховуючи рекурсивного виклику. Нехай $T(n)$ – час роботи для піддерева, що містить n елементів. Якщо піддерево з коренем i складається з n елементів, то піддерева з коренями $\text{left}(i)$ та $\text{right}(i)$ містять

не більше ніж по $2n/3$ елементів кожен (найгірший випадок, коли останній рівень в піддереві заповнений наполовину). Отже, $T(n) \le T(2n/3) + \Theta(1)$.

З основної теореми про рекурентні оцінки (випадок 2) отримуємо $T(n) = \Theta(\log n)$. Цю ж оцінку можна отримати так: на кожному кроці ми опускаємось по дереву на один рівень, враховуючи, що висота дерева це $\Theta(\log n)$.

Побудова купи

Нехай дано масив $A[1...n]$, який ми хочемо перетворити на купу, переставивши його елементи. Для цього можна використати процедуру heapify, застосовуючи її по черзі до усіх вершин, починаючи з нижніх. Оскільки вершини з номерами $\lfloor n/2 \rfloor + 1$, ..., n є листям, піддерева з цими вершинами задовольняють головну властивість. Для кожної з вершин, що залишились, у порядку зменшення індексів, використовуємо процедуру heapify. Порядок обробки вершин гарантує, що кожен раз умови виклику процедури (виконання головної властивості для піддерев) будуть виконані [1].

Процедуру buildHeap(A) наведено у прикладі 2.

Приклад 2. Процедура побудови купи.

```
buildHeap (A)
1.  heapSizeA := lengthA
2.  for i:= ⌊size(a)/2⌋ downto 1
3.     heapify(A, i)
```

Приклад роботи цієї процедури наведено на рисунку 6, де показано стан даних перед кожним викликом процедури heapify (рядок 3).

Час роботи процедури buildHeap не перевищує $\Theta(n \cdot \log n)$, оскільки процедура heapify викликається $\Theta(n)$ разів, а кожне її виконання потребує часу $\Theta(\log n)$. Цю оцінку можна покращити. Для цього потрібно врахувати, що час роботи процедури heapify залежить від висоти вершини, для якої вона викликається. Оскільки число вершин висотою h у купі з n елементів не перевищує $\lceil n / 2^{h+1} \rceil$, а висота усієї купи не перевищує $\lfloor \log n \rfloor$, час роботи процедури buildHeap не перевищує $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{n+1}} \right\rceil \Theta(h) = \Theta\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$. Припускаючи, що $x = 1/2$, отримуємо верхню оцінку для суми у правій частині $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$ і час роботи процедури buildHeap: $\theta\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = \theta(n)$.

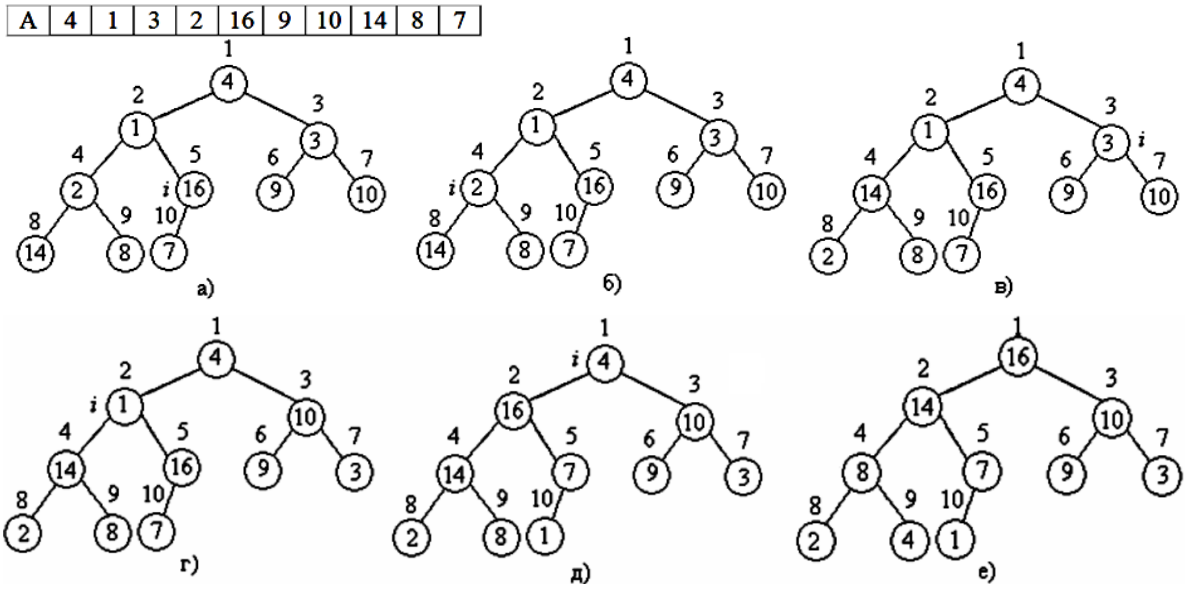


Рисунок 6 – Робота процедури buildHeap

Алгоритм сортування за допомогою купи

Алгоритм сортування за допомогою купи складається з двох частин. Спочатку викликається процедура buildHeap, після виконання якої масив стає купою (приклад 3). Ідея другої частини така: максимальний елемент масиву тепер знаходиться у корені дерева $A[1]$. Його потрібно поміняти з елементом $A[n]$, зменшити розмір купи на 1 і відновити головну властивість у кореневій вершині (оскільки піддерева з коренями left(*1*) та right(*1*) не втратили головної властивості купи, це можна зробити за допомогою процедури heapify. Після цього у корені буде знаходитися максимальний з елементів, що залишився. Так робиться доти, поки у купі не залишиться усього один елемент [1, 8].

Приклад 3. Процедура сортування масиву.

```
heapsort(A)
1.  buildHeap(A)
2.  for i ← lengthA downto 2
3.      замінити A[1]↔ A[i]
4.      heapSizeA ← heapSizeA - 1
5.      heapify(A, 1)
```

Роботу другої частини алгоритму наведено на рисунку 7. Тут зображені стани купи перед кожним виконанням циклу for (рядок 2), а заштриховані елементи вже не входять до купи. Час роботи процедури heapsort становить $\Theta(n \cdot log\, n)$. Дійсно, перша частина (побудова купи) потребує часу $\Theta(n)$, а кожне з *n-1* виконань циклу for потребує час $\Theta(log\, n)$.

Насамкінець зазначимо, що практично алгоритм сортування за допомогою купи не є найшвидшим, як правило, краще швидке сортування (Quick sort) [1].

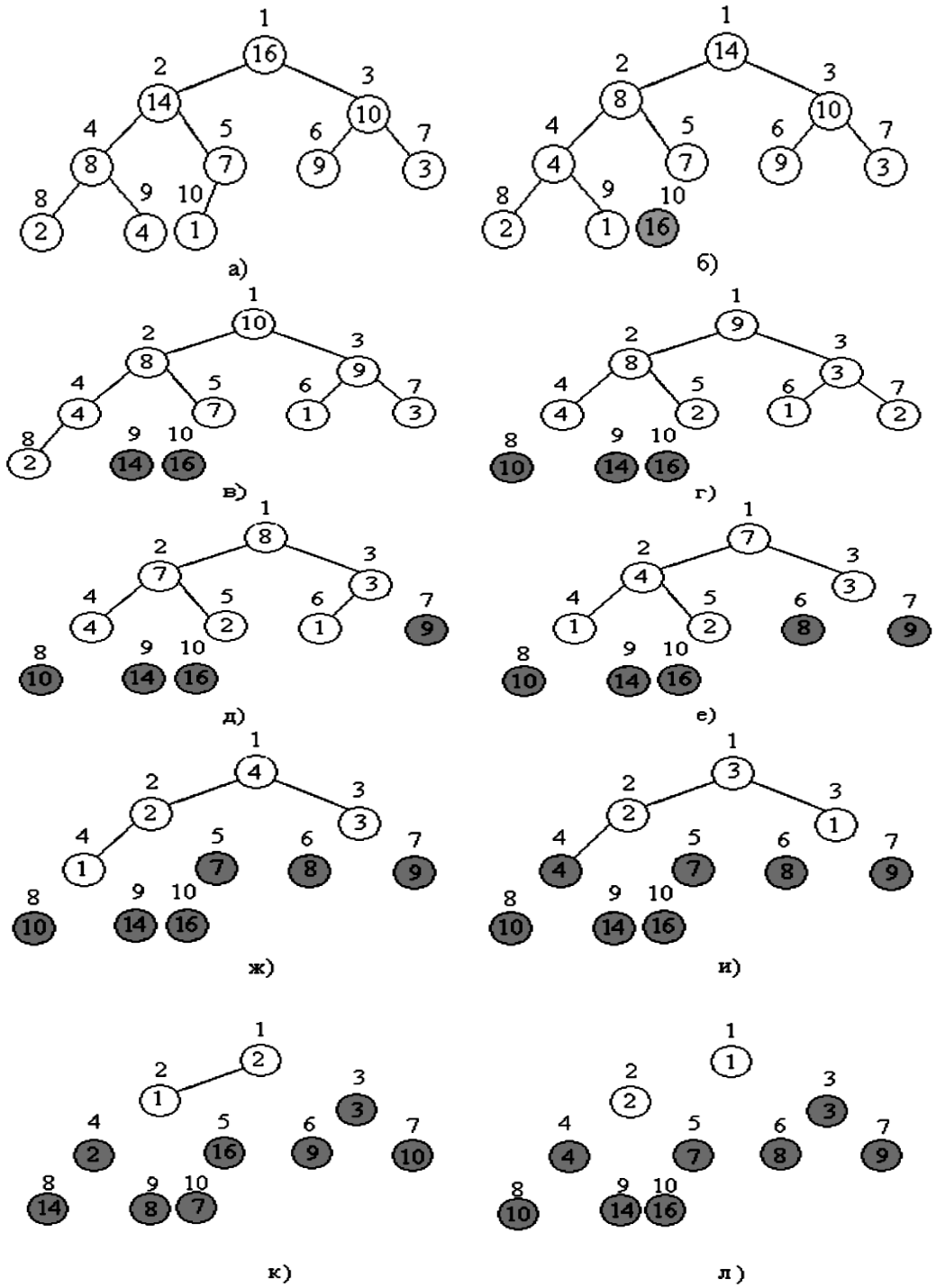


Рисунок 7 – Приклад сортування за допомогою купи

Лістинг програми, сортування методом купи наведено у прикладі 4.

Приклад 4. Лістинг програми сортування методом купи.

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <ctime>

using namespace std;
```

```
    vector <int> A;                          // масив
    int lengthA, heapSizeA;                  // довжина масиву та розмір ку-
пи

    int parent(int i);
    int left(int i);
    int right(int i);
    void heapify(vector <int> &A, int i);
    void buildHeap(vector <int> &A);
    void heapSort();

    int parent(int i)                         // Знаходження «батька» верши-
ни
    {
      return i / 2;
    }
    int left(int i)                           // Знаходження лівого «сина»
    {
      return 2 * i;
    }
    int right(int i)                         // Знаходження правого «сина»
    {
      return 2 * i + 1;
    }
    void heapify(vector <int> &A, int i)    // Знаходження найбільшого еле-
мента
    {
      int l = left(i);
      int r = right(i);
      int largest;
      if (l <= heapSizeA && A[l] > A[i])// Якщо лівий «син» більше «батька»
      {
            largest = l;                // Запам'ятаємо його індекс
      }
      else
      {
            largest = i;                // Інакше індекс поточного
      }
      if (r <= heapSizeA && A[r] > A[largest]) // Якщо правий син більше
      {                                        // найбільшого з попередніх
            largest = r;                       // запам'ятаємо його індекс
      }
      if (largest != i)
      {
            swap(A[i], A[largest]);       // Ставимо у «батька» найбільше
            heapify(A, largest);          // значення з трійки
      }
    }
    void buildHeap(vector <int> &A)           // Будуємо дерево
    {


    heapSizeA = lengthA;                 // Спочатку розмір купи = масиву
      for (int i = lengthA / 2; i >= 1; i--)
      {
            heapify(A, i);                // Будуємо дерево
      }
    }
    void heapSort()                             // Сортування купою
    {
      buildHeap(A);                         // Будуємо дерево
      for (int i = lengthA; i >= 2; i--)
      {
```

```
        swap(A[i], A[1]);              // Беремо найбільший елемент
        heapSizeA--;                   // Записуємо у кінець масиву
        heapify(A, 1);                 // Зменшуємо розмір купи
    }                                   // Та знову відновлюємо купу

}

int countElements;

void main()
{
    setlocale(LC_ALL, "Russian");
    cout << "Введіть кількість елементів масиву:\n";
    cin >> countElements;             // Зчитуємо кількість елементів

    A.resize(countElements + 1);
    lengthA = countElements;

    printf("Введіть елементи масиву:\n");
    for (int i = 1; i <= countElements; i++)
    {
        cin >> A[i]$                       // Зчитуємо масив
    }

    heapSort();                            // Викликаємо сортування

    cout << "Відсортований масив:\n";
    for (int i = 1; i <= countElements; i++)
    {
        cout << A[i] << " ";              // Виводимо відсортований масив
    }
    cout << endl;

}
```

ЧЕТВЕРТИЙ ТИП ЗАДАЧ

Поясніть, як за допомогою динамічного програмування розв'язати задачу обчислення деякого числа f_i ($3 \le i \le 46$) у послідовності Фібоначчі, де $f_1=1, f_2=1, f_i=f_{i-1}+f_{i-2}$. Наведіть загальний алгоритм розв'язання задачі та програмний код його реалізації.

Перш ніж наводити процес розв'язання задачі потрібно зауважити, що динамічне програмування (ДП) застосовується для розв'язання деяких комбінаторних задач та задач оптимізації (останні передбачають мінімізацію або максимізацію певної цільової функції), тобто до задач, що мають певну властивість – властивість субоптимальності їх підзадач. ДП – це такий спосіб розв'язання складних задач, що передбачає їх розбиття на простіші підзадачі, де останні повинні мати оптимальну підструктуру. Отже, наша задача виглядає як набір підзадач, що перекриваються (тобто, деякі з цих підзадач повторюються) і характеризуються складністю дещо меншою за вихідну. У такому випадку час обчислень за допомогою ДП (порівняно з «наївними» методами), можна значно скоротити.

Ідея ДП полягає в тому, що для розв'язання поставленої задачі, потрібно розв'язати її окремі частини (підзадачі), а потім об'єднати розв'язки,

отримавши шуканий розв'язок. Часто буває так, що багато з вищезгаданих підзадач однакові. ДП передбачає, що розв'язання кожної підзадачі потрібно виконати лише один раз, скоротивши тим самим кількість обчислень. Це особливо корисно у випадках, коли кількість задач, що повторюються, дуже велика. Реалізація такого однократного розв'язання підзадач полягає у запам'ятовуванні результатів розв'язку підзадач, які можуть повторно зустрітися надалі.

Прикладами оптимізаційних задач, що можна розв'язати за допомогою ДП є, наприклад, вибір маршруту оптимальної довжини (мінімізація довжини маршруту); складання плану оптимального виробництва (є виробництво ряду груп товарів; мета – максимізувати прибуток); визначення біржового портфеля (максимізація прибутку); заміна деякого обладнання (мінімізація витрат); різні ігри (максимізація виграшу) тощо. Прикладами комбінаторних задач є задачі з аналізом графів та дерев (скільки є графів, дерев, що мають певну властивість; визначення способів дістатися з пункту *A* до пункту *B*; визначення способів повернути певну суму грошей тощо.

Загальні кроки розв'язання задач методом ДП такі [1]:

1) описати будову оптимальних рішень;
2) виписати рекурентне співвідношення, що пов'язує оптимальні значення параметра для підзадач;
3) рухаючись знизу вгору, обчислити оптимальне значення параметра для підзадач;
4) користуючись отриманою інформацією, побудувати оптимальне рішення.

Тепер подивимось, як за допомогою ДП можна розв'язати цю задачу.

Розв'язання

Перше, що приходить на думку, це розв'язати задачу так, як наведено нижче у прикладі 5.

Хоча таке розв'язання просте та очевидне, воно має великий недолік: час роботи програми має дуже велику швидкість зростання і для $i>40 \ldots 50$ комп'ютер вже буде «задумуватись» (залежно від його потужності). Це відбуватиметься тому, що для обчислення, наприклад, *f(50)* спочатку обчислюється *f(49)* та *f(48)*. Причому обчислення *f(48)* відбувається наново, без врахування того, що воно вже обчислювалося, під час визначення *f(49)* і т. д. Тобто значення функції при одному і тому ж значенні аргументу обчислюється дуже багато разів. Якщо виключити такі повторні розрахунки, то обчислення стане набагато ефективнішим. Проте, для цього потрібно виділити вектор-масив розміру n_{max} (де n_{max} – максимально можливий номер для послідовності Фібоначчі), наприклад *Mf*, в якому зберігатимуться значення нашої функції. Під час ініціалізації масив потрібно заповнити числами, що гарантовано не можуть бути значеннями нашої функції, наприклад нулями.

Приклад 5.

```
Function f(i: integer): longint;
    1)  if (i = 1) or (i = 2)
    2)     then f:= 1
    3)     else f:= f(i - 1) + f(i - 2)
```

Тепер, за необхідності обчислити деяке конкретне значення, програма дивиться, чи не вираховувалось воно раніше, і якщо так, то бере готовий результат. Псевдокод такої функції наведений у прикладі 6.

Приклад 6.

```
Function f(i: integer): longint;
       4)  if Mf[i] = 0
       5)           then
 6)     if (i = 1) or (i = 2)
 7)      then Mf[i]:= 1
 8)      else Mf[i]:= f(i - 1) + f(i - 2)
 9)   f:= Mf[i]
```

Цей псевдокод можна ще спростити та пришвидшити швидкодію роботи, якщо прибрати рекурсію. У цьому випадку це можна у циклі, як наведено у прикладі 7.

Приклад 7.

```
Read(n)
Mf[1]:= 1; Mf[2]:= 1;
For i:= 3 to n do
     Mf[i]:= Mf[i-1] + Mf[i-2]
```

Програму на мові С++, що дозволяє реалізувати вищенаведений підхід, наведено у прикладі 8.

Приклад 8. Лістинг програми обчислення чисел Фібоначчі методом ДП.

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;
int n;
vector <int> Mf;
void main()
{
    cin >> n;// Зчитуємо номер елемента, що потрібно вивести
    Mf.resize(n + 1); // Задаємо розмір масиву. Беремо n + 1, оскільки
останній індекс який нам потрібен n,
    Mf[1] = 1; // Задаємо базу динаміки
    Mf[2] = 1;
    for (int i = 3; i <= n; i++)
    {
```

```
                Mf[i] = Mf[i - 1] + Mf[i - 2]; // Рахуємо всі значення функ-
ції від 3-го до n-го елементу включно
        }
        cout << Mf[n] << endl; // Виводимо відповідь
    }
```

П’ЯТИЙ ТИП ЗАДАЧ

Нехай дані *n* заявок на проведення занять в одній і тій же аудиторії. Два різних заняття не можуть перетинатися у часі. У кожній заявці зазначені початок і кінець заняття (s_i і f_i для *i*-ї заявки), тобто вона характеризується проміжком (s_i, f_i) так, що кінець одного заняття може збігатися з початком іншого, і це не вважається перетином. Проте різні заявки можуть перетинатися, і тоді можна задовольнити тільки одну з них. Формально кажучи, заявки з номерами *i* і *j* сумісні (compatible), якщо інтервали $[s_i, f_i)$ і $[s_j, f_j)$ не перетинаються (іншими словами, якщо $f_i \le s_j$ або $f_j \le s_i$). Задача про вибір заявок (activity-selection problem) полягає у тому, щоб задовольнити максимальну кількість сумісних одна з одною заявок [1].

Перш ніж наводити розв’язання задачі, наведемо деякі теоретичні аспекти «жадібних» («скупих») алгоритмів (ЖА) [1, 5, 8].

Для багатьох задач оптимізації є більш прості та швидкі алгоритми, ніж ДП, які називають ЖА (greedy algorithms). Такий алгоритм робить на кожному кроці локально оптимальний вибір, з розрахунком, що підсумкове розв’язання також виявиться оптимальним. Це не завжди так, але для багатьох задач ЖА дійсно дають оптимум. Так, наприклад, вони дозволяють ефективно розв’язати задачу про мінімальне покриваюче дерево, алгоритм Дейкстри для пошуку найкоротших шляхів з цієї вершини, «жадібна» евристика, задачі про покриття множин, мінімальні покриваючі дерева тощо.

Як довідатися, чи дасть ЖА оптимум для певної задачі? Загальних рецептів тут немає, але *існують дві особливості, характерні для задач, розв’язуваних ЖА – це принцип «жадібного» вибору й властивість оптимальності для підзадач*. Розглянемо ці особливості детальніше.

Принцип «жадібного» вибору

Говорять, що до оптимізаційної задачі застосовують *принцип «жадібного» вибору* (greedy-choice property), якщо послідовність локально оптимальних («жадібних») виборів дає глобально оптимальне розв’язання. Розходження між ЖА та ДП можна пояснити так: на кожному кроці ЖА бере «найжирніший шматок», а потім уже намагається зробити найкращий вибір серед тих, що залишилися, які б вони не були; алгоритм ДП приймає розв’язання, прорахувавши заздалегідь наслідки для всіх варіантів.

Як довести, що ЖА дає оптимальне розв’язання? Це не завжди тривіально, але в типовому випадку таке доведення базується на тому, що: спочатку ми доводимо, що «жадібний» вибір на першому кроці не закриває шляхи до оптимального розв’язання: для всякого розв’язання є інше, пого-

джене з «жадібним» вибором і не гірше першого; потім ми показуємо, що підзадача, що виникає після «жадібного» вибору на першому кроці, аналогічна вихідній, і міркування завершується за індукцією.

Оптимальність для підзадач

Розв'язувані за допомогою ЖА задачі мають властивість *оптимальності для підзадач* (have optimal substructure): оптимальне розв'язання всієї задачі містить у собі оптимальні розв'язання підзадач. (Із цією властивістю ми вже зустрічалися, кажучи про ДП.)

ЖА або ДП?

І ЖА, і ДП ґрунтуються на властивості оптимальності для підзадач, тому може виникнути спокуса застосувати ДП у ситуації, де вистачило б ЖА, або навпаки, застосувати ЖА до задачі, у якій він не дасть оптимуму. Проілюструємо можливі пастки на прикладі двох варіантів класичної оптимізаційної задачі.

Дискретна задача про рюкзак (0-1 knapsack problem). Нехай злодій пробрався на склад, на якому зберігається n речей. Річ з номером i коштує v_i доларів і важить w_i кілограмів (v_i і w_i – цілі числа). Злодій хоче вкрасти товару на максимальну суму, причому максимальна вага, що він може винести у рюкзаку, дорівнює W (число W теж ціле). Що він повинен покласти у рюкзак?

Неперервна задача про рюкзак (fractional knapsack problem) відрізняється від дискретної тим, що злодій може дробити крадені товари на частини й укладати у рюкзак ці частини, а не обов'язково брати речі цілими (якщо у дискретній задачі злодій має справу із золотими злитками, то у неперервній – із золотим піском).

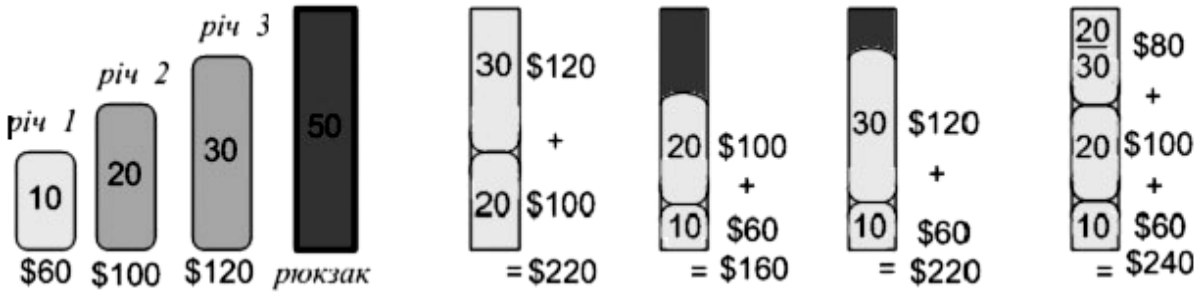
Обидві задачі про рюкзак мають властивість оптимальності для підзадач. Справді, розглянемо дискретну задачу. Вийнявши річ номер j з оптимально завантаженого рюкзака, отримаємо розв'язання задачі про рюкзак з максимальною вагою $W - w_j$ і набором з n - 1 речей (усі речі, крім j-ї). Аналогічне міркування проходить і для неперервної задачі: вийнявши з оптимально завантаженого рюкзака, в якому лежить w кілограмів товару номер j, одержимо оптимальне розв'язання неперервної задачі, в якій максимальна вага дорівнює $W - w$ (замість W), а кількість j-го товару дорівнює $w_j - w$ (замість w_j).

Хоча дві задачі про рюкзак і схожі, ЖА дає оптимум у неперервній задачі про рюкзак і не дає в дискретній. Справді, розв'язання неперервної задачі про рюкзак за допомогою ЖА виглядає так. Обчислимо ціни (розраховуючи на кілограм) всіх товарів (ціна товару номер $г$ дорівнює v_i/w_i). Спочатку злодій бере максимум найдорожчого товару; якщо весь цей товар скінчився, а рюкзак не заповнений, злодій бере наступний за ціною товар, і так далі, поки не набере ваги W. Оскільки товари потрібно спочатку відсортувати за цінами, на що піде час $O(n \log n)$, час роботи описаного алгоритму буде $O(n \log n)$, що дійсно дає оптимум.

Щоб переконатися у тому, що аналогічний ЖА не зобов'язаний давати оптимум у дискретній задачі про рюкзак, подивіться на рисунку 8, а. Вантажопідйомність рюкзака 50 кг, на складі є три речі, що важать 10, 20 і 30 кг і вартістю 60, 100 і 120 доларів, відповідно. Ціна їх розраховуючи на одиницю ваги дорівнює 6, 5 і 4. ЖА для початку покладе у рюкзак річ номер 1; однак оптимальне розв'язання включає предмети номер 2 і 3.

Для неперервної задачі з тими ж вихідними даними ЖА, що пропонує почати з товару номер 1, дає оптимальне розв'язання (рисунок 8, в). У дискретній задачі така стратегія не спрацьовує: поклавши в рюкзак предмет номер 1, злодій втрачає можливість заповнити рюкзак «під зав'язку», а порожнє місце в рюкзаку знижує ціну накраденого, розраховуючи на одиницю ваги. Тут, щоб вирішити, чи класти цю річ у рюкзак, потрібно зрівняти розв'язання двох підзадач: коли ця річ свідомо лежить у рюкзаку, і коли цієї речі в рюкзаку свідомо немає. Тим самим дискретна задача про рюкзак породжує множину підзадач, що перекриваються – типову ознаку задач, для яких ефективно застосовувати ДП.

Оптимальний вибір – друга та третя речі; якщо покласти у рюкзак першу, то вибір оптимальним не буде, хоча саме вона дорожча всіх, розраховуючи на одиницю ваги (рисунок 8, в). Для неперервної задачі про рюкзак з тими ж вихідними даними вибір товарів у порядку зменшення ціни на одиницю ваги буде оптимальний.



а) б) в)

Рисунок 8 – У дискретній задачі про рюкзак «жадібна» стратегія може не спрацювати (а); злодій повинен вибрати дві речі із трьох для того, щоб їхня сумарна вага не перевищила 50 кг (б)

Основні кроки розв'язання задачі за допомогою ЖА [1]

1. Привести задачу оптимізації до вигляду, коли після зробленого вибору залишається розв'язати тільки одну підзадачу.
2. Довести, що завжди існує таке оптимальне розв'язання вихідної задачі, яку можна отримати шляхом «жадібного» вибору, так що цей вибір завжди є допустимим.
3. Показати, що після «жадібного» вибору залишається підзадача, яка має таку властивість, що об'єднання оптимального розв'язання підзадачі зі зробленим «жадібним» вибором приводить до оптимального розв'язання вихідної задачі.

Розв'язання

Припустимо, що заявки впорядковані у порядку зростання часу закінчення:

$$f_i \le f_2 \le \ldots \le f_n.$$

Якщо це не так, то можна відсортувати їх за час *O(nlogn)*; заявки з однаковим часом закінчення розташовуємо у довільному порядку. Тоді псевдокод ЖА виглядає як наведено у прикладі 6 (*order – масив пар, в якому другий елемент пари – елемент умовного масиву s (початок відрізку часу в заявці), а перший – елемент масиву f (кінець відрізку часу)*) [1].

Приклад 9.

```
greedyActivitySelector (order)
1 n:= lengthOrder
2 A:= {1}
3 j:= 1
4 for i:= 2 to n
5         if order[i].second ≥ order[j].first
6     then A:= A∪{i}
7                j:= i
8 return A
```

Додаткова інформація: детальніше про пари можна почитати у джерелі [19].

Робота цього алгоритму показана на рисунку 9. Множина A складається з номерів обраних заявок, j – номер останньої з них; при цьому

$$order_j.first = \max\{order_k.first : k \in A\},$$

оскільки заявки відсортовані за зростанням часу закінчення.

Спочатку A містить заявку номер 1, і $j = 1$ (рядки 2 – 3). Далі (цикл у рядках 4 – 7) шукається заявка, що починається не раніше закінчення заявки номер j. Якщо така знайдена, вона включається в множину A і змінній j присвоюється її номер (рядки 6 – 7).

Алгоритм greedyActivitySelector потребує всього лише $\Theta(n)$ кроків (не враховуючи попереднього сортування). Як і властиво «жадібному» алгоритму, на кожному кроці він робить вибір так, щоб вільний час, що залишається, був максимальним.

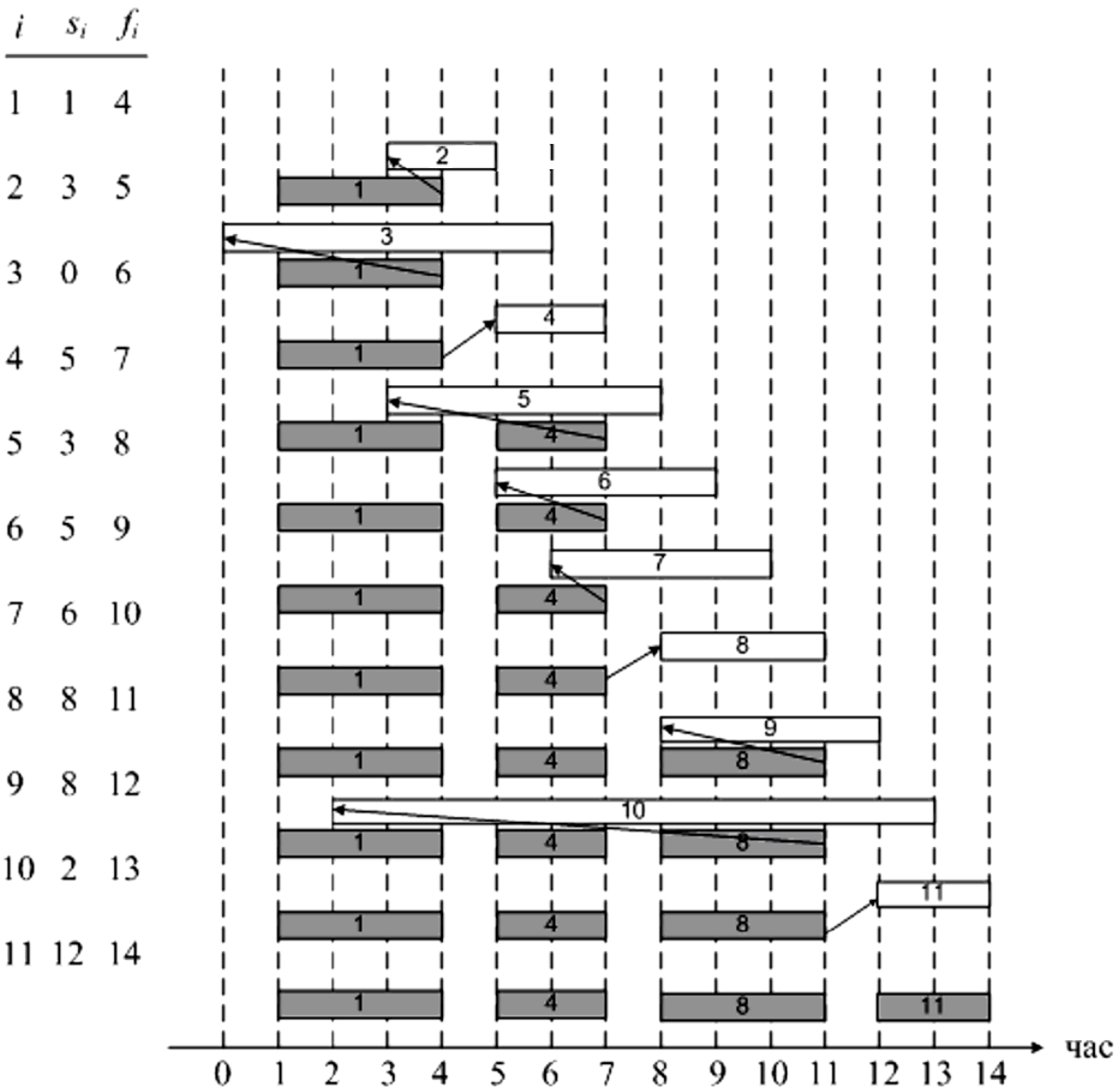


Рисунок 9 – Робота алгоритму greedyActivitySelector для 11 заявок (таблиця зліва)

Правильність алгоритму

Не для усіх задач «жадібний» алгоритм дозволяє отримати оптимальне розв'язання, але для нашої дозволяє. Переконаємося у цьому.

Теорема 1. Алгоритм greedyActivitySelector дає набір з найбільшої можливої кількості спільних заявок.

Доведення. Нагадаємо, що заявки відсортовані за зростанням часу закінчення. Насамперед доведемо, що існує оптимальне розв'язання задачі про вибір заявок, що містить заявку номер 1 (із найранішим часом закінчення).

Справді, якщо в якійсь оптимальній множині заявок заявка номер 1 не втримується, то можна замінити в ній заявку із найранішим часом закінчення на заявку номер 1, що не зашкодить спільності заявок (тому що заявка номер 1 закінчується раніше, ніж попередня, і ні з чим перетнутися не може) і не змінить їхньої загальної кількості. Отже, можна шукати оптима-

льну множину заявок *A* серед утримуючих заявку номер 1: існує оптимальне розв'язання, що починається з «жадібного» вибору.

Після того, як ми домовилися розглядати тільки набори, що містять заявку номер 1, усі неспільні з нею заявки можна викинути, і задача зводиться до вибору оптимального набору заявок із множини заявок, що залишилися (спільних із заявкою номер 1). Інакше кажучи, ми звели задачу до аналогічної задачі з меншою кількістю заявок. Розмірковуючи за індукцією, одержуємо, що, роблячи на кожному кроці «жадібний» вибір, ми прийдемо до оптимального розв'язання [1].

Кожний рядок на рисунку відповідає одному проходу циклу в рядках 4 – 7. Сірі заявки вже включені в *A*, біла зараз розглядається. Якщо лівий кінець білого прямокутника лівіше правого кінця правого сірого (стрілка йде вліво), то заявка відкидається, у протилежному випадку вона додається до *A*.

Лістинг програми мовою C++, для нашої задачі, що реалізує ЖА наведено у прикладі 10.

Приклад 10. Лістинг програми про вибір заявок.

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

using namespace std;

vector <int> greedyActivitySelector(vector <pair <int, int> > &order)
{
  int n = order.size();          // Запам'ятовуємо кількість елементів
  vector <int> A;
  A.push_back(1);                // Додаємо до відповіді перший елемент
  int j = 1;
  for (int i = 2; i < n; i++)
  {
   if (order[i].second >= order[j].first)// Якщо початок i-го більше
        {                                 // кінця останнього доданого(j)
         A.push_back(i)                  // Додаємо його до відповіді
              j = i;                    // Запам'ятовуємо його як останній
        }                               // доданий відрізок
  }
  return A;                            // Повертаємо масив-відповідь
}
void main()
{
  int n;
  setlocale(LC_ALL, "Russian");


  cout << "Введіть кількість елементів масиву:\n";
  cin >> n;                             // Зчитуємо кількість елементів
        vector < pair<int, int> > order(n + 1);  // Об'являємо масив
  printf("Введіть пари чисел такі, що перший елемент - початок
відрізку(l),\na другий - кінець(r):\n");
  for (int i = 1; i <= n; i++)
  {
```

```
            cin >> order[i].second >> order[i].first;  // Зчитуємо відрізки
      }

      sort(order.begin(), order.end()); // Сортуємо масив за кінцями

      vector <int> ans = greedyActivitySelector(order); // Викликаємо ЖА

      cout << "Кількість відрізків максимального покриття та самі
відрізки:\n";
      cout << ans.size() << endl;                  // Виводимо відповідь
      for (int i = 0; i < ans.size(); i++)
      {
            cout << order[ans[i]].second << " " << order[ans[i]].first <<
endl;
      }
}
```

ВИМОГИ ДО ОФОРМЛЕННЯ КОНТРОЛЬНОЇ РОБОТИ

Контрольна робота з дисципліни «Теорія алгоритмів» складається з шести завдань (одного теоретичного питання та п’яти завдань практичного характеру).

Контрольна робота оформлюється на листках формату А4 і спочатку містить відповіді на теоретичні питання, а потім – розв’язання задач. Усі відповіді наводяться державною мовою.

Відповідаючи на теоретичні питання, необхідно дотримуватися таких умов: матеріал повинен викладатися логічно, містити необхідні роз’яснення до формул, які зустрічаються. Відповіді не мають переписуватися з підручників, а мають відображати творчий підхід студента до викладення теоретичного матеріалу, що вивчається.

При поясненні задач необхідно давати пояснення розв’язків щодо формул та співвідношень, які використовуються при їх розв’язанні.

В кінці контрольної роботи наводиться список літературних джерел, з яких безпосередньо взяті цитати у вигляді окремих речень, формул, таблиць. У тексті роботи кожне посилання на літературне джерело дають у квадратних дужках, в яких вказується порядковий номер джерела за списком, наприклад, [7].

СПИСОК ЛІТЕРАТУРИ

1. Алгоритмы: построение и анализ / [Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.]. – М. : Издательский дом «Вильямс», 2013. – 1328 с.
2. Арсенюк І. Р. Теорія алгоритмів : навчальний посібник / Арсенюк І. Р., Колодний В. В., Яровий А. А. – Вінниця : ВНТУ, 2006. – 150 с.
3. Ліщина, Н. М. Теорія алгоритмів : конспект лекцій для студентів напряму підготов. «Комп'ютерні науки» денної форми навч. / Ліщина Н. М. – Луцьк : Луцький НТУ, 2015. – 72 с.
4. Горлова Т. М. Теорія алгоритмів. Конспект лекцій для студентів напряму підготовки 6.050101 «Комп'ютерні науки» денної та заочної форм навчання / Горлова Т. М., Бобрівник К. Є., Ліманська Н. В. – К. : НУХТ, 2015. – 95 с.
5. Кормен Т. Х. Алгоритмы. Вводный курс / Кормен Т. Х. – М. : Издательский дом «Вильямс», 2014. – 208 с.
6. Макконнелл Дж. Основы современных алгоритмов / Макконноелл Дж. – [2-е изд.]. – М. : Техносфера, 2004 – 368 с.
7. Кнут Д. Э. Искусство программирования. Том 3. Сортировка и поиск / Кнут Д. Э. ; пер. с англ. – [2-е изд.]. – М. : Издательский дом «Вильямс», 2000. – 832 с.
8. Ахо А. Построение и анализ вычислительных алгоритмов / Ахо А., Хопкрофт Д., Ульман Д. – М. : Мир, 1979. – 536 с.
9. Вирт Н. Алгоритмы и структуры данных / Вирт Н. – М. : Мир, 1989. – 360 с.
10. Марков А. А. Теория алгоритмов / А. А. Марков, Н. М. Нагорный. – М. : Наука, 1984. – 432 с.
11. Криницкий Н. А. Алгоритмы вокруг нас / Криницкий Н. А. – М. : Наука, 1984. – 224 с.
12. Молчановський О. І. Теорія алгоритмів / Молчановський О. І. [Електронний ресурс]. Режим доступу: http://oim.asu.kpi.ua/courses/theory-of-algorithms.
13. Поздоров С. Ю. Полный конспект лекций по курсу теория алгоритмов / Поздоров С. Ю. [Електронний ресурс]. Режим доступу: http://window.edu.ru/catalog/pdf2txt/127/28127/11349.
14. Сінько Ю. І. Лекції з математичної логіки та теорії алгоритмів (Інформатика) / Сінько Ю. І. [Електронний ресурс]. Режим доступу: http://dls.ksu.kherson.ua/dls/Library/LibdocView.aspx?id=612fd82a-c912-4c4a-a039-43651f0454e3.
15. Standard C++ Library reference. Vector. [Електронний ресурс]. Режим доступу: http://www/cplusplus/com/reference/vector/vector.

16. Стандартная библиотека языка программирования C++ [Електронний ресурс]. Режим доступу: https://ru.wikipedia.org/wiki/Vector_(C%2B%2B).

17. Передача параметров функции по ссылке и по значению / Основы Visual C++ / Visual C++ [Електронний ресурс]. Режим доступу: http://netcode.ru/cpp/?click=013-1.aspx.htm.

18. Особенности языка С. Учебное пособие. Функции. Передача аргументов по значению и по ссылке [Електронний ресурс]. Режим доступу: http://younglinux.info/c/function.

19. Standard C++ Library reference. Pair. [Електронний ресурс]. Режим доступу: http://www.cplusplus.com/reference/utility/pair.

Навчальне видання

Методичні вказівки і завдання до контрольної роботи з дисципліни «Теорія алгоритмів» для студентів заочної форми навчання освітньо-кваліфікаційного рівня бакалавр, галузі знань 12 – «Інформаційні технології», спеціальності 122 – «Комп'ютерні науки та інформаційні технології», спеціалізації «Комп'ютерні науки»

Редактор Є. Плетньова
Укладачі: Арсенюк Ігор Ростиславович
Яровий Андрій Анатолійович
Майданюк Володимир Павлович

Оригінал-макет підготовлено І. Арсенюком

Підписано до друку 16.01.2018
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. 2,36
Наклад 40 (1-й запуск 1-20) пр. Зам. № 2018-020.

Видавець та виготовлювач
інформаційний редакційно-видавничий центр.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
Тел. (0432) 65-18-06.
press.vntu.edu.ua;
E-mail: kivc.vntu@gmail.com.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.