ЕЛЕКТРОННІ ІНФОРМАЦІЙНІ РЕСУРСИ: СТВОРЕННЯ, ВИКОРИСТАННЯ, ДОСТУП

ЗБІРНИК МАТЕРІАЛІВ

Міжнародної науково-практичної Інтернет-конференції

9-10 листопада 2020 р.

Міністерство освіти і науки України

Вінницький національний технічний університет
Національна академія Державної прикордонної служби України
ім. Богдана Хмельницького
Вінницький національний медичний університет ім. М.І. Пирогова
Комунальний заклад вищої освіти «Вінницька академія безперервної освіти»
Комунальний заклад «Сумський обласний інститут післядипломної педагогічної освіти»
Люблінська політехніка (Польща)
Новий університет Лісабону (Португалія)

«ЕЛЕКТРОННІ ІНФОРМАЦІЙНІ РЕСУРСИ: СТВОРЕННЯ, ВИКОРИСТАННЯ, ДОСТУП»

ЗБІРНИК МАТЕРІАЛІВ

Міжнародної науково-практичної Інтернет-конференції

9-10 листопада 2020 р.

Суми/Вінниця
НІКО/ВНТУ
2020

УДК 004
ББК 32.97
Е50

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 5 від 26.11.2020 р.)

Електронні інформаційні ресурси: створення, використання, доступ: Збірник матеріалів Міжнародної науково-практичної Інтернет конференції 9-10 листопада 2020 р. – Суми/Вінниця : НІКО/ВНТУ, 2020. – 280 с.

ISBN 978-617-7422-13-5

Збірник містить матеріали Міжнародної науково-практичної Інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ».

Матеріали збірника подано у авторській редакції. Автори опублікованих матеріалів несуть повну відповідальність за підбір, точність наведених фактів, цитат, статистичних даних, власних імен та інших відомостей, Матеріали відтворюються зі збереженням змісту, орфографії та синтаксису текстів, наданих авторами.

УДК 004
ISBN 978-617-7422-13-5

ЕЛЕКТРОННІ ІНФОРМАЦІЙНІ РЕСУРСИ: СТВОРЕННЯ, ВИКОРИСТАННЯ, ДОСТУП:
Збірник матеріалів
Міжнародної науково-практичної Інтернет-конференції
9-10 листопада 2020 р.

Редактор С.А.Пойда, Н.А. Ніколаєнко
Комп'ютерне верстання С.А.Пойда, М.С. Ніколаєнко

Підписано до друку 01.11.2020 Гарнітура Times New Roman
Формат 60х84/16 Папір офсетний
Друк цифровий Ум. друк. арк. 16,3
Тираж 300 пр. Зам. № 2/20

Видавництво НІКО
м.Суми, вул.Харківська, 54
Свідоцтво про внесення до Державного реєстру
суб'єктів видавничої справи України
серія СМв № 044
від 15.10.2012
E-mail: ms.niko@i.ua
Телефон для замовлень: +38(066) 270-64-68

Коваленко Оксана Олексіївна,
к*андидат технічних наук,*
доцент кафедри програмного забезпечення
Вінницький національний технічний університет,
Корягіна Діана Олександрівна,
студентка групи 2ПІ-19б,
факультет інформаційних технологій та комп'ютерної інженерії,
Вінницький національний технічний університет

ВИКОРИСТАННЯ АЛГОРИТМІВ У БІБЛІОТЕКАХ МОВ ПРОГРАМУВАННЯ

У данній статті було розглянуто використання алгоритмів у бібліотеках мов програмування.

Ключові слова: *алгоритми, сортування, лінійний пошук, двійковий пошук, пошук стрибками, інтерполяційний пошук, експоненціальний пошук, жадібний алгоритм.*

This article had regarded the use of algorithms in programming language libraries.

Keywords: a*lgorithms, sorting, linear search, binary search, jumping search, interpolation search, exponential search, greedy algorithm.*

За всю історію комп'ютерних наук склалося розуміння, які алгоритми та структури даних (способи їх зберігання) потрібні для вирішення практичних завдань, певний набір, який повинен знати кожен розробник. Наприклад, сортування: товари в магазині сортують за вартістю або терміну придатності, а ресторани – за віддаленістю або рейтингу. Хеш-таблиці допомагають перевірити коректність пароля та не зберігати його на сайті у відкритому вигляді, графи – знаходити найкоротший шлях і зберігати зв'язки між користувачами в соцмережах.

Алгоритми – це послідовність точно визначених дій, які призводять до вирішення поставленої задачі чи певного завдання і на сьогодні уже створено величезну кількість алгоритмів для вирішення важких задач, що полегшують написання коду будь-якому програмісту, особливо початківцям [1].

Метою роботи є виявлення найбільш популярних алгоритмів у бібліотеках мов програмування.

Усі алгоритми та структури даних вже давно реалізовані в бібліотеках популярних мов програмування. Більше ніхто не пише вручну алгоритм сортування чисел, а щоб користуватися хеш-таблицями, навіть не потрібно знати, як вони влаштовані.

Але наявність безлічі готових бібліотек не означає, що не потрібно розуміти, як влаштовані алгоритми. Фундаментальні знання допомагають дізнатися, що всередині, як воно працює і чому одне рішення краще, аніж інше у конкретній ситуації. Якщо зрозуміти, як влаштовані класичні алгоритми, то можна створювати власні рішення, комбінувати методи один з одним, щоб вирішувати більш складні завдання.

У програмуванні стандартна бібліотека — це бібліотека, що доступна в усіх реалізаціях даної мови програмування. Зміст такої бібліотеки зазвичай описано у специфікації мови, однак також він може частково або повністю визначатися більш неформальними практиками програмістів, що користуються нею. Більшість стандартних бібліотек включають у себе визначення принаймні таких найчастіше використовуваних інструментів як:

- Алгоритми (такі як алгоритми сортування);
- Структури даних (наприклад, списки, дерева, хеш-таблиці);
- Взаємодія з відповідною платформою (введення-виведення, системні виклики та ін.).

Пошук – поширена дія, яка виконується в бізнес-додатках. Розглянемо деякі реалізації відомих алгоритмів пошуку [2] на Java.

Лінійний або послідовний пошук – найпростіший алгоритм пошуку. Він рідко використовується через свою неефективність. По суті, це метод повного перебору, і він поступається іншим алгоритмам [3]. У лінійного пошуку немає передумов до стану структури даних. Алгоритм шукає елемент у заданій структурі даних, поки не досягне кінця структури. При знаходженні елемента повертається його позиція у структурі даних. Якщо елемент не знайдений, повертаємо -1. Лінійний пошук можна використовувати для малого, несортоване набору даних, який не збільшується в розмірах.

Реалізація:

```
public static int linearSearch(int arr[], int elementToSearch) {
    for (int index = 0; index < arr.length; index++) {
        if (arr[index] == elementToSearch)
            return index;
    }
    return -1;
}
```

Двійковий або логарифмічний пошук часто використовується через швидкий час пошуку. Цей вид пошуку вимагає попереднього сортування набору даних. Алгоритм ділить вхідну колекцію на рівні половини, і з кожною ітерацією порівнює цільовий елемент з елементом у середині. Пошук закінчується при знаходженні елемента. Інакше продовжуємо шукати елемент, розділяючи і вибираючи відповідний розділ масиву. Цільовий елемент порівнюється із середнім. Ось чому важливо мати відсортовану колекцію при використанні двійкового пошуку [4]. Пошук закінчується, коли firstIndex(вказівник) досягає lastIndex(останнього елемента). Отже перевіривши весь масив Java не було знайдено елемента.

Реалізація:

```
public static int binarySearch(int arr[], int elementToSearch) {
   int firstIndex = 0;
   int lastIndex = arr.length - 1;
   // умова припинення(елемент не представлено)
   while(firstIndex <= lastIndex) {
      int middleIndex = (firstIndex + lastIndex) / 2;
      // якщо середній елемент – цільовий елемент, повернути його індекс
      if (arr[middleIndex] == elementToSearch) {
         return middleIndex;
      }
      // якщо середній елемент менше
      // направляємо індекс у middle+1, забираючи першу частину з
переглянутого
      else if (arr[middleIndex] < elementToSearch)
               firstIndex = middleIndex + 1;
      // якщо середній елемент більше
      // направляємо індекс у middle-1, забираючи другу частину з
переглянутого
      else if (arr[middleIndex] > elementToSearch)
         lastIndex = middleIndex - 1;

   }
   return -1;
}
```

Пошук стрибками, цей алгоритм від двійкового пошуку відрізняється рухом виключно вперед. Такий пошук вимагає відсортованої колекції. Стрибаючи вперед на інтервал sqrt (arraylength), досягаючи елемента більшого, ніж поточний елемент або кінця масиву. При кожному стрибку записується

попередній крок. Стрибки припиняються, коли знайдений елемент більше шуканого. Потім запускаємо лінійний пошук між попереднім і поточним кроками. Це зменшує поле пошуку та робить лінійний пошук життєздатним варіантом [5].

Реалізація:

```
public static int jumpSearch(int[] integers, int elementToSearch) {
   int arrayLength = integers.length;
   int jumpStep = (int) Math.sqrt(integers.length);
   int previousStep = 0;
   while (integers[Math.min(jumpStep, arrayLength) - 1] < elementToSearch) {
      previousStep = jumpStep;
      jumpStep += (int)(Math.sqrt(arrayLength));
      if (previousStep >= arrayLength;
         return -1;
}
   while (integers[previousStep] < elementToSearch) {
      previousStep++;
      if (previousStep == Math.min(jumpStep, arrayLength));
         return -1;
}
   if (integers[previousStep] == elementToSearch)
      return previousStep;
   return -1;
}
```

Інтерполяційний пошук використовується для пошуку елементів у відсортованому масиві. Він корисний для рівномірно розподілених у структурі даних. При рівномірно розподілених даних місцезнаходження елемента визначається точніше. Тут і розкривається відміна алгоритму від бінарного пошуку, де потрібно знайти елемент у середині масиву. Для пошуку елементів у масиві алгоритм використовує формули інтерполяції. Найефективніше застосовувати ці формула для великих масивів. В іншому випадку алгоритм працює як лінійний пошук.

Реалізація:

```
public static int interpolationSearch(int[] integers, int elementToSearch) {
   int startIndex = 0;
   int lastIndex = (integers.length - 1);
   while ((startIndex <= lastIndex) && (elementToSearch >=
integers[startIndex]) &&
         (elementToSearch <= integers[lastIndex])) {
```

```
// використовуємо формулу інтерполяції для пошуку можливої кращої позиції для відомого елемента
        int pos = startIndex + (((lastIndex-startIndex) /
          (integers[lastIndex]-integers[startIndex]))*
                    (elementToSearch - integers[startIndex]));
        if (integers[pos] == elementToSearch)
          return pos;
        if (integers[pos] < elementToSearch)
          startIndex = pos + 1;
        else
          lastIndex = pos - 1;
      }
      return -1;
    }
```

Експоненціальний пошук використовується для пошуку елементів шляхом переходу в експоненціальні позиції, тобто у другу ступінь. У цьому пошуку потрібно знайти порівняно менший діапазон і застосовувати на ньому двійковий алгоритм для пошуку елемента. Для роботи алгоритму колекція повинна бути відсортована. Експоненціальний пошук використовується з великими масивами, коли бінарний пошук витратний. Такий пошук розділяє дані на більш доступні для пошуку розділи.

Реалізація:

```
public static int exponentialSearch(int[] integers, int elementToSearch) {

    if (integers[0] == elementToSearch)
      return 0;
    if (integers[integers.length - 1] == elementToSearch)
      return integers.length;
    int range = 1;
    while (range < integers.length && integers[range] <= elementToSearch) {
      range = range * 2;
    }
  return Arrays.binarySearch(integers, range / 2, Math.min(range, integers.length), elementToSearch);
  }
```

Також, у теорії алгоритмів жадібні алгоритми відіграють важливу роль. Вони прості для розуміння та реалізації, працюють порівняно швидко, відомо багато різноманітних задач, які можна вирішити за допомогою таких алгоритмів [6]. Однак не завжди можна довести можливість застосовності

жадібного алгоритму для знаходження точного вирішення багатьох завдань.

Жадібний алгоритм – метод розв'язання оптимізаційних задач, заснований на тому, що процес прийняття рішення можна розбити на елементарні кроки, на кожному з яких приймається окреме рішення. Рішення, прийняте на кожному кроці, має бути оптимальним тільки на поточному кроці та повинне прийматися без врахування попередніх або наступних рішень.

У жадібному алгоритмі завжди робиться вибір, який здається найкращим у даний момент - тобто виробляється локально оптимальний вибір у надії, що він приведе до оптимального рішення глобальної задачі. Жадібні алгоритми не завжди приводять до оптимального рішення, але в багатьох завданнях вони дають потрібний результат. Цей алгоритм володіє достатньою потужністю та добре підходить для широкого класу задач. Алгоритми пошуку мінімальних остових дерев є класичним прикладом застосування жадібної стратегії [7].

Ознаки того, що задачу можливо вирішити за допомогою жадібного алгоритму:

- задачу можна розбити на підзадачі;
- величини, що розглядаються в задачі, можна дробити так само на підзадачі;
- сума оптимальних рішень для двох підзадач надає оптимальне рішення для всієї задачі.

Розглянемо простий приклад завдання, що розв’язується жадібним алгоритмом:

Наприклад, розглянемо проблему виплати 98 копійок монетами номіналом 1, 2, 5, 10 і 25 копійок так, щоб загальна кількість монет було мінімально.

Рішення:

Жадібний алгоритм у цьому випадку полягає в тому, щоб на кожному кроці побудови рішення використовувати монети максимального номіналу, і тим, щоб їх було якомога менше (досягнення локального мінімуму). Для початку необхідно три монети по 25 копійок (4 монети дають більшу суму, ніж потрібно). Залишається виплатити 98 – 25*3 = 23 копійки.

На наступному кроці потрібно обрати чергові найбільші за номіналом монети, якими можна видати решту суми, — дві монети по 10 копійок.

Два наступні кроки – це по одній монеті номіналом 1 і 2 копійки, тим самим дозволяючи виплатити всю суму 7 монетами.

Висновки.

Кожна система містить набір обмежень і вимог. Правильно підібраний алгоритм пошуку, що враховує ці обмеження відіграє визначальну роль у продуктивності системи. Алгоритми, призначені для вирішення завдань оптимізації, звичайно являють собою послідовність кроків, на кожному з яких

надається деяка множина виборів. Визначення найкращого вибору, керуючись принципами динамічного програмування, у багатьох задачах оптимізації нагадує стрілянину з гармати по горобцях; іншими словами, для цих завдань краще підходять більш прості й ефективні алгоритми.

Тому основне завдання програміста - аналізувати і вирішувати проблеми, де код - це всього лише інструмент досягнення мети. Часто виникають проблеми, які важко вирішити, тоді програмісту слід розробити новий алгоритм або поміркувати, як використовувати існуючий. Адже якщо знати про принципи роботи алгоритмів, тоді існує більша ймовірність знайти краще рішення. Іноді навіть нову проблему можна звести до старої, але для цього потрібно володіти фундаментальними знаннями.

Список використаної літератури

1. Вікіпедія GPGPU [Електронний ресурс]. Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/GPGPU – Назва з екрану. Зачем программисту изучать алгоритмы. Tproger [Електронний ресурс] – Режим доступу до ресурсу: https://tproger.ru/articles/why-learn-algorithms/
2. Алгоритми пошуку. UA5.ORG [Електронний ресурс] – Режим доступу до ресурсу: https://www.ua5.org/osnprog/418-algoritmi-poshuku.html
3. Нужны ли программисту алгоритмы и структуры данных. Dou [Електронний ресурс] – Режим доступу до ресурсу:
4. https://dou.ua/lenta/articles/algorithms-and-structures/
5. Про алгоритмы для новичков. Proglib [Електронний ресурс] – Режим доступу до ресурсу: https://proglib.io/p/what-is-an-algorithm/
6. Алгоритми і структури даних. Distance Learning [Електронний ресурс] Режим доступу до ресурсу: https://dl.sumdu.edu.ua/textbooks/95351/522217/index.html
7. Алгоритмы и структуры данных в Java. Proselyte [Електронний ресурс] – Режим доступу до ресурсу: https://proselyte.net/algorithms-and-data-structures-in-java/
8. Жадібні алгоритми. Distance learning [Електронний ресурс] – Режим доступу до ресурсу: https://dl.sumdu.edu.ua/textbooks/95351/522264/index.html