Оглавление

Поразрядная сортировка / Radix sort

Битонная сортировка / Bitonic sort

Виды поиска и их сравнение

Жадный алгоритм (greedy algorithm)

Queue, Deque, stack, Heap

EnumSet

Бинарное дерево

Красно-черное дерево

Мемоизация

Big O

«O» (большое) — математическое обозначение для сравнения асимптотического поведения (асимптотики) функций.

Под асимптотикой понимается характер изменения функции при стремлении ее аргумента к определенной точке.

Фраза «сложность алгоритма есть O(f(n)) означает, что с увеличением параметра n, характеризующего количество входной информации алгоритма, время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на f(n).

«O» описывает, насколько быстро работает алгоритм. Время выполнения «O» имеет вид O(n). Постойте, но где же секунды? А их здесь нет — «O» не сообщает скорость в секундах, а позволяет сравнить количество операций. Оно указывает, насколько быстро возрастает время выполнения алгоритма. Такая запись O(n) — сообщает количество операций, которые придется выполнить алгоритму.

Временная сложность алгоритмов

Временная сложность алгоритма определяется как функция от длины строки,

представляющей входные данные, равная времени работы алгоритма на данном входе.

Временная сложность алгоритма обычно выражается с использованием нотации Big O, которая учитывает только слагаемое самого высокого порядка, а также не учитывает константные множители, т. е. коэффициенты.

Временная сложность обычно оценивается путем подсчета числа элементарных операций, осуществляемых алгоритмом.

Время исполнения одной такой операции при этом берется константой, т. е. асимптотически оценивается как O(1).

В таких обозначениях полное время исполнения и число элементарных операций, выполненных алгоритмом, отличаются максимум на постоянный множитель, который не учитывается в O-нотации.

В порядке возрастания сложности:
· O(1) — константная, чтение по индексу из массива;
· O(log(n)) — логарифмическая, бинарный поиск в отсортированном массиве;
· O(n) — линейная, перебор массива в цикле, два цикла подряд, линейный поиск наименьшего или наибольшего элемента в неотсортированном массиве;
· O(n*log(n)) — квазилинейная, сортировка

слиянием, сортировка кучей;
· O(n^2) — полиномиальная (квадратичная), вложенный цикл, перебор двумерного массива, сортировка пузырьком, сортировка вставками;
· O(n^3) — кубическая сложность, обычное умножение двух n × n матриц;
· O(2^n) — экспоненциальная, алгоритмы разложения на множители целых чисел;
· O(n!) — факториальная, решение задачи коммивояжера полным перебором.

Быстродействие операций

	Временная сложность (среднее/худшее)			
	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)
HashSet	–	O(1) O(n)	O(1) O(n)	O(1) O(n)
LinkedHashSet	–	O(1) O(n)	O(1) O(n)	O(1) O(n)
TreeSet	–	O(log(n))	O(log(n))	O(log(n))
Hashtable	–	O(1) O(n)	O(1) O(n)	O(1) O(n)
HashMap	–	O(1) O(n)	O(1) O(n)	O(1) O(n)
LinkedHashMap	–	O(1) O(n)	O(1) O(n)	O(1) O(n)
TreeSet	–	O(log(n))	O(log(n))	O(log(n))

ArrayList и LinkedList

LinkedList в подавляющем большинстве случаев проигрывает ArrayList, но в оставшемся меньшинстве он вне конкуренции.

	ArrayList	LinkedList
get()	всегда	—
set()	начало и середина	конец
add()	середина и конец	начало

remove()	середина и конец	начало

Сравнение сложности:

	Временная сложность			
	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)

Устройство HashMap

HashMap — ассоциативный массив, позволяющий хранить пары «ключ-значение». Каждая ячейка массива — бакет (корзина), хранящая в себе ссылки на списки элементов, узлов (Node).

В корзине может быть один или больше Nod, хранящихся в виде двусвязного списка. При добавлении новой пары «ключ-значение», вычисляется хэш-код ключа, на основании которого вычисляется номер корзины (номер ячейки массива), в которую попадает новый элемент.

Если корзина пустая, то в нее сохраняется ссылка на вновь добавляемый элемент, если же там уже есть элемент, то происходит последовательный переход по ссылкам между элементами в цепочке, в поисках последнего элемента, от которого и ставится ссылка на вновь добавленный элемент.

Если в списке был найден элемент с таким же ключом (по equals()), то он заменяется.

Если в списке был найден элемент, ключ которого имеет такой же хэш-код, но разный `equals()`, значит произошла коллизия и в списке может быть больше одного узла (Node).

Если в списке все ключи имеют одинаковый хэшкод, но разный `equals()`, все элементы добавляются в одну корзину и `HashMap` теряет все свои преимущества, поскольку вырождается в простой двусвязный список элементов.

Рекурсия

Рекурсия — способ отображения какого-либо процесса внутри самого этого процесса, т. е. ситуация, когда процесс является частью самого себя. Для того, чтобы понять рекурсию, надо сначала понять рекурсию.

Рекурсия состоит из базового случая и шага рекурсии. Базовый случай представляет собой самую простую задачу, которая решается за одну итерацию. Например,

```
1. if (n == 0) return 1;
```

В базовом случае обязательно присутствует условие выхода из рекурсии. Смысл рекурсии в движении от исходной задачи к базовому случаю, пошагово уменьшая

размер исходной задачи на каждом шаге рекурсии.

После того, как будет найден базовый случай, срабатывает условие выхода из рекурсии, и стек рекурсивных вызовов разворачивается в обратном порядке, пересчитывая результат исходной задачи, который основан на результате, найденном в базовом случае.

Так работает рекурсивное вычисление факториала:

```
1. int factorial(int n) {
2.     if (n == 0) return 1;
3.     else return n * factorial(n - 1);
4. }
```

Рекурсия имеет линейную сложность O(n).

Рекурсия и Iterator

Циклы дают лучшую производительность, чем рекурсивные вызовы, поскольку вызовы методов потребляют больше ресурсов, чем исполнение обычных операторов. Циклы гарантируют отсутствие переполнения стека. В случае рекурсии стек вызовов разрастается, и его необходимо просматривать для получения конечного ответа. При использовании головной рекурсии также необходимо принимать во внимание размер стека.

Виды сортировок и их сравнение

Сортировка пузырьком (bubble sort) O(n^2)

Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Заметим, что после первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. После двух итераций на правильном месте будут стоять два наибольших элемента, и так далее. Очевидно, не более чем после n итераций массив будет отсортирован. Т. о., асимптотика в худшем и среднем случае — O(n^2), в лучшем случае — O(n).

```
 1. public void main() {
 2.     int[] testData =
 3.             {10, 4, 43, 5, 4, 67, 12, 0, 99, 19};
 4.     System.out.println(Arrays.toString(bubbleSort(
 5.             testData)));
 6. }
 7.
 8. public static int[] bubbleSort(int[] array) {
 9.     boolean sorted = false;
10.     int temp;
11.     while (!sorted) {
12.         sorted = true;
13.         for (
14.                 int i = 0;
15.                 i < array.length - 1;
16.                 i++) {
17.             if (array[i] > array[i+1]) {
18.                 temp = array[i];
19.                 array[i] = array[i+1];
20.                 array[i+1] = temp;
21.                 sorted = false;
22.             }
```

```
            }
        }
        return array;
    }
```

Шейкерная сортировка (shaker sort) $O(n^2)$

Также известна как сортировка перемешиванием и коктейльная сортировка. Заметим, что сортировка пузырьком работает медленно на тестах, в которых маленькие элементы стоят в конце (их еще называют «черепахами»). Такой элемент на каждом шаге алгоритма будет сдвигаться всего на одну позицию влево. Поэтому будем идти не только слева направо, но и справа налево. Будем поддерживать два указателя begin и end, обозначающих, какой отрезок массива еще не отсортирован. На очередной итерации при достижении end вычитаем из него единицу и движемся справа налево, аналогично, при достижении begin прибавляем единицу и двигаемся слева направо. Асимптотика у алгоритма такая же, как и у сортировки пузырьком, однако реальное время работы лучше.

Сортировка расческой (xomb sort) $O(n^2)$

Еще одна модификация сортировки пузырьком. Для того, чтобы избавиться от «черепах», будем переставлять элементы, стоящие на расстоянии. Зафиксируем его и будем идти слева направо, сравнивая

элементы, стоящие на этом расстоянии, переставляя их, если необходимо. Очевидно, это позволит «черепахам» быстро добраться в начало массива. Оптимально изначально взять расстояние равным длине массива, а далее делить его на некоторый коэффициент, равный примерно 1.247. Когда расстояние станет равно единице, выполняется сортировка пузырьком. В лучшем случае асимптотика равна O(n*logn), в худшем — $O(n^2)$. Какая асимптотика в среднем мне не очень понятно, на практике похоже на O(n*logn).

Сортировка вставками / Insertion sort $O(n^2)$

Создадим массив, в котором после завершения алгоритма будет лежать ответ. Будем поочередно вставлять элементы из исходного массива так, чтобы элементы в массиве-ответе всегда были отсортированы. Асимптотика в среднем и худшем случае — $O(n^2)$, в лучшем — O(n). Реализовывать алгоритм удобнее по-другому (создавать новый массив и реально что-то вставлять в него относительно сложно): просто сделаем так, чтобы отсортирован был некоторый префикс исходного массива, вместо вставки будем менять текущий элемент с предыдущим, пока они стоят в неправильном порядке.

```
 1.  public static void main(String[] args) {
 2.      int[] testData =
 3.              {10, 4, 43, 5, 4, 67, 12, 0, 99, 19};
 4.      System.out.println(Arrays.toString(
 5.              insertionSort(testData)));
 6.  }
 7.
 8.  public static int[] insertionSort(int[] array) {
 9.      for (int i = 1; i < array.length; i++) {
10.          int current = array[i];
11.          int j = i - 1;
12.          while(j >= 0 && current < array[j]) {
13.              array[j+1] = array[j];
14.              j--;
15.          }
16.          array[j+1] = current;
17.      }
18.      return array;
19.  }
```

Сортировка Шелла (shell sort) $O(n^2)$

Используем ту же идею, что и сортировка с расческой, и применим к сортировке вставками. Зафиксируем некоторое расстояние. Тогда элементы массива разобьются на классы — в один класс попадают элементы, расстояние между которыми кратно зафиксированному расстоянию. Отсортируем сортировкой вставками каждый класс. В отличие от сортировки расческой, неизвестен оптимальный набор расстояний. Существует довольно много последовательностей с разными оценками. Последовательность Шелла — первый элемент равен длине массива, каждый следующий вдвое меньше предыдущего.

Асимптотика в худшем случае — $O(n^2)$. Последовательность Хиббарда — 2n — 1, асимптотика в худшем случае — O(n1,5), последовательность Седжвика (формула нетривиальна, можете ее посмотреть по ссылке ниже) — O(n4/3), Пратта (все произведения степеней двойки и тройки) — O(n*log2n). Отмечу, что все эти последовательности нужно рассчитать только до размера массива и запускать от большего от меньшему (иначе получится просто сортировка вставками). Также я провел дополнительное исследование и протестировал разные последовательности вида si = a * si — 1 + k * si — 1 (отчасти это было навеяно эмпирической последовательностью Циура — одной из лучших последовательностей расстояний для небольшого количества элементов). Наилучшими оказались последовательности с коэффициентами a = 3, k = 1/3; a = 4, k = 1/4 и a = 4, k = -1/5.

Сортировка деревом (tree sort) O(n*logn)

Будем вставлять элементы в двоичное дерево поиска. После того, как все элементы вставлены достаточно обойти дерево в глубину и получить отсортированный массив. Если использовать сбалансированное

дерево, например красно-черное, асимптотика будет равна O(n*logn) в худшем, среднем и лучшем случае. В реализации использован контейнер multiset.

Гномья сортировка (gnome sort)

Алгоритм похож на сортировку вставками. Поддерживаем указатель на текущий элемент, если он больше предыдущего или он первый — смещаем указатель на позицию вправо, иначе меняем текущий и предыдущий элементы местами и смещаемся влево.

Сортировка выбором (selection sort) $O(n^2)$

На очередной итерации будем находить минимум в массиве после текущего элемента и менять его с ним, если надо. Т. о., после i-ой итерации первые i элементов будут стоять на своих местах. Асимптотика: $O(n^2)$ в лучшем, среднем и худшем случае. Нужно отметить, что эту сортировку можно реализовать двумя способами — сохраняя минимум и его индекс или просто переставляя текущий элемент с рассматриваемым, если они стоят в неправильном порядке. Первый способ оказался немного быстрее.

```
1.  public class SelectionSort {
2.      public static void main(String[] args) {
3.      int[] testData =
4.              {10, 4, 43, 5, 4, 67, 12, 0, 99, 19};
5.      System.out.println(Arrays.toString(
```

```
  6.                 selectionSort(testData)));
  7.     }
  8.
  9.     public static int[] selectionSort(
 10.             int[] array) {
 11.         for (int i = 0; i < array.length; i++) {
 12.             int min = array[i];
 13.             int minId = i;
 14.             for (int j = i + 1; j < array.length; j++)
 15.     {
 16.                 if (array[j] < min) {
 17.                     min = array[j];
 18.                     minId = j;
 19.                 }
 20.             }
 21.             int temp = array[i];
 22.             array[i] = min;
 23.             array[minId] = temp;
 24.         }
 25.         return array;
 26.     }
```

Пирамидальная сортировка (heapsort) O(n*logn)

Развитие идеи сортировки выбором. Воспользуемся структурой данных «куча» (или «пирамида», откуда и название алгоритма). Она позволяет получать минимум за O(1), добавляя элементы и извлекая минимум за O(logn). Т. о., асимптотика O(n*logn) в худшем, среднем и лучшем случае.

```
  1.     public class HeapSort {
  2.         int[] testData =
  3.                 {8, 0, -3, 5, 6, 9, 8, -4, 2, -99,
  4.                     43};
  5.
  6.         sort(testData);
  7.         System.out.println("Sorted array is");
```

```
 8.
 9.      printArray(testData);
10.  }
11.
12.  public void sort(int[] array) {
13.      int n = array.length;
14.
15.      //Построение кучи (перегруппируем массив)
16.      for (int i = n / 2 - 1; i >= 0; i--)
17.          heapify(array, n, i);
18.
19.      //Один за другим извлекаем элементы из кучи
20.      for (int i=n-1; i>=0; i--) {
21.          //Перемещаем текущий корень в конец
22.          int temp = array[0];
23.          array[0] = array[i];
24.          array[i] = temp;
25.
26.          //Вызываем процедуру heapify на
27.  уменьшенной куче
28.          heapify(array, i, 0);
29.      }
30.  }
31.
32.  //Процедура для преобразования в двоичную кучу
33.  поддерева с корневым узлом i, что является
34.  //индексом в arr[]. n - размер кучи
35.  void heapify(int[] arr, int n, int i) {
36.      int largest = i; //Инициализируем наибольший
37.  элемент как корень
38.      int l = 2*i + 1; //левый = 2*i + 1
39.      int r = 2*i + 2; //правый = 2*i + 2
40.
41.      //Если левый дочерний элемент больше корня
42.      if (l < n && arr[l] > arr[largest])
43.          largest = l;
44.
45.      //Если правый дочерний элемент больше, чем
46.  самый большой элемент на данный момент
47.      if (r < n && arr[r] > arr[largest])
48.          largest = r;
49.      //Если самый большой элемент не корень
50.      if (largest != i) {
51.          int swap = arr[i];
```

```
52.         arr[i] = arr[largest];
53.         arr[largest] = swap;
54.
55.         //Рекурсивно преобразуем в двоичную кучу
56. затронутое поддерево
57.         heapify(arr, n, largest);
58.     }
59. }
60.
61. /*Вспомогательная функция для вывода на экран
62. массива размера n*/
63. static void printArray(int arr[]) {
64.     int n = arr.length;
65.     for (int i=0; i < n; ++i)
66.         System.out.print(arr[i] + " ");
67.     System.out.println();
68. }
```

Быстрая сортировка (quicksort) $O(n^2)$

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие — направо. Рекурсивно вызовемся от каждой из частей. В итоге получим отсортированный массив, т. к. каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: $O(n*\log n)$ в среднем и лучшем случае, $O(n^2)$. Наихудшая оценка достигается при неудачном выборе опорного элемента.

```
 1. public static void main(String[] args) {
 2.     int[] testData =
 3.             {8, 0, -3, 5, 6, 9, 8, -4, 2, -99,
 4.                 43};
 5.
 6.     int low = 0;
 7.     int high = testData.length - 1;
 8.
 9.     quickSort(testData, low, high);
10.     System.out.println(Arrays.toString(testData));
```

```
11.  }
12.
13.  public static void quickSort(
14.          int[] array,
15.          int low,
16.          int high) {
17.      if (array.length == 0)
18.          return;
19.          // завершить выполнение, если длина
20.          // массива равна 0
21.
22.      if (low >= high)
23.          return;
24.          // завершить выполнение если уже нечего
25.          // делить
26.
27.      // выбрать опорный элемент
28.      int middle = low + (high - low) / 2;
29.      int opora = array[middle];
30.
31.      // разделить на подмассивы, который больше
32.      // и меньше опорного элемента
33.      int i = low, j = high;
34.      while (i <= j) {
35.          while (array[i] < opora) {
36.              i++;
37.          }
38.
39.          while (array[j] > opora) {
40.              j--;
41.          }
42.
43.          if (i <= j) {//меняем местами
44.              int temp = array[i];
45.              array[i] = array[j];
46.              array[j] = temp;
47.              i++;
48.              j--;
49.          }
50.      }
51.
52.      // вызов рекурсии для сортировки левой
53.      // и правой части
54.      if (low < j)
```

```
55.             quickSort(array, low, j);
56.
57.         if (high > i)
58.             quickSort(array, i, high);
59.     }
```

Сортировка слиянием / Merge sort O(n*logn)

Сортировка, основанная на парадигме «разделяй и властвуй». Разделим массив пополам, рекурсивно отсортируем части, после чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй — на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Слияние работает за O(n), уровней всего logn, поэтому асимптотика O(n*logn). Эффективно заранее создать временный массив и передать его в качестве аргумента функции.
Эта сортировка рекурсивна, как и быстрая, а потому возможен переход на квадратичную при небольшом числе элементов.

```
 1.  public static void main(String[] args) {
 2.      int[] array1 =
 3.              {8, 0, -3, 5, 6, 9, 8, -4, 2, -99,
 4.                  43};
 5.      int[] result = mergesort(array1);
 6.      System.out.println(Arrays.toString(result));
 7.  }
 8.
 9.  public static int[] mergesort(int[] array1) {
10.      int[] buffer1 =
```

```
11.             Arrays.copyOf(array1, array1.length);
12.     int[] buffer2 = new int[array1.length];
13.     int[] result = mergesortInner(
14.             buffer1,
15.             buffer2,
16.             0,
17.             array1.length);
18.     return result;
19. }
20.
21. /**
22.  * @param buffer1 Массив для сортировки.
23.  * @param buffer2 Буфер. Размер должен быть равен
24. размеру buffer1.
25.  * @param startIndex Начальный индекс в buffer1
26. для сортировки.
27.  * @param endIndex Конечный индекс в buffer1 для
28. сортировки.
29.  * @return
30.  */
31. public static int[] mergesortInner(
32.         int[] buffer1,
33.         int[] buffer2,
34.         int startIndex,
35.         int endIndex) {
36.     if (startIndex >= endIndex - 1) {
37.         return buffer1;
38.     }
39.
40.     // уже отсортирован.
41.     int middle = startIndex +
42.             (endIndex - startIndex) / 2;
43.     int[] sorted1 = mergesortInner(
44.             buffer1,
45.             buffer2,
46.             startIndex,
47.             middle);
48.     int[] sorted2 = mergesortInner(
49.             buffer1,
50.             buffer2,
51.             middle,
52.             endIndex);
53.
54.     // Слияние
```

```
55.      int index1 = startIndex;
56.      int index2 = middle;
57.      int destIndex = startIndex;
58.      int[] result =
59.              sorted1 ==
60.                  buffer1 ? buffer2 : buffer1;
61.      while (index1 < middle && index2 < endIndex) {
62.          result[destIndex++] =
63.                  sorted1[index1] <
64.                      sorted2[index2] ?
65.                          sorted1[index1++] :
66.                          sorted2[index2++];
67.      }
68.      while (index1 < middle) {
69.          result[destIndex++] = sorted1[index1++];
70.      }
71.      while (index2 < endIndex) {
72.          result[destIndex++] = sorted2[index2++];
73.      }
74.      return result;
75.  }
```

Сортировка подсчетом (counting sort)

Создадим массив размера r − l, где l — минимальный, а r — максимальный элемент массива. После этого пройдем по массиву и подсчитаем количество вхождений каждого элемента. Теперь можно пройти по массиву значений и выписать каждое число столько раз, сколько нужно. Асимптотика — O(n + r − l). Можно модифицировать этот алгоритм, чтобы он стал стабильным: для этого определим место, где должно стоять очередное число (это просто префиксные суммы в массиве значений) и будем идти по исходному массиву слева направо, ставя элемент на правильное место и увеличивая

позицию на 1. Эта сортировка не тестировалась, поскольку большинство тестов содержало достаточно большие числа, не позволяющие создать массив требуемого размера. Однако она, тем не менее, пригодилась.

Поразрядная сортировка / Radix sort

Также известна как цифровая сортировка. Существует две версии этой сортировки, в которых, мало общего, кроме идеи воспользоваться представлением числа в какой-либо системе счисления (например, двоичной).

LSD (least significant digit). Представим каждое число в двоичном виде. На каждом шаге алгоритма будем сортировать числа т. о., чтобы они были отсортированы по первым k * i битам, где k — некоторая константа. Из данного определения следует, что на каждом шаге достаточно стабильно сортировать элементы по новым k битам. Для этого идеально подходит сортировка подсчетом (необходимо 2k памяти и времени, что немного при удачном выборе константы). Асимптотика: O(n), если считать, что числа фиксированного размера (а в противном случае нельзя было бы считать, что сравнение двух чисел выполняется за единицу времени).

MSD (most significant digit). На самом деле, некоторая разновидность блочной сортировки. В один блок будут попадать числа с равными k битами. Асимптотика такая же, как и у LSD версии. Реализация очень похожа на блочную сортировку, но проще. В ней используется функция digit, определенная в реализации LSD версии.

Битонная сортировка / Bitonic sort

Идея данного алгоритма заключается в том, что исходный массив преобразуется в битонную последовательность — последовательность, которая сначала возрастает, а потом убывает. Ее можно эффективно отсортировать следующим образом: разобьем массив на две части, создадим два массива, в первый добавим все элементы, равные минимуму из соответственных элементов каждой из двух частей, а во второй — равные максимуму. Утверждается, что получатся две битонные последовательности, каждую из которых можно рекурсивно отсортировать тем же образом, после чего можно склеить два массива (т. к. любой элемент первого меньше или равен любого элемента второго). Для того, чтобы преобразовать исходный массив в битонную последовательность, сделаем следующее: если массив состоит из двух элементов, можно

просто завершиться, иначе разделим массив пополам, рекурсивно вызовем от половинок алгоритм, после чего отсортируем первую часть по порядку, вторую в обратном порядке и склеим. Очевидно, получится битонная последовательность. Асимптотика: O(n*log2n), поскольку при построении битонной последовательности мы использовали сортировку, работающую за O(n*logn), а всего уровней было logn. Также заметим, что размер массива должен быть равен степени двойки, так что, возможно, придется его дополнять фиктивными элементами (что не влияет на асимптотику).

Гибридная сортировка (Timsort), совмещающая сортировку вставками и сортировку слиянием. Разобьем элементы массива на несколько подмассивов небольшого размера, при этом будем расширять подмассив, пока элементы в нем отсортированы. Отсортируем подмассивы сортировкой вставками, пользуясь тем, что она эффективно работает на отсортированных массивах. Далее будем сливать подмассивы как в сортировке слиянием, беря их примерно равного размера (иначе время работы приблизится к квадратичному). Для этого удобного хранить подмассивы в стеке, поддерживая инвариант — чем дальше от вершины,

тем больше размер, и сливать подмассивы на верхушке только тогда, когда размер третьего по отдаленности от вершины подмассива больше или равен сумме их размеров. Асимптотика: O(n) в лучшем случае и O(n*logn) в среднем и худшем случае.

Виды поиска и их сравнение

Линейный поиск — O(n);

Бинарный поиск — O(log(n)); сложность ниже, но массив должен быть отсортирован

Поиск в глубину и в ширину в деревьях;

Жадный алгоритм (greedy algorithm)

Жадный алгоритм — алгоритм, который на каждом шагу делает локально наилучший выбор в надежде, что итоговое решение будет оптимальным. К примеру, алгоритм Дейкстры нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными.

К слову, алгоритм Флойда, который тоже ищет кратчайшие пути в графе (правда, между всеми вершинами), не является примером жадного алгоритма.

Флойд демонстрирует другой метод — метод динамического программирования.

Есть область применимости жадных алгоритмов.

Общих рецептов тут нет, но есть довольно мощный инструмент, с помощью которого в большинстве случаев можно определить, даст ли жадина оптимальное решение. Этот инструмент — матроид.

Матроид — пара (X, I), где X — конечное множество, называемое носителем матроида, а I — некоторое множество подмножеств X, называемое семейством независимых множеств. При этом должны выполняться следующие условия: - Множество I непусто. Даже если исходное множество X было пусто — X = Ø, то I будет состоять из одного элемента — множества, содержащего пустое. I = {{Ø}} - Любое подмножество любого элемента множества I также будет элементом этого множества. - Если множества A и B принадлежат множеству I, а также известно, что размер A меньше B, то существует какой-нибудь элемент x из B, не принадлежащий A, такое что объединение x и A будет принадлежать множеству I. Это свойство является не совсем тривиальным, но чаще всего наиважшейшим из всех остальных.

Queue, Deque, stack, Heap

Queue — очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) — соответственно извлечение элемента осуществляется с начала очереди, вставка элемента — в конец очереди.

Хотя этот принцип нарушает, к примеру PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.

Deque (Double Ended Queue) расширяет Queue и, согласно документации, это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого, реализации интерфейса Deque могут строится по принципу FIFO, либо LIFO.

Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

Heap (куча) используется Java Runtime для выделения памяти под объекты и классы. Создание нового объекта также происходит в куче. Это же является областью работы сборщика мусора. Любой объект, созданный

в куче, имеет глобальный доступ и на него могут ссылаться из любой части приложения.

Stack (стек) это область хранения данных также находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. Размер стековой памяти намного меньше объема памяти в куче. Стек в Java работает по схеме LIFO (Последний зашел — Первый вышел).

Различия между Heap и Stack памятью:
· куча используется всеми частями приложения, в то время как стек используется только одним потоком исполнения программы;
· всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится лишь ссылка на него;
· память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче;
· объекты в куче доступны с любой точке программы, в то время как стековая память не может быть доступна для других потоков;

· стековая память существует лишь какое-то время работы программы, а память в куче живет с самого начала до конца работы программы;
· если память стека полностью занята, то Java Runtime бросает исключение java.lang.StackOverflowError;
· если заполнена память кучи, то бросается исключение java.lang.OutOfMemoryError: Java Heap Space;
· размер памяти стека намного меньше памяти в куче;
· из-за простоты распределения памяти, стековая память работает намного быстрее кучи.

EnumSet

Специализированная коллекция Set для работы с enum классами.

Он реализует интерфейс Set и расширяется от AbstractSet. Хотя AbstractSet и AbstractCollection предоставляют реализации почти для всех методов интерфейсов Set и Collection, EnumSet переопределяет большинство из них.

Когда мы планируем использовать EnumSet, мы должны принять во внимание некоторые важные моменты:
· может содержать только enum значения и все значения должны принадлежать к тому

же enum;
· не позволяет добавлять нулевые значения, выбрасывая NullPointerException в попытке это сделать;
· EnumSet не потоко-безопасный, поэтому нам нужно его синхронизировать;
· элементы хранятся в том порядке, в котором они объявлены в enum;
· EnumSet использует отказоустойчивый итератор, который работает с копией, поэтому он не выбрасывает ConcurrentModificationException, если коллекция изменяется при итерации.

Всегда следует отдавать предпочтение использованию EnumSet перед любой другой реализацией Set, когда мы храним значения enum.

Все методы в EnumSet реализованы с использованием арифметических побитовых операций. Эти вычисления очень быстрые, и поэтому все основные операции выполняются за константное время O(1).

Бинарное дерево

Иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Каждый узел в дереве задает поддерево,

корнем которого он является. Оба поддерева (левое и правое) являются двоичными деревьями. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Красно-черное дерево

Красно-черное дерево — один из видов самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла.

Сбалансированность достигается за счет введения дополнительного атрибута узла дерева — «цвета».

Этот атрибут может принимать одно из двух возможных значений — «черный» или «красный».

· Узел может быть либо красным, либо черным и имеет двух потомков.

· Корень — как правило черный. Это правило слабо влияет на работоспособность модели, т. к. цвет

корня всегда можно изменить с красного на черный.

· Все листья — черные и не содержат данных.

· Оба потомка каждого красного узла — черные.

· Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано.

Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-черным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Мемоизация

Один из способов оптимизации, для увеличения скорости выполнения программ — сохранение результатов выполнения функций для предотвращения повторных вычислений.

Перед вызовом функции проверяется, вызывалась ли функция ранее:

· если не вызывалась, то функция вызывается, и результат ее выполнения сохраняется;
· если вызывалась, то используется сохраненный результат. Может применяться, чтобы сократить количество дублирующих рекурсивных вызовов.