Radix Sort

Код:

```c
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>

double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

int getrand(int min, int max)
{
    return (double)rand() / (RAND_MAX + 1.0) * (max - min) + min;
}

// Функция для нахождения максимального значения в массиве
int getMax(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
    }
    return max;
}

// Функция для сортировки массива по разрядам
void countSort(int arr[], int n, int exp)
{
    int output[n]; // Результирующий массив
    int i, count[10] = {0};

    // Заполнение вспомогательного массива
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Изменение вспомогательного массива так, чтобы он содержал
    // позиции каждого элемента в результирующем массиве
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Заполнение результирующего массива
    for (i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Копирование результирующего массива в исходный массив
    for (i = 0; i < n; i++)
```

```
55          arr[i] = output[i];
56  }
57
58  // Основная функция для сортировки массива методом Radix sort
59  void radixsort(int arr[], int n)
60  {
61      // Нахождения максимального значения в массиве
62      int mx = getMax(arr, n);
63
64      // Сортировка массива по разрядам
65      for (int exp = 1; mx / exp > 0; exp *= 10)
66      {
67          countSort(arr, n, exp);
68      }
69  }
70
71  int main()
72  {
73      int min = 0, max = 100000;
74      int x; scanf("%d",&x);
75      int arr[x];
76
77      // Заполнение массива случайными числами
78      for (int i = 0; i < x; i++)
79          arr[i] = getrand(min, max);
80
81      //вычисляем длинну массива
82      int n = sizeof(arr) / sizeof(arr[0]);
83
84      // Вывод элементов массива до сортировки
85      for (int i = 0; i < n; i++)
86      {
87          printf("%d ", arr[i]);
88      }
89
90
91      radixsort(arr, n);
92
93       printf("\n");
94      printf("Sorted array: ");
95       printf("\n");
96      for (int i = 0; i < n; i++)
97      {
98          printf("%d ", arr[i]);
99      }
100      printf("\n");
101     return 0;
102 }
```

Принцип работы алгоритма

1. Разбивается исходный массив на подгруппы по разрядам (например, по единицам, десяткам, сотням и т.д.). 2. Элементы каждой подгруппы сортируются в соответствии с их значением в данном разряде. 3. Подгруппы объединяются в один массив в порядке, соответствующем значению разряда, начиная с наименьшего разряда и заканчивая наибольшим. 4. Процесс повторяется для каждого разряда до тех пор, пока все разряды не будут пройдены. Алгоритм Radix Sort основан на поразрядной сортировке и может быть использован для сортировки любых объектов, которые можно разбить на разряды. Сложность алгоритма зависит от размера входных данных и количества разрядов. В худшем случае, когда количество разрядов равно k, а размер входных данных равен n, сложность алгоритма будет O(kn). Однако, в большинстве случаев, количество разрядов k является константой, что делает сложность алгоритма линейной, O(n)

Сложность

Сложность Radix sort в лучшем, худшем и среднем случае - O(d * (n + k)), где d - количество разрядов в сортируемых числах, n - количество элементов в массиве, k - максимальное значение числа в массиве.

Свойства

Некоторые свойства Radix sort: • Стабильная сортировка (stable sorting) • Сложность в лучшем, среднем и худшем случае - O(d * (n + k)), где d - количество разрядов в сортируемых числах, n - количество элементов в массиве, k - максимальное значение числа в массиве. • Не требует сравнения элементов, поэтому может быть эффективнее в некоторых случаях, чем сортировки, основанные на сравнении элементов. • Работает только для сортировки целых чисел.

Selection Sort

Код:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>


double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

int getrand(int min, int max)
{
    return (double)rand() / (RAND_MAX + 1.0) * (max - min) + min;
}

// Функция для нахождения наименьшего элемента в массиве
int findMinIndex(int arr[], int start, int end) {
    int minIndex = start;
    for (int i = start + 1; i < end; i++) {
        if (arr[i] < arr[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}

// Основная функция для сортировки массива методом Selection sort
void selectionSort(int arr[], int n) {
    int minIndex;

    // Перебираем все элементы массива, кроме последнего
    for (int i = 0; i < n - 1; i++) {
        // Находим наименьший элемент в неотсортированной части
массива
        minIndex = findMinIndex(arr, i, n);

        // Меняем наименьший элемент местами с текущим элементом
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

int main() {
    int min = 0, max = 100000;
    int x; scanf("%d",&x);
    int arr[x];
```

```
50
51      // Заполнение массива случайными числами
52      for (int i = 0; i < x; i++)
53          arr[i] = getrand(min, max);
54
55
56      int n = sizeof(arr) / sizeof(arr[0]);
57
58      // Вывод элементов массива до сортировки
59      for (int i = 0; i < n; i++)
60      {
61          printf("%d ", arr[i]);
62      }
63      selectionSort(arr, n);
64
65      printf("\n");
66      printf("Sorted array: ");
67      printf("\n");
68      for (int i = 0; i < n; i++)
69      {
70          printf("%d ", arr[i]);
71      }
72      printf("\n");
73      return 0;
    }
```

Принцип работы алгоритма

1. Находится минимальный элемент в массиве. 2. Минимальный элемент меняется местами с первым элементом в массиве. 3. Процесс повторяется для оставшихся элементов, начиная со второго элемента и до конца массива. 4. Находится минимальный элемент в оставшейся части массива и меняется местами с первым элементом в оставшейся части массива. 5. Процесс повторяется до тех пор, пока все элементы не будут отсортированы. Алгоритм выбирает наименьший элемент и перемещает его в начало массива. Затем он выбирает следующий наименьший элемент и перемещает его на следующую позицию в массиве, и так далее, пока все элементы не будут отсортированы. Сложность алгоритма Selection Sort составляет O(n^2), где n - количество элементов, которые нужно отсортировать.

Сложность

Сложность сортировки Selection sort в худшем, среднем и лучшем случае - O(n^2).

Heap Sort

Код:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

double wtime()
{
struct timeval t;
gettimeofday(&t, NULL);
return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

int getrand(int min, int max)
{
return (double)rand() / (RAND_MAX + 1.0) * (max - min) + min;
}

// Функция "просеивания" через кучу - формирование кучи
void siftDown(int *arr, int x, int bottom)
{
  int max; // индекс максимального потомка
  int done = 0; // флаг того, что куча сформирована
  // Пока не дошли до последнего ряда
  while ((x * 2 <= bottom) && (!done))
  {
    if (x * 2 == bottom)     // если мы в последнем ряду,
      max = x * 2;     // запоминаем левый потомок
    // иначе запоминаем больший потомок из двух
    else if (arr[x * 2] > arr[x * 2 + 1])
      max = x * 2;
    else
      max = x * 2 + 1;
    // если элемент вершины меньше максимального потомка
    if (arr[x] < arr[max])
    {
      int temp = arr[x]; // меняем их местами
      arr[x] = arr[max];
      arr[max] = temp;
      x = max;
    }
    else
      done = 1; // пирамида сформирована
  }
}
// Функция сортировки на куче
void heapSort(int *arr, int array_size)
{
```

```
47   // Формируем нижний ряд пирамиды
48   for (int i = (array_size / 2); i >= 0; i--)
49     siftDown(arr, i, array_size - 1);
50   // Просеиваем через пирамиду остальные элементы
51   for (int i = array_size - 1; i >= 1; i--)
52   {
53     int temp = arr[0];
54     arr[0] = arr[i];
55     arr[i] = temp;
56     siftDown(arr, 0, i - 1);
57   }
58 }
59
60 int main()
61 {
62
63   int min=0,max=100000;
64   int x; scanf("%d",&x);
65   double n=wtime();
66   int a[x];
67   // Заполнение массива случайными числами
68   for (int i = 0; i<x; i++)
69     a[i] = getrand(min, max);
70   // Вывод элементов массива до сортировки
71   for (int i = 0; i<x; i++)
72     printf("%d ", a[i]);
73   printf("\n");
74   heapSort(a, x); // вызов функции сортировки
75   // Вывод элементов массива после сортировки
76   for (int i = 0; i<x; i++)
77     printf("%d ", a[i]);
78   double m=wtime();
79   printf("%lf\n",m-n);
80   return 0;
81 }
```

Принцип работы алгоритма

1. Строится двоичная куча из исходного массива. 2. Максимальный элемент (находящийся в корне кучи) перемещается в конец массива. 3. Куча перестраивается, исключая последний элемент, чтобы получить новый максимальный элемент. 4. Процесс повторяется для оставшихся элементов, пока все элементы не будут отсортированы. Алгоритм Heap Sort использует двоичную кучу для хранения элементов и упорядочения их в порядке возрастания или убывания. В худшем случае, сложность алгоритма Heap Sort составляет O(n log n), где n - количество элементов, которые нужно отсортировать.

Сложность

Сложность Heap sort в лучшем, худшем и среднем случае - O(n*log n), где n - количество элементов в массиве.

Свойства

Некоторые свойства Heap sort: • Сортировка на месте (in-place sorting) • Нестабильная сортировка (unstable sorting) • Сложность в лучшем, среднем и худшем случае - O(n*log n) • Не требует дополнительной памяти, кроме нескольких переменных для временного хранения данных • Хорошо работает на больших массивах. (изменено)
Как работает алгоритм построения кучи в Heap sort?
Каковы основные преимущества Heap sort перед другими алгоритмами со...
Какие еще алгоритмы сортировки на месте ты знаешь?

Контрольные вопросы

1) Что такое вычислительная сложность алгоритма?

Вычислительная сложность алгоритма - это количество ресурсов, необходимых для выполнения алгоритма, в зависимости от размера входных данных. Обычно вычислительная сложность измеряется в асимптотической нотации, такой как O-большое, Θ-большое или Ω-большое, и показывает, как быстро растет время выполнения алгоритма при увеличении размера входных данных. Вычислительная сложность является важным показателем при выборе наиболее эффективного алгоритма для решения определенной задачи.

(Сложность алгоритмов оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных.

O(f(n)) означает, что время работы алгоритма или объём занимаемой памяти растёт в зависимости от объема входных данных не быстрее, чем некоторая константа, умноженная на f(n).)

2) Что означают записи f(n) = O(g(n)), f(n) = (Θ g(n)), f(n) = Ω(g(n))?

O-большое - ограничение сверху (верхняя граница).
Омега-большое - ограничение снизу (нижняя граница).
Тета - ограничение и сверху, и снизу (верхняя и нижняя граница).

Запись f(n) = O(g(n)) означает, что функция f(n) асимптотически не превосходит функцию g(n), то есть f(n) растет не быстрее, чем g(n).

Запись f(n) = Θ(g(n)) означает, что функция f(n) асимптотически эквивалентна функции g(n), то есть f(n) растет так же быстро, как и g(n).

Запись f(n) = Ω(g(n)) означает, что функция f(n) асимптотически не меньше функции g(n), то есть f(n) растет не медленнее, чем g(n).

3) Какой алгоритм сортировки называется устойчивым (stable)?

Устойчивым (stable) называется алгоритм сортировки, который сохраняет относительный порядок элементов с одинаковыми значениями. Другими словами, если в массиве есть два элемента с одинаковым значением, то устойчивый алгоритм сортировки не изменит их порядок после сортировки. Например, сортировка слиянием (merge sort) и сортировка вставками (insertion sort) являются устойчивыми, в то время как быстрая сортировка (quick sort) не является устойчивой.

4) Какой алгоритм сортировки называется сортировкой «на месте» (in-place)?

Алгоритм сортировки, который работает только с ограниченным количеством памяти и не требует дополнительного выделения памяти для хранения временных данных, называется сортировкой «на месте» (in-place). Такие алгоритмы могут быть особенно полезны при работе с большими объемами данных, когда доступная память ограничена. Примерами таких алгоритмов являются быстрая сортировка (quick sort), сортировка вставками (insertion sort) и сортировка выбором (selection sort).

5) Какая вычислительная сложность в худшем случае у реализованных вами алгоритмов?

Сложность Heap sort в лучшем, худшем и среднем случае - O(n*log n), где n - количество элементов в массиве. Логарифмический член log n в сложности O(nlog n) возникает из-за особенностей алгоритмов, которые разделяют массив на меньшие части и рекурсивно сортируют каждую из них. Такие алгоритмы, например, быстрая сортировка и сортировка слиянием, имеют логарифмическую сложность из-за того, что каждый раз массив делится на две части, и количество разбиений равно log n. Каждый разбирается массив размером n, что дает общую сложность O(nlog n)

Сложность сортировки Selection Sort составляет O(n^2), где n - количество элементов, которые нужно отсортировать потому что алгоритм использует два вложенных цикла для перебора всех элементов массива. Внешний цикл выполняется n раз, где n - количество элементов в массиве, а внутренний цикл также выполняется n раз для каждой итерации внешнего цикла. Это приводит к общему количеству операций, равному n * n, что дает сложность O(n^2). Это означает, что время выполнения алгоритма будет увеличиваться квадратично с увеличением количества элементов.

Сложность сортировки Radix Sort зависит от размера входных данных и количества разрядов. В худшем случае, когда количество разрядов равно k, а размер входных данных равен n, сложность алгоритма будет O(kn). Однако, в большинстве случаев, количество разрядов k является константой, что делает сложность алгоритма линейной, O(n).

6) Объясните поведение кривых на графиках, которые вы построили. Согласуются ли

экспериментальные результаты с оценкой вычислительной сложности алгоритмов?

Самым долгим оказался алгоритм сортировки выбором, так как для любого случая его сложность по времени n^2. Экспериментальные результаты с оценкой вычислительной сложности алгоритмов согласуются.

7) Какие алгоритмы сортировки с вычислительной сложностью O(nlogn) для худшего случая вам известны?

Некоторые известные алгоритмы сортировки с вычислительной сложностью O(nlogn) для худшего случая включают быструю сортировку (quicksort), сортировку слиянием (merge sort) и сортировку кучей (heap sort). Все эти алгоритмы используют подход "разделяй и властвуй", который позволяет разбивать массив на меньшие части и рекурсивно сортировать их. Это позволяет достигать сложности O(nlogn) в худшем случае.

Merge sort (сортировка слиянием) Heap sort (пирамидальная сортировка)

8) Известны ли вам алгоритмы сортировки, работающие быстрее O(nlogn) для худшего

случая?

Для сортировки в худшем случае быстрее O(nlogn) нет алгоритмов, работающих в общем случае. Однако, существуют специализированные алгоритмы, которые могут быть эффективны в определенных случаях, например, для частично отсортированных данных или для данных с ограниченным диапазоном значений. Примерами таких алгоритмов являются сортировка подсчетом (counting sort) и поразрядная сортировка (radix sort).