Отчет

По лабораторной работе 1

Bucket sort

Выполнила:  
Сергеева Олеся 21Пи-3

**Алгоритм**

Блочная сортировка (Bucket sort) — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Каждый блок затем сортируется отдельно. Затем элементы помещаются обратно в [массив](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Сортировка сильно деградирует при большом количестве мало отличных элементов (большинство элементов попадёт в одну корзину). Поэтому такой тип сортировки использовать, когда велика вероятность того, что числа редко повторяются (например, последовательность случайных чисел).

**Псевдокод:**

functiprn bucket-sort(A, n) is

buckets ← новый массив из n пустых элементов

for i = 0 to (length(A)-1) do

вставить A[i] в конец массива buckets[msbits(A[i], k)]

for i = 0 to n - 1 do

next-sort(buckets[i])

return Конкатенация массивов buckets[0], ..., buckets[n-1]

**Реализация:**

Сортировка реализована для отрицательных и положительных целых чисел.

Для сортировки каждой отдельной корзины используется сортировка вставками.

Максимальная допустимая длина сортируемого массива 10000.

При вызове сортировки необходимо передать в параметрах сам массив (void \*base ), размер массива (size\_t n), максимальное значение в массиве или число, которое значения не превышают (int16\_t max\_val), компаратор (int (\_\_cdecl \*compare) (const void \*, const void \*)).

#define N 10000

int16\_t ECOCALLMETHOD CEcoLab1\_BucketSort(/\* in \*/ struct IEcoLab1\* me, /\* in \*/ void \*base, size\_t n, int16\_t max\_val, int (\_\_cdecl \*compare) (const void \*, const void \*)) {

CEcoLab1\* pCMe = (CEcoLab1\*)me;

size\_t i;

size\_t j;

size\_t s;

int16\_t index;

int16\_t\* array = (int16\_t\*)base;

int16\_t (\*positive\_buckets)[N] = (int16\_t(\*)[N])pCMe->m\_pIMem->pVTbl->Alloc(pCMe->m\_pIMem, max\_val \* N \* sizeof(int16\_t));

size\_t\* positive\_bucket\_sizes = (size\_t\*)pCMe->m\_pIMem->pVTbl->Alloc(pCMe->m\_pIMem, max\_val \* sizeof(size\_t));

int16\_t (\*negative\_buckets)[N] = (int16\_t(\*)[N])pCMe->m\_pIMem->pVTbl->Alloc(pCMe->m\_pIMem, max\_val \* N \* sizeof(int16\_t));

size\_t\* negative\_bucket\_sizes = (size\_t\*)pCMe->m\_pIMem->pVTbl->Alloc(pCMe->m\_pIMem, max\_val \* sizeof(size\_t));

/\* Проверка указателей \*/

if (me == 0) {

return -1;

}

max\_val++;

for (i = 0; i < max\_val; ++i) {

positive\_bucket\_sizes[i] = 0;

negative\_bucket\_sizes[i] = 0;

}

for (j = 0; j < n; ++j) {

index = array[j];

if (index >= 0) {

positive\_buckets[index][positive\_bucket\_sizes[index]++] = array[j];

} else {

negative\_buckets[-index][negative\_bucket\_sizes[-index]++] = -array[j];

}

}

for (s = 0; s < max\_val; ++s) {

if (positive\_bucket\_sizes[s] > 0) {

int16\_t\* positive\_bucket = positive\_buckets[s];

size\_t positive\_size = positive\_bucket\_sizes[s];

for (j = 1; j < positive\_size; ++j) {

int16\_t key = positive\_bucket[j];

int16\_t k = j - 1;

while (k >= 0 && compare(&positive\_bucket[k], &key) > 0) {

positive\_bucket[k + 1] = positive\_bucket[k];

k = k - 1;

}

positive\_bucket[k + 1] = key;

}

}

}

for (s = max\_val - 1; s != SIZE\_MAX; --s) {

if (negative\_bucket\_sizes[s] > 0) {

int16\_t\* negative\_bucket = negative\_buckets[s];

size\_t negative\_size = negative\_bucket\_sizes[s];

for (j = 1; j < negative\_size; ++j) {

int16\_t key = negative\_bucket[j];

int16\_t k = j - 1;

while (k >= 0 && compare(&negative\_bucket[k], &key) < 0) {

negative\_bucket[k + 1] = negative\_bucket[k];

k = k - 1;

}

negative\_bucket[k + 1] = key;

}

}

}

index = 0;

for (i = max\_val-1; i != 0; --i) {

for (j = 0; j < negative\_bucket\_sizes[i]; ++j) {

array[index++] = -negative\_buckets[i][j];

}

}

for (i = 0; i < max\_val; ++i) {

for (j = 0; j < positive\_bucket\_sizes[i]; ++j) {

array[index++] = positive\_buckets[i][j];

}

}

pCMe->m\_pIMem->pVTbl->Free(pCMe->m\_pIMem, positive\_buckets);

pCMe->m\_pIMem->pVTbl->Free(pCMe->m\_pIMem, negative\_buckets);

return 0;

}

**Асимптотика**

Блочная сортировка имеет линейную асимптотику в среднем случае, в худшем – может быть квадратичной.

Создание корзин занимает O(max\_val), где max\_val – это максимальное возможное значение в массиве.

Заполнение корзин O(n), где n – размер массива.

Отдельная сортировка каждой корзины – сортировка вставками – O(N \* log N), где N – общее количество корзин.

Объединение корзин в один массив O(n).

Таким образом, сложность алгоритма O(n + N \* log N).

**Результаты тестирования**

Результаты работы Bucket Sort и Quick Sort на разных размерах входного массива n и максимальных значениях max\_val:

n: 1000 max\_val: 100

Time Bucket Sort: 1000.000000 ms

Time Quick Sort: 0.000000 ms

n: 1000 max\_val: 1000

Time Bucket Sort: 3000.000000 ms

Time Quick Sort: 1000.000000 ms

n: 10000 max\_val: 1000

Time Bucket Sort: 4000.000000 ms

Time Quick Sort: 11000.000000 ms

n: 10000 max\_val: 10000

Time Bucket Sort: 22000.000000 ms

Time Quick Sort: 13000.000000 ms

По графику видно, что quick sort работает быстрее в большинстве случаев. Bucket sort лучше сортирует, когда на большом количестве элементов числа равномерно распределены (благодаря небольшому диапазону значений).