Министерство образования и науки Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Факультет фотоники и оптоинформатики

Кафедра Компьютерной фотоники и видеоинформатики

Отчет по практической работе

Тема:

Исследование и реализация детерминированного потокового проходно-параметризуемого алгоритма Мунро-Патерсона для решения задачи поиска k-ой статистики в условиях ограниченных ресурсов памяти

Студент _____ _Краюшкин О.Н._ Группа 5357
(подпись) (Ф.И.О)

Руководитель_____
(Ф.И.О.,должность, уч. степень)

_____ _____
(подпись) (дата)

Работа защищена "___"_____ 20__ г.

с оценкой _____

_____
(подпись)

Санкт-Петербург
2015 г.

Введение

Если при решении задачи поиска *k*-ой порядковой статистики на объеме данных *n* требуется получить точный ответ, например, за один проход, то мы столкнемся с проблемой, т.к. для этого потребуется $\Omega(n)$ количество памяти.

Данная работа рассматривает применение для решения этой проблемы алгоритма Мунро-Патерсона, параметризуемый числом проходов по объему данных.

Цель работы

Исследовать и реализовать алгоритм Мунро-Патерсона, позволяющий решать задачу поиска *k*-ой последовательной статистики, контролируя объем требуемой для этого памяти.

Задачи

1. Изучение литературы по алгоритмам обработки потоковых данных
2. Изучение литературы по алгоритму Мунро-Патерсона
3. Реализация алгоритма Мунро-Патерсона, параметризуемого числом проходов.
4. Проверка корректности реализации.

Реализация

Реализован основной метод, имплементирующий целевой алгоритм:

```
int munro_paterson(int* a, int an, int p, int rank);
```

и две вспомогательные функции:

```
double log2(double n);
void merge(std::vector<int> & v1, std::vector<int> &v2, int s);
```

Код с подробными комментариями доступен в приложении.

Тестирование

В ходе тестирования генерируется некоторое количество случайных последовательностей, в которых требуется найти элемент ранга `rank`. Последовательности обрабатываются целевым алгоритмом, результат работы которого проверяется на соответствие результату наивного метода (сортировка всей последовательности, обращение к элементу `rank+1`). Запросы к генерируемым последовательностям подаются в различных комбинациях всех параметров целевого метода (длина последовательности, количество проходов, ранг искомого элемента)

Кроме того, в сам метод заложена проверка соответствия количества используемой памяти к заданной параметром проходов оценке сложности алгоритма по памяти.

Тесты на корректность ответа и количества используемой памяти пройдены успешно. Код реализации алгоритма на языке C++ с подробными комментариями находится в приложении.

Приложение

Весь код реализации, результаты тестирования и текст этой работы доступен в репозитории по следующему адресу:

https://github.com/Allight7/munro_paterson/

munro_paterson.h

```
#include <iostream>
#include <cassert>
#include <cfloat>
#include <cmath>
#include <climits>
#include <vector>
#include <algorithm>
#include <exception>

#define forn(i, n) for (int i = 0; i < (int)(n); ++i)

inline double log2(double n){
	return log(n) / log(2);
}

//выбирает каждый второй элемент из отсортированного объединения двух векторов

inline void merge(std::vector<int> & v1, std::vector<int> &v2, int s){
	int j = 0, k = 0;
	std::vector<int> res(s);
	forn(i, s){
		k >=s || (j < s && v1[j] < v2[k]) ? j++ : k++;
		res[i] = k >=s || (j < s && v1[j] < v2[k]) ? v1[j++] : v2[k++];
	}
	v1 = res;
	return;
};

/*
	int* a - исходная последовательность
	int an - длина последовательности
	int p - число проходов по последовательности
	int rank - ранг искомого элемента в исходной последовательности

	return - искомый элемент с рангом rank
*/

inline int munro_paterson(int* a, int an, int p, int rank){
	assert(a && an > 0);
	//непустая последовательность
	assert(p > 0);
	//ненулевое кол-во проходов
	assert(rank >= 0 && rank < an);
	//валидное значение ранга
	if(an == 1) return *a;
```

```
        float m = an;
        float c = 2.;   //const addition
        int lower = INT_MIN;
        int upper = INT_MAX;


        float first_predicted_space = pow(m, 1./p) * c * pow(log2(m), 2. -
2./p);  //оценка максимума памяти для алгоритма в 1/sizeof(int).

        forn(i,p-1){
                if(m <= first_predicted_space) break;
                float space = pow(m, 1./(p-i)) * c * pow(log2(m), 2. - 2./(p-
i));
                int s = floor(space/log2(m));
                int t = ceil(log2(m/s));
                int m_taken = pow(2,t)*s;
                int t_2 = pow(2,t);
                int rank_offset = 0;
                if(i == 0){
                        int real_space = (t+1)*s;
                        std::cout << "p = " << p << ";\treal_space = " <<
real_space;
                }
                std::vector<int> v_curr(s);
                std::vector<bool> b_per_level(t+1, false);
                std::vector<std::vector<int>> v_per_level(t+1);
                forn(i,t+1)
                        v_per_level[i].resize(s);

                for(int curr = 0, taken = 0; taken < m_taken;){
                        if(curr < an){
                                forn(j,s){
                                        while(curr < an && (a[curr] < lower ||
a[curr] > upper)){
                                                if(a[curr] < lower)
++rank_offset;
                                                ++curr;
                                        }
                                        v_curr[j] = curr < an ? a[curr++] :
INT_MAX;
                                }
                                taken += s;
                                sort(v_curr.begin(), v_curr.end());
                        }
                        else{
                                v_curr = std::vector<int>(s, INT_MAX);
                                taken += s;
                        }
                        int level = 0;
                        while(b_per_level[level]){
                                merge(v_curr, v_per_level[level], s);
                                b_per_level[level++] = 0;
                        }
                        v_per_level[level] = v_curr;
                        b_per_level[level] = 1;
                }

                forn(i,t)
                        assert(!b_per_level[i]);
                assert(b_per_level[t]);

                v_curr = v_per_level[t];
```

```
                int l = floor(static_cast<float>(rank - rank_offset) / (t_2))
- t;                    //расчет индексов новых границ внутри t-сэмпла
                int u = ceil(static_cast<float>(rank - rank_offset) / (t_2));

                if(l >= 0 && l < s && v_curr[l] < upper && v_curr[l] > lower)
lower = v_curr[l];      //обновление границ с проверкой корректности
                if(u >= 0 && u < s && v_curr[u] > lower && v_curr[u] < upper)
upper = v_curr[u];

                int lbound = t_2 * (l);
                int ubound = t_2 * (u + t);
                m = ubound - lbound > 0 ? ubound - lbound : 1;
        }


        assert(first_predicted_space >= m);
        std::cout << "\t\tm = " << m << "\t\tpred_space = " <<
static_cast<int>(first_predicted_space) << std::endl;


        int rank_offset = 0;
        std::vector<int> res;
        for(int curr = 0; curr < an;){
                while(curr < an && (a[curr] < lower || a[curr] > upper)){
                        if(a[curr] < lower) ++rank_offset;
                        ++curr;
                }
                if(curr >= an)
                        break;
                res.push_back(a[curr++]);
        }

        sort(res.begin(), res.end());

        assert(rank - rank_offset >= 0);
        return res[rank-rank_offset];
}
```

test wa.cpp (проверка корректности найденной k-ой пор. стат.)

```
#include "munro_paterson.h"
#include <iostream>
#include <ctime>

const int MAX_SIZE = 10000;
int a[MAX_SIZE];
int b[MAX_SIZE];

int main(){
        int n[] = {1, 3, 10, MAX_SIZE}; //запуск на различных объемах данных
        int nn = 4;
        int p[] = {1, 2, 4, 6, 20};           //запуск с различным числом
проходов
        int pn = 5;

        srand(time(nullptr));

        forn(q,100){
                forn(i,nn){
                        forn(j,n[i]){
```

```
                                a[j] = rand() % INT_MAX;
                                b[j] = a[j];
                        }
                        std::sort(b,b+n[i]);
                        forn(k,pn){
                                int rank = 0;
        //запуск с различным числом проходов
                                assert(b[rank] == munro_paterson(a, n[i], p[k],
rank));

                                rank = rand() % n[i];
                                assert(b[rank] == munro_paterson(a, n[i], p[k],
rank));

                                rank = n[i] - 1;
                                assert(b[rank] == munro_paterson(a, n[i], p[k],
rank));
                        }
                }
        }
        std::cout << "Test WA: OK" << std::endl;

        return 0;
}
```

test_tl.cpp (проверка времени работы)

```
#include "munro_paterson.h"
#include <iostream>
#include <vector>

#define forn(i, n) for (int i = 0; i < (int)(n); i++)

void gen( int n, vector <int> &a ) {
  a.resize(n);
  forn(i, n)
    a[i] = rand();
}

const int maxN = 1e5;
Heap<int, INT_MAX, maxN> h;

int main() {
  vector <int> x;
  int n = maxN;
  forn(t, 10) { // number of tests
    // test extractMin
    gen(n, x);
    h.build(n, x.begin());
    forn(i, n)
      h.extractMin();
    // test add
    gen(n, x);
    h.clear();
    forn(i, n)
      h.add(x[i]);
  }
  fprintf(stderr, "Test for TL: OK. Time = %.2f\n", 1. * clock() /
CLOCKS_PER_SEC);
}
```