Algorithms
National Taiwan University, Fall 2022

Assignment 5

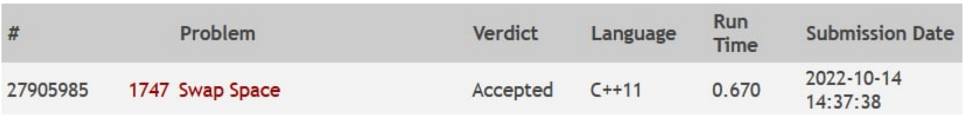
Yu-Chieh Kuo B07611039[†]

[†]Department of Information Management, National Taiwan University

Problem: Swap Space

Online Judge Testing

I tested my program on `UVa Online Judge`, an online automated judge system hosted by University of Valladolid, and received `Accepted` (see Fig. 1) to verify the correctness of my program. [1] The compiling environment is C++11 5.3.0, with compiler options `-lm -lcrypt -O2 -std=c++11 -pipe -DONLINE_JUDGE`.



#	Problem	Verdict	Language	Run Time	Submission Date
27905985	1747 Swap Space	Accepted	C++11	0.670	2022-10-14 14:37:38

Figure 1: Online judge submission record

Algorithm Design

To better illustrate the algorithm, I define a few auxiliary variables to ease note. Denote $\boldsymbol{d}_i\,(c_{i1}, c_{i2})$ by the i-th hard driver with the original capacity c_{i1} and the reformatted capacity c_{i2}, and $\mathcal{D} = \{\boldsymbol{d}_1, \boldsymbol{d}_2, \cdots, \boldsymbol{d}_n\}$ by the set of the given n hard drivers.

Capacity and Category

First I observe that the reformation might lead to both an increasing, identical, or decreasing capacity. Intuitively, it's better to start with hard drivers with increasing capacities after the reformation to create more vacant space for the reformation. Hence, we can separate hard drivers satisfying $c_{i1} < c_{i2}$ into the `increasing` cluster C_I and the remaining into the `decreasing` cluster C_D. As a side note, $\mathcal{D} = C_I \cup C_D$.

Reformation Order

Next task is to determine the order to reformat hard drivers. Here I simply adopt the greedy algorithm; that is, start from the hard driver with the smallest original capacity in the `increasing` cluster since it requires the least space to start. Thus, the order relationship $>_I$ for the `increasing` cluster C_I should follow the following properties:

[1] I genuinely thank my friend Andrew Shen (who does not enroll in this course) for his tremendous help of this assignment.

1. $\boldsymbol{d}_i >_I \boldsymbol{d}_j$ represents the hard driver i is reformatted earlier than j, $\forall \boldsymbol{d}_i, \boldsymbol{d}_i \in C_I$.
2. $\boldsymbol{d}_i >_I \boldsymbol{d}_j$ satisfies $c_{i1} < c_{j1} \vee \{c_{i1} = c_{j1} \wedge c_{i2} > c_{j2}\}$.
3. $>_I$ satisfies the completeness and transitivity.

Similar properties with proper modification can be imposed on the order relationship $>_D$ for the `decreasing` cluster C_D:

1. $\boldsymbol{d}_i >_D \boldsymbol{d}_j$ represents the hard driver i is reformatted earlier than j, $\forall \boldsymbol{d}_i, \boldsymbol{d}_i \in C_D$.
2. $\boldsymbol{d}_i >_D \boldsymbol{d}_j$ satisfies $c_{i2} > c_{j2} \vee \{c_{i2} = c_{j2} \wedge c_{i1} > c_{j1}\}$.
3. $>_D$ satisfies the completeness and transitivity.

We denote $C_I^\star$ and $C_D^\star$ by the sorted clusters, where $f_{>_I} : C_I \to C_I^\star$ and $f_{>_D} : C_D \to C_D^\star$.

Reformation

After clustering and ordering hard drivers, we then initiate the reformation by $C_I^\star$. The initial vacant capacity is 0. When iterating the sequence of $C_I^\star$ to reformat hard drivers, if the vacant capacity is not adequately large to complete reformation, we then purchase the exact size of extra storage $\varepsilon_k \in \mathbb{N}$, where k denotes the k-th purchase. Each iteration in $C_I^\star$ generates the additional vacant capacity. We then iterate the sequence of $C_D^\star$ after the completion of $C_I^\star$ with a positive vacant capacity size. Identically, we then purchase the exact size of extra storage if the vacant capacity is not large enough to complete reformation. Consequently, the minimized size of extra storage purchased is derived by $\sum_k \varepsilon_k$.

Complexity Analysis

Given n hard drivers as input, I then analyze the complexity of this algorithm to examine the efficiency.

Categorize: To separate the input $\mathcal{D}$ into C_I and C_D, the traverse of all input data is necessary. Hence, the complexity is $\Theta(n)$.

Sort: From the C++11/14 standard, `std::sort` is guaranteed to have $O(n \log n)$ on average. Therefore, to sort C_I and C_D requires $\Theta(n \log n)$.

Reformat: Reformatting all hard drivers in $\mathcal{D}$ requires n step. That is, the time complexity is $\Theta(n)$.

As a consequence, the overall complexity of the algorithm is

$$\Theta(n) + \Theta(n \log n) + \Theta(n) \in \Theta(n \log n).$$

Code Enclosure

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
```

```cpp

using namespace std;

#define NDEBUG

class UVa1747
{
public:
    UVa1747(const vector< pair<int, int> > &drive_capacity): drive_capacity(drive_capacity)
    {
#ifndef NDEBUG
        cout << "Drive Capacity: \n";
        for(size_t i = 0;i < drive_capacity.size(); ++i)
        {
            cout << drive_capacity[i].first << " " << drive_capacity[i].second << endl;
        }
        cout << "Drive Capacity --END--\n";
#endif //NDEBUG
    }
    long long int get_minimum_extra_capacity()
    {
        extra_capacity = 0;
        free_capacity = 0;
        classify_dc();
        sort_dc_increase();
        sort_dc_decrease();
#ifndef NDEBUG
        print_dc_ptr(dc_increase_ptr);
        print_dc_ptr(dc_decrease_ptr);
#endif //NDEBUG
        calc_extra_capacity(dc_increase_ptr);
        calc_extra_capacity(dc_decrease_ptr);
        return extra_capacity;
    }
private:
    const vector< pair<int, int> > &drive_capacity;
    vector< pair<int, int> const* > dc_increase_ptr, dc_decrease_ptr; // increase include equal
    long long int extra_capacity, free_capacity;
    void classify_dc()
    {
        for(size_t i = 0;i < drive_capacity.size(); ++i)
        {
            if(drive_capacity[i].first > drive_capacity[i].second)
                dc_decrease_ptr.push_back(&drive_capacity[i]);
            else
                dc_increase_ptr.push_back(&drive_capacity[i]);
        }

    }
    void sort_dc_increase()
    {
        sort(dc_increase_ptr.begin(),
                dc_increase_ptr.end(),
                [this](const pair<int, int>* const a, const pair<int, int>* const b)
                {
                    if(a->first == b->first) return a->second > b->second;
                    else return a->first < b->first;
                });
    }
    void sort_dc_decrease()
    {
        sort(dc_decrease_ptr.begin(),
                dc_decrease_ptr.end(),
                [this](const pair<int, int>* const a, const pair<int, int>* const b)
                {
                    if(a->second == b->second) return a->first > b->first;
                    else return a->second > b->second;
                    //if(a->first == b->first) return a->second < b->second;
                    //else return a->first > b->first;
                });
    }
    void calc_extra_capacity(const vector< pair<int, int> const* > & dc_ptr)
    {
        for(size_t i = 0;i < dc_ptr.size(); ++i)
        {
            if(dc_ptr[i]->first > free_capacity)
            {
                extra_capacity += dc_ptr[i]->first - free_capacity;
                free_capacity += dc_ptr[i]->first - free_capacity;
            }
            free_capacity -= dc_ptr[i]->first;
            free_capacity += dc_ptr[i]->second;
#ifndef NDEBUG
^^I    cout << "extra: " << extra_capacity << " free: " << free_capacity << endl;
#endif //NDEBUG
        }
    }
#ifndef NDEBUG
    void print_dc_ptr(const vector< pair<int, int> const* > & dc_ptr)
    {
        cout << "Drive Capacity: \n";
        for(size_t i = 0;i < dc_ptr.size(); ++i)
        {
            cout << dc_ptr[i]->first << " " << dc_ptr[i]->second << endl;
        }
        cout << "Drive Capacity --END--\n";
    }
```

```
#endif //NDEBUG
};

int main(int argc, char* argv[])
{
//    ios::sync_with_stdio(false);
//    cin.tie(0);

    int t;
    while(cin >> t)
    {
        vector< pair<int, int> > drive_capacity;
        while(t--)
        {
            int drive_capacity_old, drive_capacity_new;
            cin >> drive_capacity_old >> drive_capacity_new;
            drive_capacity.push_back( make_pair(drive_capacity_old,
 ↪  drive_capacity_new) );
        }

        UVa1747 uva_1747(drive_capacity);
        cout << uva_1747.get_minimum_extra_capacity() << endl;
    }

    clog << "Time used = " << (double)clock() / CLOCKS_PER_SEC << endl;
    return 0;
}


```