Minimum Average Waiting Time

locked

by tuananhnb93

Editorial by Tanzir5

This problem can be solved using a **greedy approach.**

We have to remember that Tieu cannot know about future orders beforehand. So he will have to serve one of the orders that are currently waiting to be served. But how should he decide which order to serve now ?

This is where the greedy approach comes. He should always serve the order which has minimum cooking time i.e. the order with the minimum L. Let's prove this approach to be the optimal strategy using proof by contradiction.

Proof by contradiction:

Let's say currently we have X orders that need to be served. Let L_{min} be the cooking time of the order with the minimum cooking time. Then if we decide to serve this order now, it will contribute $(X \times L_{min} + C_1)$ time to the total waiting time, where C_1 is the time waited by new customers who have placed orders while we were cooking this order.

Now let's say, it was actually optimal to serve the j^{th} order where $L_j > L_{min}$. Then this will contribute $(X \times L_j + C_2)$ time to the total waiting time, where C_2 is the time waited by new customers who have placed orders while we were cooking the j^{th} order.
Now we have,

1. $L_j > L_{min}$
2. $C_2 > C_1$ (Since the j^{th} order takes more time to be cooked, the number of new customers who have placed order in the mean time will be higher or equal to the first scenario and they will have to wait more time as $L_j > L_{min}$)

So we can conclude that,

$$(X \times L_{min} + C_1) < (X \times L_j + C_2)$$

But this is a contradiction to the claim that serving the j^{th} order with $L_j > L_{min}$, is optimal. Hence, using proof by contradiction, serving the order with the lowest cooking time is proved to be the optimal strategy.

Storing the data efficiently:

At first, we need to sort the orders according to increasing order time. An array is sufficient here. But then we need to store the currently available orders according to increasing cooking time. We can use a **min heap** for this purpose. We will loop through the

Statistics

Difficulty: Hard
Time Complexity: O(N log (N)) Required
Knowledge: Priority Queue, Heaps
Publish Date: Jun 30 2014

array. Upon arriving at the i^{th} order, we will insert it into the min heap if its order time is less than or equal to the current time. Otherwise, we will keep serving from the top of the heap until current time becomes larger than or equal to the order time of the i^{th} order or the heap becomes empty. We will update the current time while inserting an order into the heap (if needed) and also while serving an order.

Editorialist's solution

C++

```
#include<bits/stdc++.h>
using namespace std;

typedef long long int LL;

int n;
struct order{
    LL T, L;
}customer[100010];

bool operator < (order a, order b)
{
    return a.T < b.T;
}

bool compare(order a, order b)
{
    return a.L > b.L;
}
priority_queue<order, vector<order>, function<bool(order, order)>> min_heap
(compare);

void add_order(order current_order, LL &current_time)
{
    if(min_heap.empty() == true)
        current_time = max(current_time, current_order.T);
    min_heap.push(current_order);
}

LL serve_order(LL &current_time)
{
    order current_order = min_heap.top();
    min_heap.pop();
    current_time += current_order.L;
    return current_time - current_order.T;
}

LL solve(int n)
{
    LL current_time = 0, total_waiting_time = 0;
    for(int i = 1; i<=n; i++)
    {
        if(i == 1 || customer[i].T <= current_time)
            add_order(customer[i], current_time);
        else if(min_heap.empty() == false)
        {
            while(min_heap.empty() == false && customer[i].T > current_time
)
                total_waiting_time += serve_order(current_time);
            add_order(customer[i], current_time);
        }
    }
    while(min_heap.empty() == false)
        total_waiting_time += serve_order(current_time);
    return total_waiting_time/n;
}
int main()
{
    cin >> n;
    for(int i = 1; i<=n; i++)
        cin >> customer[i].T >> customer[i].L ;
```

```
    sort(customer+1, customer+n+1);
    cout << solve(n) << endl;
    return 0;
}
```

 Set by tuananhnb93

Problem Setter's code :

```
/*
Solution: pick the order that takes shortest time to cook.
*/
#include <cstdio>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include <queue>
#include <stack>
#include <set>
#include <map>
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <string>
#include <memory.h>
#include <sstream>
#include <complex>
#include <cassert>
#include <climits>

#define REP(i,n) for(int i = 0, _n = (n); i < _n; i++)
#define REPD(i,n) for(int i = (n) - 1; i >= 0; i--)
#define FOR(i,a,b) for (int i = (a), _b = (b); i <= _b; i++)
#define FORD(i,a,b) for (int i = (a), _b = (b); i >= _b; i--)
#define FORN(i,a,b) for(int i=a;i<b;i++)
#define FOREACH(it,c) for (__typeof((c).begin()) it=(c).begin();it!=(c).end
();it++)
#define RESET(c,x) memset (c, x, sizeof (c))

#define PI acos(-1)
#define sqr(x) ((x) * (x))
#define PB push_back
#define MP make_pair
#define F first
#define S second
#define Aint(c) (c).begin(), (c).end()
#define SIZE(c) (c).size()

#define DEBUG(x) { cerr << #x << " = " << x << endl; }
#define PR(a,n) {cerr<<#a<<" = "; FOR(_,1,n) cerr << a[_] << ' '; cerr <<en
dl;}
#define PR0(a,n) {cerr<<#a<<" = ";REP(_,n) cerr << a[_] << ' '; cerr << end
l;}
#define LL long long

using namespace std;

#define maxn 100111
int n;
pair<int, int> order[maxn];
multiset<pair<int, int> > available_order;

bool getChar(char& c) {
    return (scanf("%c", &c) == 1);
}

void nextInt(long long& u, char endline, int l, int r) {
    int sign = 1;
    long long sum = 0;
    char c;
```

```
    assert(getChar(c));
    if (c == '-') sign = -1;
    else {
        assert('0' <= c && c <= '9');
        sum = (c - '0');
    }

    int cnt = 0;

    for (int i = 1; i <= 15; i++) {
        assert(getChar(c));
        if (c == endline) break;
        assert('0' <= c && c <= '9');
        sum = sum * 10 + (c - '0');
        cnt ++;
        assert(cnt <= 15);
    }

    sum = sign * sum;
    assert (l <= sum && sum <= r);
    u = sum;
}

void nextInt(int& u, char endline, int l, int r) {
    long long tmp;
    nextInt(tmp, endline, l, r);
    u = tmp;
}

int main() {
    //freopen("a.in", "rb", stdin); freopen("a.out", "wb", stdout);
    nextInt(n, '\n', 1, 1e5);
    for (int i = 0; i < n; i++) {
        nextInt(order[i].first, ' ', 0, 1e9);
        nextInt(order[i].second, '\n', 1, 1e9);
    }

    //sort the orders in the increaseing order of the submit time.
    sort(order, order + n);

    //we maintain the available_order set, which stores the submitted order
in the increasing order of the time to cook
    int pos = 0;//pos is the next order to be considered to be push in avai
lable_order
    long long current_time = order[0].first;
    //the total waiting time may be out of 64bit integer range so we mainta
in two values average and mod so that total waiting time = average * n + mo
d.
    long long average = 0, mod = 0;

    for (int i = 0; i < n; i++) {
        //push all submitted orders into available_order
        while (pos < n && order[pos].first <= current_time) {
            //note that in available_order, the order is stored in the incr
easing order of time to cook
            available_order.insert(make_pair(order[pos].second, order[pos].
first));
            pos++;
        }

        //take out the order that has a shortest time to cook
        pair<int, int> item = *available_order.begin();

        available_order.erase(available_order.begin());

        //update current_time, average and mod
        current_time += item.first;
        //this is the waiting_time of the current order
        long long waiting_time = current_time - item.second;

        mod += waiting_time % n;
        average += waiting_time / n;

        if (mod >= n) {
            average += mod / n;
```

```
            mod %= n;
        }
    }

    cout << average << endl;
    char __c;
    assert(!getChar(__c));
    return 0;
}
```

 Tested by darkshadows

Problem Tester's code :

```
#include<bits/stdc++.h>
#define assn(n,a,b) assert(n>=a && n<=b)
#define F first
#define S second
#define mp make_pair
using namespace std;
int main()
{
    int n,i;
    vector < pair < int, int > > ar;
    long long cur=0,ans=0;
    cin >> n;
    ar.resize(n);
    assn(n,1,1e5);
    for(i=0; i<n; i++)
    {
    cin >> ar[i].F >> ar[i].S;
    assn(ar[i].F, 0, 1e9);
    assn(ar[i].S, 1, 1e9);
    }

    sort(ar.begin(),ar.end());
    priority_queue< pair < int, int > , vector< pair < int, int> >, greater
<  pair < int, int > > > mheap;
    i=1;
    mheap.push(mp(ar[0].S,ar[0].F));
    cur=ar[0].F;
    while(!mheap.empty() || i<n)
    {
    while(i<n && ar[i].F<=cur)
    {
        mheap.push(mp(ar[i].S,ar[i].F));
        i++;
    }
    if(mheap.empty() && i<n)
    {
        cur=ar[i].F;
        mheap.push(mp(ar[i].S,ar[i].F));
        i++;
    }
    pair < int, int > p = mheap.top();
    mheap.pop();
    cur+=(long long)(p.F);
//  printf("cur:%d cook-time:%d coming-time:%d\n",cur,p.F,p.S);
    ans+=(long long)(cur)-(long long)(p.S);
    }
    cout << ans/n << endl;
    return 0;
}
```

Find the Running Median

locked

by amititkgp

Editorial by Tanzir5

This problem can be solved using two **heaps**.

Let's say we are taking input for the i^{th} number. If somehow we had the previous $(i-1)$ numbers sorted, then we can easily add the i^{th} number in $O(n)$ time to the appropriate place in our list so that the list remains sorted and find the median in $O(1)$ time. But we cannot afford to add every number with $O(n)$ complexity.

So let's say we had two sorted arrays. The first array holds the smaller half of the numbers in decreasing order. The second array contains the larger half of the numbers in increasing order. Now, after taking input for the i^{th} number, we can easily decide which half the i^{th} number belongs to and add it there in the appropriate place. If any of the arrays becomes much larger than the other array, we can remove the first element from that array and add it to the appropriate place of the other array. We have reduced time complexity by some constant factor. But obviously, that is not enough.

Now let's look at the indices of the array where we need to access at any moment for getting the median. If the total number of elements is odd, then we need the first element of the array with the higher number of elements. If the total number of elements is even, then we need the average of the first elements of both the arrays. So the only indices we need to access while getting the median and adding the elements are the first index of both of the arrays. So which data structure can help in storing data in sorted order and accessing the top element efficiently? The answer is a **heap**.

We will use a **max heap** for storing data of the smaller half of the numbers and a **min heap** for storing data of the larger half of the numbers. Now let's see how we will add the numbers and get the medians.

Adding the i^{th} number:

While adding the i^{th} number we will check the following conditions and add accordingly:

1. If the i^{th} number is greater than or equal to the max element of the max heap then it surely belongs to the larger half of the numbers i.e. the min heap. So we will add it there.
2. Otherwise, we will add it to the max heap.

Now it may happen that one of the heap becomes much larger than the other if we add the numbers in this way. So to stop this situation from taking place, we need to check the size of the heaps after adding every number. If the difference between the number of elements of the two heaps becomes more than one, then we need to pop the top element

Statistics

Difficulty: Hard
Time Complexity: O(nlogn) Required
Knowledge: Heaps, Priority Queue
Publish Date: May 07 2015

from the heap with more elements and push that element to the other heap. If we work in this way, the difference can never be more than one.

Getting the median:

To get the median after adding the i^{th} number we will check if i is odd or even. If i is odd, then surely one of the heap has one more element than the other. The median will be the top element of that heap then. If i is even, then the two heaps must have the same number of elements. So the median will be the average of the top elements of the two heaps.

Editorialist's solution:

```
#include<bits/stdc++.h>
using namespace std;

priority_queue<int, vector<int>, greater <int> > min_heap;
priority_queue<int> max_heap;
void add(int a)
{
    if( max_heap.size() && a >= max_heap.top())
        min_heap.push(a);
    else
        max_heap.push(a);

    if(abs(max_heap.size() - min_heap.size()) > 1)
    {
        if(max_heap.size() > min_heap.size())
        {
            int temp = max_heap.top();
            max_heap.pop();
            min_heap.push(temp);
        }
        else
        {
            int temp = min_heap.top();
            min_heap.pop();
            max_heap.push(temp);
        }
    }
}

double get_median()
{
    int total = min_heap.size() + max_heap.size();
    double ret;
    if(total%2 == 1)
    {
        if(max_heap.size() > min_heap.size())
            ret = max_heap.top();
        else
            ret = min_heap.top();
    }
    else
    {
        ret = 0;
        if(max_heap.empty() == false)
            ret += max_heap.top();
        if(min_heap.empty() == false)
            ret += min_heap.top();
        ret/=2;
    }
    return ret;
}

int main()
{
    cout << setprecision(1) << fixed;
    int n, a;
    cin >> n;
```

```
    for(int i = 1; i<=n; i++)
    {
        cin >> a;
        add(a);
        cout << get_median() << endl;
    }
}
```

QHEAP1

locked

by dcod5

Editorial by dcod5

This is a basic heap data structure question. The operations of a heap can be easily understood here.

Keep in mind that elements are distinct so we can keep their index (i.e. location in the heap) in a map and then perform deletions. Look at the code below for better understanding.

Please note that the question can also be done easily using in-built sets data structure, but it is intended to be done via heaps.

Statistics

Difficulty: Easy
Time Complexity: O(Q*logQ)
Required Knowledge: Heaps
Publish Date: Dec 16 2015

Set by dcod5

Problem Setter's code :

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

// Map to maintain the index of values in the heap.
map<int, int> value_index;
int heap[500002], heap_size = 0;

void insert_val(int val)
{
    if(heap_size == 0)
    {
        heap[++heap_size] = val;
        value_index[val] = heap_size;
        return;
    }
    heap[++heap_size] = val;
    value_index[val] = heap_size;
    int iter = heap_size;
    while(iter > 1)
    {
        if(heap[iter] < heap[iter/2])
        {
            value_index[heap[iter]] = iter/2;
            value_index[heap[iter/2]] = iter;
            int temp = heap[iter];
            heap[iter] = heap[iter/2];
            heap[iter/2] = temp;
            iter /= 2;
        }
        else
```

```
            break;
    }
}

void delete_val(int val)
{
    int index = value_index[val];
    value_index[val] = 0;
    value_index[heap[heap_size]] = index;
    heap[index] = heap[heap_size--];
    while(true)
    {
        int left_child = 2*index, right_child = 2*index + 1;;
        if(left_child <= heap_size)
        {
            if(right_child <= heap_size)
            {
                if(heap[index] > heap[left_child] || heap[index] > heap[rig
ht_child])
                {
                    int swap_index = (heap[left_child] < heap[right_chil
d])? left_child:right_child;
                    value_index[heap[swap_index]] = index;;
                    value_index[heap[index]] = swap_index;;
                    int temp = heap[index];
                    heap[index] = heap[swap_index];
                    heap[swap_index] = temp;
                    index = swap_index;
                }
                else
                    break;
            }
            else
            {
                if(heap[index] > heap[left_child])
                {
                    value_index[heap[left_child]] = index;
                    value_index[heap[index]] = left_child;
                    int temp = heap[index];
                    heap[index] = heap[left_child];
                    heap[left_child] = temp;
                    index = left_child;
                }
                else
                    break;
            }
        }
        else
            break;
    }
}

int main()
{
    int queries;
    cin>>queries;
    while(queries--)
    {
        int type, val;
        cin>>type;
        if(type == 1) // insert
        {
            cin>>val;
            insert_val(val);
        }
        else if(type == 2) // delete
        {
            cin>>val;
            delete_val(val);
        }
        else
        {
            cout<<heap[1]<<endl;
        }
    }
```

```
    return 0;
}
```

Jesse and Cookies

locked

by vatsalchanana

Editorial by vatsalchanana

This problem can be solved using a min heap. Initially, we add all the cookies to the heap. We repeatedly pop 2 cookies with the least sweetness and combine them and add the resulting sweetness ($1\times$ *least sweet cookie* + $2\times$ *2nd least sweet cookie*) to the heap till the sweetness of minimum becomes $\geq K$.

Statistics

Difficulty: Easy
Time Complexity: O(n * log(n)) Required
Knowledge: Priority Queue
Publish Date: Jan 19 2016

Set by vatsalchanana

Problem Setter's code :

```
#include<iostream>
#include<vector>
#include<cstdio>
#include<algorithm>
#include<utility>
#include<set>
#include<map>
#include<cstring>
#include<cmath>
#include<string>
#include<cstdlib>
#include<queue>

using namespace std;

int main()
{
    #define int long long
    int n,k;
    cin>>n>>k;
    priority_queue<int, std::vector<int>, std::greater<int> > pq;
    for(int i=0;i<n;i++)
    {
        int val;
        cin>>val;
        pq.push(val);
    }
    int count=0;
    bool ans=true;
    while(1)
    {
        if(pq.empty())
        {
            ans=false;
            break;
        }
        int a1=pq.top();
        pq.pop();
        if(a1>=k)
```

```
        {
            break;
        }
        if(pq.empty())
        {
            if(a1<k)
            {
                ans=false;
            }
            break;
        }

        int a2=pq.top();
        pq.pop();

        int nv=a1+2*a2;
        count++;
        pq.push(nv);

    }
    if(ans)
        cout<<count;
    else
        cout<<"-1";
    cout<<endl;
}
```