**RMQ**

|  |
| --- |
| **Съдържание**  [[скриване](javascript:toggleToc())]   * [1 Дефиниция на проблема](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.94.D0.B5.D1.84.D0.B8.D0.BD.D0.B8.D1.86.D0.B8.D1.8F_.D0.BD.D0.B0_.D0.BF.D1.80.D0.BE.D0.B1.D0.BB.D0.B5.D0.BC.D0.B0) * [2 Решения](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.A0.D0.B5.D1.88.D0.B5.D0.BD.D0.B8.D1.8F)   + [2.1 Статично](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.A1.D1.82.D0.B0.D1.82.D0.B8.D1.87.D0.BD.D0.BE)     - [2.1.1 Алгоритъм](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.90.D0.BB.D0.B3.D0.BE.D1.80.D0.B8.D1.82.D1.8A.D0.BC)     - [2.1.2 Примерен Код](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.9F.D1.80.D0.B8.D0.BC.D0.B5.D1.80.D0.B5.D0.BD_.D0.9A.D0.BE.D0.B4)   + [2.2 Позволяващо промяна](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.9F.D0.BE.D0.B7.D0.B2.D0.BE.D0.BB.D1.8F.D0.B2.D0.B0.D1.89.D0.BE_.D0.BF.D1.80.D0.BE.D0.BC.D1.8F.D0.BD.D0.B0)     - [2.2.1 Алгоритъм](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.90.D0.BB.D0.B3.D0.BE.D1.80.D0.B8.D1.82.D1.8A.D0.BC_2)     - [2.2.2 Примерен Код](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.9F.D1.80.D0.B8.D0.BC.D0.B5.D1.80.D0.B5.D0.BD_.D0.9A.D0.BE.D0.B4_2)   + [2.3 Други алгоритми](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.94.D1.80.D1.83.D0.B3.D0.B8_.D0.B0.D0.BB.D0.B3.D0.BE.D1.80.D0.B8.D1.82.D0.BC.D0.B8) * [3 Приложения](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.9F.D1.80.D0.B8.D0.BB.D0.BE.D0.B6.D0.B5.D0.BD.D0.B8.D1.8F)   + [3.1 Lowest Common Ancestor (LCA)](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#Lowest_Common_Ancestor_.28LCA.29)     - [3.1.1 Дефиниция на проблема](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.94.D0.B5.D1.84.D0.B8.D0.BD.D0.B8.D1.86.D0.B8.D1.8F_.D0.BD.D0.B0_.D0.BF.D1.80.D0.BE.D0.B1.D0.BB.D0.B5.D0.BC.D0.B0_2)     - [3.1.2 Свеждане към RMQ](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.A1.D0.B2.D0.B5.D0.B6.D0.B4.D0.B0.D0.BD.D0.B5_.D0.BA.D1.8A.D0.BC_RMQ)   + [3.2 Примерна Задача](http://judge.openfmi.net:9080/mediawiki/index.php/RMQ#.D0.9F.D1.80.D0.B8.D0.BC.D0.B5.D1.80.D0.BD.D0.B0_.D0.97.D0.B0.D0.B4.D0.B0.D1.87.D0.B0) |

**Дефиниция на проблема**

*RMQ* е съкращение от *Range Minimum Query*.

Даден е масив от N елемента. Извършват се запитвания за минималният елемент измежду елементите с индекси между i и j. Възможно е в даден момент стойността на елемент да бъде променена. Ако има такава операция имаме динамично RMQ, ако такава оперция липсва - статично RMQ.

**Решения**

Съществуват няколко подхода за реализиране на статично и динамично RMQ. Сложността на RMQ се разделя на две части. Тъй като в различните алгоритми предварително се извършват някакви изчисления с цел по-бързото отговаряне на заявки се определя сложност на предварителните изчисления. Важна е и сложността, с която се отговаря на заявки. Ето защо в тази статия ще отбелязваме сложността на един алгоритъм като двойка <math>(f(n), g(n))</math>. f(n) обозначава сложността на преварителните изчисления, а g(n) сложността за отговаряне на една заявка, като броят на елементите е n.

За статично RQM най-бързият известен алгоритъм е със сложност (<math>O(n),O(1)</math>). За съжаление неговате реализация е прекалено тежка, за да бъде лесно реализиран по време на състезание. Значително по-лесен като реализация е подхода със сложност <math>(O(nlogn),O(1))</math>.

Ще представим и алгоритъм за решаване на динамично RMQ като сложността е <math>(O(n),O(logn)</math>.

**Статично**

Тук ще опишем решение, което има сложност <math>(O(n \log n),\ O(1))</math>. Реализацията му е сравнително лесна и е подходящ за състезание.

**Алгоритъм**

Решението се базира на подхода динамично оптимиране. Идеята е да се пресметне предварително минималният елемент във всички интервали с дължина степен на двойката. С M[i][j] обозначаваме индекса на минималният елемент в интервал с начало елемента на позиция i и дължина 2j. За да се пресметне стойността за един елемент M[i][j] въпросният интервал се разделя на две равни части. Тези два подинтервала са [i, i+2j-1-1] и [i+2j-1, i+2j-1]. Минималните елементи в тези подинтервали са на позиции M[i][j-1] и M[i+2j][j-1]. Ето защо M[i][j] = min(A[M[i][j-1]], A[M[i+2j][j-1]]), като A е масивът с елементи. Стойностите на M се пресмятат в нарастващ ред на параметъра j. Така се гарантира, че когато се пресмята M[i][j] стойносттите за подинтервалите, на които се разделя големият интервал са вече пресметнати.

При отговаряне на заявка се подава някакъв интервал [i, j], в който се търси индекса на минималният елемент. С така направените предварителни изчисления отговарянето на заявки става за константно време. Намира се максималното число, което е степен на двойката и не е по-голямо от дължината на зададеният интервал. Нека това число е 2t. Интервалите [i, i+2t-1] и [j-2t+1, j] се застъпват. От тук следва, че ако се намерят минималните елементи в тези два интервала и се вземе по-малкият от двата той ще е минимален за търсеният интервал [i, j]. Индексите на минималните елементи в двата интервала са M[i,t] и M[j-2t+1, t]. Отговора на заявката е min(A[M[i,t]], A[M[j-2t+1, t]]).

**Примерен Код**

int Log2(int x); //in Numbers.cpp

int PairMin(int k1, int v1, int k2, int v2){

return k1 < k2 ? v1 : v2;

}

// finds the last minimum

// in PairMin replace < with <= for the first minimum

// with > for the last maximum

// or with >= for the first maximum

// RMQ build O(n log n), query O(1) by Milo

struct StaticRmq

{

vector<int> a;

vector< vector<int> > p;

StaticRmq(vector<int> const& \_a){

a = \_a;

int n = (int)a.size();

int ln = Log2(n);

p.clear();

p.resize(ln + 1, vector<int>(n));

for(int i = 0; i < n; i++) p[0][i] = i;

for(int i = 1; i <= n; i++)

for(int j = 0; j < n - (1 << i) + 1; j++)

{

p[i][j] = PairMin(a[p[i-1][j]], p[i-1][j],

a[p[i-1][j + (1 << (i - 1))]], p[i-1][j + (1 << (i - 1))]);

}

}

// Index of the minimum

int Rmq(int beg, int end)

{

assert(beg < end);

int ln = Log2(end - beg);

return PairMin(

a[p[ln][beg]],

p[ln][beg],

a[p[ln][end - (1 << ln)]],

p[ln][end - (1 << ln)]);

}

};

**Позволяващо промяна**

Възможно е едновременно със запитванията за минимален елемент в даден интервал да има промяна стойността на даден елемент в масива. Този вид задача може да се реши със сложност (<math>O(n)</math>, <math>O(\log n)</math>). Промяната на стойност на елемент е със сложност <math>O(\log n)</math>.

**Алгоритъм**

За реализацията на този алгоритъм ще използваме двоично дърво. Елементите на масива се разполагат в листата му. Тъй като листата са степен на двойката на брой може да е нужно в някои листа да се запишат служебни стойности. След това върховете на дървото се обхождат по нива като се започва от нивото точно на листата. Във всеки връх се записва по-малката от двете стойности, които се съдържат в двата му наследника. Тъй като при тази операция всеки връх на дървото се обхожда по един път, а броят на върховете е линеен по броят на елементите в масива, то сложността на тези предварителни изчисления е линейна.

При запитване за минимален елемент в даден интервал [i, j] се намират листата в дървото, които отговарят на двата края на интервала. Започва се едновременно "изкачване" нагоре по дървото от тези две листа докато се достигне един и същ връх. При изкачването от един връх се продължава към родителя му и се поддържа минимален елемент до момента. При изкачването от листото с индекс i ако поредният разглеждан връх е ляв наследник на родителя си се сравнява досегашният минимален елемент със стойността в брата на текущият връх. След това се продължава нагоре към родителя. Същото важи и при изкачването от листото с индекс j. Разликата там е че ако текущо разглеждания връх е десен наследник на родителя си се сравнява досега намереният минимум с брата на този връх. При достигане на един и същ връх се взема по-малката от двете намерени стойности. Това е минималният елемент в дадения интервал. Сложността е логаритмична, тъй като при всяка стъпка от изкачването се издигаме към връх с височина, която е с 1 по-голяма от тази на предходния връх. Тъй като височината на дървото е <math>log n</math> общият брой стъпки при двете изкачвания не е по-голям от <math>2\*log n</math>.

При заявка за промяна стойността на един елемент първо се промения стойността в съответното листо. След това се прави изкчаване нагоре по дървото докато се достигне корена на дървото. За всеки обходен връх за негова нова стойност се взема минимума от стойностите в двата му наследника. Сложността отново е логаритмична.

**Примерен Код**

// RMQ build O(n), query O(log n), update O(log n) by Milo

struct DynamicRmq{

vector<int> a;

vector< vector<int> > p;

DynamicRmq(vector<int> const& val){

a = val;

p.clear();

int t = (int)a.size();

do{

p.push\_back(vector<int>(t));

t /= 2;

}

while(p.back().size() > 1);

for(int i = 0; i < p[0].size(); i++) p[0][i] = i;

for(int i = 1; i < p.size(); i++)

for(int j = 0; j < p[i].size(); j++)

{

p[i][j] = PairMin(

a[p[i-1][j\*2]],

p[i-1][j\*2],

a[p[i-1][j\*2+1]],

p[i-1][j\*2+1]);

}

}

void Set(int pos, int v){

a[pos] = v;

for(int i = 1; i < p.size(); i++)

{

if((pos >> i) < (int)p[i].size()){

p[i][pos >> i] = PairMin(

a[p[i-1][(pos>>i)\*2]],

p[i-1][(pos>>i)\*2],

a[p[i-1][(pos>>i)\*2+1]],

p[i-1][(pos>>i)\*2+1]);

}

}

}

int Rmq(int beg, int end){

int l = 0;

vector<int> checkBeg;

vector<int> checkEnd;

int res = beg;

while(beg != end){

if(beg & 1){

checkBeg.push\_back(p[l][beg]);

++beg;

}

if(end & 1){

--end;

checkEnd.push\_back(p[l][end]);

}

beg /= 2;

end /= 2;

++l;

}

for(int i = 0; i < checkBeg.size(); i++)

res = PairMin(a[res], res, a[checkBeg[i]], checkBeg[i]);

for(int i = checkEnd.size()-1; i >= 0; i--)

res = PairMin(a[res], res, a[checkEnd[i]], checkEnd[i]);

return res;

}

};

**Други алгоритми**

Съществува алгоритъм за статично RMQ със сложност (<math>O(n), O(1)</math>). Неговата реализация е сложна и трудно може да бъде написан по време на състезание. Тук ще опишем идеята на този алгоритъм.

Елементите от даденият масив се разполагат в така нареченото Декартово дърво (Cartesian tree). Това е двоично дърво, в корена на което се съдържа минималният елемент в масива. Левият наследник на корена е корен на дървото съответстващо на подмасива образуван от елементите наляво от минималния. Десният наследник е корен на дървото съответсващо на подмасива от елементите, които са надясно от минималният елемент. Построяването на такова дърво се прави като първо се построява дърво с един връх съдържащ първият елемент от масива. Построяването продължава като на всяка стъпка се добавя следващият елемент от масива. Добавянето става като се обхожда дървото като се започва от най-десният връх и се върви нагоре докато текущо разглеждания елемент е по-голям от този, който се опитваме да добавим. Когато се стигне до връх V, който съдържа не по-голяма стойност или пък е достигнат корена на дървото, новият връх се поставя на мястото на този връх V, а V става ляв наследник на новодобавеният връх в дървото. Построяването на това дърво става с линейна сложност. Това е така защото полученото дърво има n-1 ребра, като n е броя на елементите в масива и съответно броя на върховете в дървото. Когато за поставянето на един връх в дървото се обхожат някакви ребра те са част от пътя от корена до най-десният елемент. След поставянето на новият връх обаче тези ребра вече не са част от този път. Следователно всяко ребро се обхожда само по веднъж.

В така построеното дърво ще дефинираме LCA(i,j) да бъде върха, който е по пътя от корена и до двата върха и е най-далече от самия корен. Можем да твърдим, че RMQ(i,j) = LCA(i,j), като RMQ(i,j) е минималният елемент в интервала между числата с индекси i и j. Следователно, когато се търси минимален елемент в някакъв интервал в масива от числа задачата може да се сведе до намиране на LCA на два върха в дърво. Сега ще опишем как се решава LCA отново с линейна сложност.

Първо се прави така нареченото Ойлерово обхождане на дървото (Euler tour). Прави се обхождане в дълбочина на дървото като се започва от корена и всеки път когато се обхожда един връх той се записва на върха на един списък L. В Този списък има <math>2\*n-1</math> записани върха. Това е така, защото при преминаване по едно ребро се прави един запис в списъка, а ребрата се обхождат два пъти, по веднъж във всяка посока. Освен това в началото в списъка L се записва корена на дървото. Някои върхове може да фигурират в този списък по повече от веднъж. Нека за всеки връх i стойността F[i] показва най-лявата позиция в списъка, където се появява дадения връх. Трябва да се направи един масив H, в който H[i] показва дълбочината на върха L[i] в дървото. Сега чрез RMQ върху полученият списък може да се намери LCA в дървото. Когато има запитване LCA(i,j), отговора може да се намери чрез RMQ(F[i],F[j]) за списъка H. Така за някакви два върха i и j се намира точно най-ниско разположения връх, който е част от пътя от корена и до двата зададени върха. А това е отговора на LCA(i,j). Сега остава да се направи RMQ върху масива L с линейна подготовка и константно отговаряне на заявките, за да се получи линейно LCA и съответно да се получи това, което се искаше в началото - RMQ с линейна подготовка и константно отговаряне на заявки. Изглежда, че решаването на задачата се свежда дo самата нея. Това обаче не е така. Може да се забележи, че в масива L два съседни елемента се различават с 1. Тази особеност на стойностите в масива L може да се използва. Първо елементите в масива L се разделят на групи, като във всяка група има по <math>\frac{\log n}{2}</math> елемента. С линейна сложност може да се намери кой е минималният елемент във всяка група. След това се прави RMQ със сложност (<math>O(nlogn), O(1)</math>) върху масива от тези минимуми за получените групи. Сложността на това RMQ е линейна като цяло. Това е така, защото броят на групите е <math>\frac{\log n}{2}</math>. Когато има запитване за минималният елемент в някакъв интервал от масива L този интервал може да бъде разделен на три части. Ако в този интервал се съдържат някакъв брой цели групи за тях с константна сложност можем да намерим минимума, използвайки пресметнатите стойности от RMQ, което преди малко описахме. В двата края на интервала евентуално остават части от блокове. Възможно е в търсеният интервал да не се съдържа нито една цяла група и тогава запитването ще е към някаква част от някой от блоковете, на които сме разделили масива L. Ето защо се налага да може с константна сложност да се отговаря на въпроси кой е най-малкия елемент в даден подинтервал на един блок. Тук ще използваме факта, че два съседни елемента се различават с 1 в резултат на обхождането на дървото. Всеки блок може да се представи чрез двоичен вектор, който показва дали разликата между всеки два съседни елемента е -1 или 1. Освен това е нужно да се пази стойността например на първият елемент, за да може да се възстанови целия блок. Различните двоични вектори, които могат да се получат са <math>\sqrt{n}</math>. Това е така, защото един блок съдържа logn/2 елемента. Сега за всеки различен вектор може да се направи RMQ. Това ще стане със сложност <math>O(\log^2 n)</math> и като се има предвид, че различните вектори са <math>\sqrt{n}</math> на брой се получава сложност <math>O(\sqrt{n} \log^2 n)</math>. Показахме как може с достатъчно добра сложност да се направят всички предварителни изчисления, а след това за всеки блок да се отговаря с константна сложност. В крайна сметка се получава, че когато има запитване за някакъв интервал в масива L с направените предварително изчисления може да се отговори с константна сложност. Предварителните изчисления се правят с линейна сложност. Така чрез този подход може да се реализира LCA с линейна сложност, което пък води до това, че може да се реализира RMQ с линейна сложност на предварителните изчисления и константно време за отговаряне на заявките.

**Приложения**

**Lowest Common Ancestor (LCA)**

(може да го изкараме и като отделна статия, а тук да има само линк) може, може

**Дефиниция на проблема**

Проблема е известен с името Lowest Common Ancestor (LCA). Дадено е дърво T. За всеки два върха в дървото трябва да може да се намери върха, който е най-далече от корена и е част от пътищата от корена до двата върха.

**Свеждане към RMQ**

Задачата за LCA може да се сведе до решаване на задачата RMQ. Може върху дървото да се направи обхождане в дълбочина и при всяко посещение на връх в дървото този връх да се записва в края на един списък L. Така се получават <math>2\*n-1</math> записа в дървото. Прави се един масив H, като H[i] посочва дълбочината на върха L[i] в дървото. Масивът F посочва за даден връх в дървото къде е първото негово срещане в L. Ако има запитване за LCA на два върха i и j, може да се направи запитване за RMQ(F[i], F[j]) в масива H. Ако отговора е някаква позиция k в масива H, то L[k] ще е върха, който е отговора на LCA(i,j). Построяването на L става линейно по броя върхове в дървото. След това за изпълнението на RMQ могат да се използват различните подходи, които съществуват. За състезание е най-подходящо да се направи RMQ със сложност (<math>O(n \log n), O(1)</math>).

**Примерна Задача**

Ето една задача, която е била дадена на регионалното състезание по програмиране за студенти: <http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=2045>

[Категория](http://judge.openfmi.net:9080/mediawiki/index.php/%D0%A1%D0%BF%D0%B5%D1%86%D0%B8%D0%B0%D0%BB%D0%BD%D0%B8:Categories): [Алгоритми](http://judge.openfmi.net:9080/mediawiki/index.php/%D0%9A%D0%B0%D1%82%D0%B5%D0%B3%D0%BE%D1%80%D0%B8%D1%8F:%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B8)