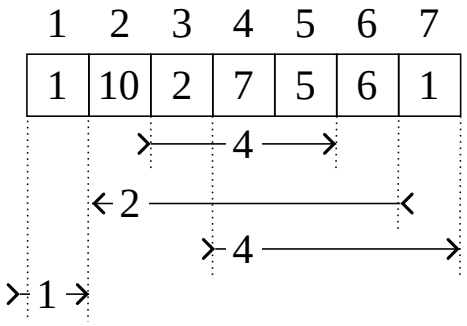
ЛЕТЕН ТУРНИР ПО ИНФОРМАТИКА
Пловдив, 6 – 8 юни 2025 г.
Група С, 7 – 8 клас

Задача С1. Deliverly – Анализ

Автор 1: Станислав Манилов

Автор 2: Иван Люцканов

Условието е базирано на задачата за отговаряне на заявки относно минималния/максималния елемент в различни интервали от даден масив – Range Min/Max Query (RMQ).



Фигура 1: Графично представяне на първия пример от задачата. Подредбата на началния и крайния индекс на всеки интервал има значение: ако има няколко къщи с еднаква височина, трябва да се изведе индекса на първата срещната от тях, в посоката на обходжане на къщите. Индексирането е от 1 и началният и крайният индекс на интервалите са включени.

Наивно Решение

Наивното решение на задачата включва или масивът да се обхожда за всяка заявка (сложност $\mathcal{O}(N \cdot Q)$), или предварително да се изчисли отговора за всяка възможна заявка, след което при получаване на заявка да се връща резултатът от таблицата за константна сложност. Вторият подход е със сложност $\mathcal{O}(N^2 + Q)$. Тъй като по условие N и Q са от сходен порядък, двете решения са еднакво добри.

Таблицата, която би се получила при реализация на второто решение, би изглеждала като в Таблица 1, за примера по-горе.

1	2	2	2	2	2	2
2	2	2	2	2	2	2
2	2	3	4	4	4	4
2	2	4	4	4	4	4
2	2	4	4	5	6	6
2	2	4	4	6	6	6
2	2	4	4	6	6	7

Таблица 1: Стойността в ред x и колона y е отговорът за заявка относно интервала x y. Индексирана от 1. Обърнете внимание, че може и да не е симетрична спрямо диагонала.

Класическо Ефикасно Решение

Класическият подход към оптимизиране на наивното решение е компресирането на таблицата, или с други думи – използването на *разредена таблица*. При този подход, вместо да преизчисляваме решението за интерали от каква да е дължина, го правим само за интервали с дължина, която да е степен на 2.

Постигаме го, чрез използване на динамично програмиране по следния начин:

1. Създаваме таблица, където индексът на реда определя началото на интервала, за който е записано решението в съответната клетка от таблицата;
2. индексът на колоната определя дължината на интервала, като тази дължина се удвоява със всяка следваща колона;
3. съответно, първата колона съдържа просто индексите от 1 до N включително;
4. за всяка клетка от другите колони е изпълнена рекурентната зависимост 1.

$$table[x,y] = \begin{cases} table[x,y-1] & \text{, ако } table[x,y-1] \geq table[x+2^{y-1},y-1] \\ table[x+2^{y-1},y-1] & \text{, ако } table[x,y-1] < table[x+2^{y-1},y-1] \end{cases} \quad (1)$$

Така получаваме *разредената таблица* за същия пример, представена в Таблица 2.

1	2	2	2
2	2	2	2
3	4	4	4
4	4	4	4
5	6	6	6
6	6	6	6
7	7	7	7

Таблица 2: Разредена таблица, попълнена на базата на входния масив и чрез прилагане на рекурентната зависимост 1. Забележете, че първата колона просто съдържа числата от 1 до N. Дадени са два примера как клетки са попълнени на базата на клетки от предходната колона. В последната колона, всяка клетка представлява отговор на заявка за интервал с дължина 8, която е по-голяма от дължината на входния масив. Подобна заявка няма да е валиден вход и на практика не би било нужно да попълваме четвъртата колона. Тук е показана, за по-голяма нагледност на рекурентната зависимост, но в реализацията може да я пропуснем.

С помощта на тази таблица, може да отговаряме на заявки за константно време по следния начин: ако дължината на интервала от заявката ($len = y - x + 1$) е точна степен на 2, тогава отпечатваме клетката с координати $[x][log_2(len)]$. Например, за заявката 4 7 (последната от примера), ще отпечатаме клетката с координати 4 и 2, т.е. числото 4.

Ако дължината на интервала от заявката не е точна степен на 2, тогава разглеждаме две различни клетки. Нека $k = floor(log_2(len))$, или с други думи максималната стойност, т. че $2^k < len$. Тогава, клетките които разглеждаме са с координати $[x][k]$ и $[y - 2^k + 1][k]$. (Втората е решението за интервала с дължина 2^k, завършващ в y.) Избираме тази от тях, която сочи към по-големия елемент във входния масив и я отпечатваме. Например, за заявка 3 5, $k = 1$. Клетките, които разглеждаме са $[3][1]$ и $[4][1]$. В същия пример и двете сочат към 4, така че отпечатваме 4.

Както видяхме, таблицата няма нужда да съдържа повече от $floor(log_2(N)) + 1$ колони, така че този алгоритъм изисква $\Theta(N.logN)$ памет и $\mathcal{O}(N.logN + Q)$ време. Това е така, понеже попълването на таблицата отнема константно време за всяка клетка, и всяка заявка отнема допълнително константно време за намиране и отпечатване на отговора.

Този подход е реализиран във файла deliverly_dp_68p.cpp в папката solutions от архива към задачата. Забележете, че при първата група от тестове няма гаранция, че x ще бъде по-малко от y, така че тази реализация включва и кода за наивното решение.

Пълно Решение

За пълно решение, алгоритъмът трябва да може да различава обхождане на елементите от масива в нарастващ и в намаляващ ред. Най-лесният начин това да се постигне е, чрез създаване на две таблици. При едната, равенствата между елементите се решават в полза на левия (този с по-малък индекс), а при другата – в полза на десния (този с по-голям индекс).

При изпълняването на заявки, това също трябва да се вземе предвид. Правилната таблица трябва да се ползва, а когато дължината на интервала не е степен на 2, логиката също да се различава по съответния начин.

Този подход е реализиран във файла deliverly_dp_100p.cpp в папката solutions от архива към задачата.

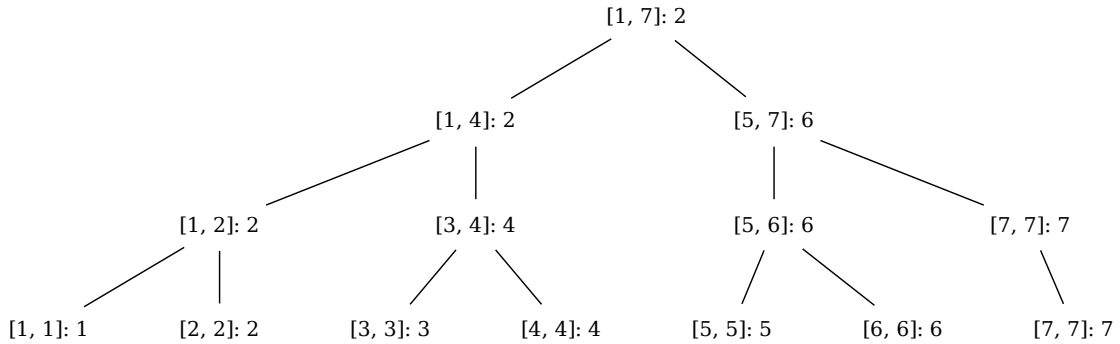
Алтернативно Ефикасно Решение

Алтернативен подход на този ползващ разредени таблици е подходът ползващ сегментно дърво[1]. При него резултатите са подредени в йерархична структура, така че в листата на дървото се пазят отговорите за интервали с дължина 1. Във всяко ниво нагоре по дървото, дължината на интервалите расте двойно и се изчислява чрез сравняване на интервала от следващото ниво. Дървото, което този подход строи за примера по-горе е изобразено във Фигура 2.

Дървото може да се построи за линейно време, започвайки от листата и работейки към корена. Ако N не е точна степен на 2, за реализацията трябва да се избере едно от няколко възможни решения. Най-лесно е да се копира последния елемент от масива, така че да се запълни до масив с размер, който е степен на 2. В предоставената реализация е предпочетено по-скоро да се внимава с достъпа до елементите на масива и са дублирани само последните върхове от нивата на дървото, които иначе биха имали нечетен брой върхове. Това решение няма значителен ефект върху ефективността на алгоритъма.

С помощта на дървото, на заявки може да се отговаря за логаритмично време. Алгоритъмът за отговаряне на заявка е по-сложен от този при подхода с динамичното програмиране и изглежда така:

[1] За подробно описание на структурата и друг анализ на това, как решава задачата RMQ, вижте https://www.informatika.bg/lectures/segment-trees.

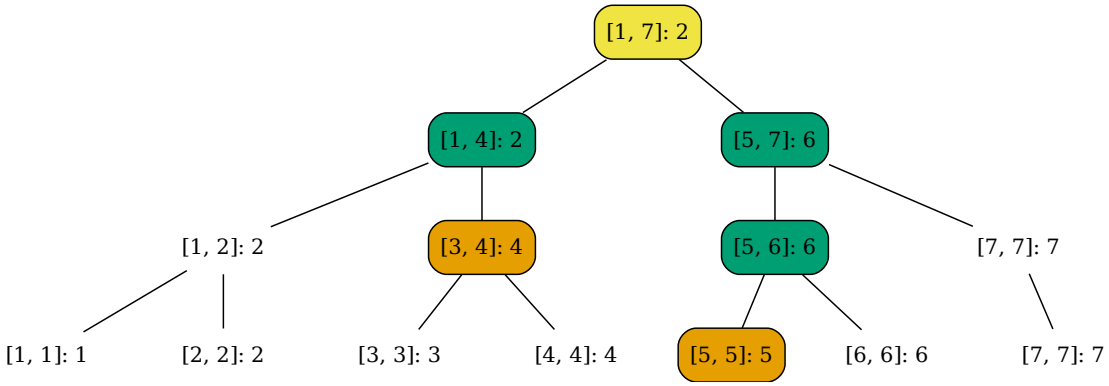


Фигура 2: Сегментно дърво, съответстващо на примера по-горе. Във всеки връх от дървото се пази отговора за дадена заявка с дължина, която е степен на 2. Забележете, че за разлика от разредената таблица, дървото съдържа значително по-малко данни. За сметка на това, отговарянето на заявки отнема логаритмично време, вместо константно.

1. Използваме рекурсивна помощна функция, която разглежда конкретен връх от дървото, започвайки от корена;
2. ако интервалът от заявката изцяло съдържа (или съвпада с) интервала, представен от текущия връх, тогава рекурсивната функция връща резултата запазен в този връх;
3. ако сечението на интервала от левия наследник на върха и интервала от заявката е празното множество, тогава рекурсивно се спускаме вдясно; симетрично и за другия случай;
4. остава случаят, когато и двата наследника имат припокриване с интервала от заявката; рекурсивно решаваме и за двата и накрая сравняваме отговорите по същия начин като от преди (всеки от тях ще бъде индекс във входния масив; избираме този, който сочи към по-голям елемент).

Да разгледаме заявката 3 5 и примерното дърво. Стъпките, които алгоритъмът изпълнява са следните (изобразено във Фигура 3):

- Коренът е [1, 7]; и двата му наследника имат припокриване с интервала от заявката [3, 5], така че продължаваме рекурсивно и за двата;
  - разглеждайки [5, 7], само левият му наследник има припокриване с [3, 5], така че продължаваме само в него;
  - разглеждайки [5, 6], само левият му наследник има припокриване с [3, 5], така че продължаваме само в него;
  - разглеждайки [5, 5], той изцяло се съдържа в интервала от заявката [3, 5], така че връщаме резултата 5 нагоре;
- да разгледаме и левият наследник на [1, 7];
  - разглеждайки [1, 4], само десният му наследник има припокриване с [3, 5], така че продължаваме само в него;
  - разглеждайки [3, 4], той изцяло се съдържа в интервала от заявката [3, 5], така че връщаме резултата 4 нагоре;
- в корена, сравняваме елементите с индекси 4 и 5 и връщаме индекс 4 като резултат от заявката.

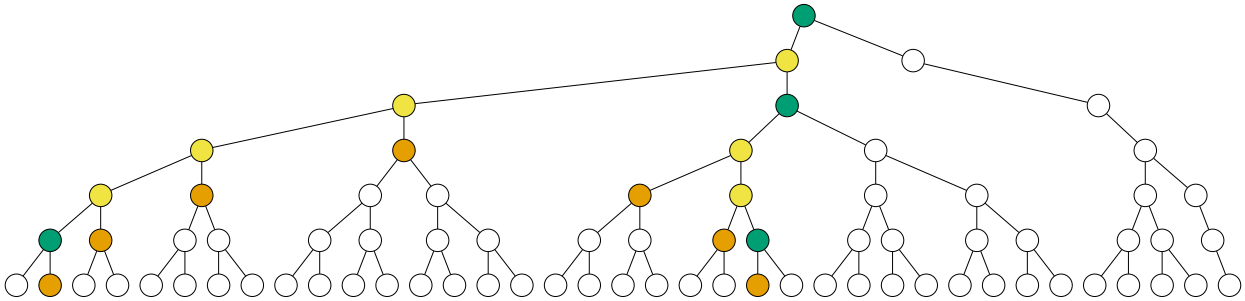


Фигура 3: Обхождане на сегментното дърво за заявката 3 5. Жълтите върхове на дървото имат два наследника, които имат припокриване със заявката. Зелените върхове имат само по един. Оранжевите се съдържат изцяло в заявката. Въпреки, че дървото съдържа отговори за върхове от оцветените по-близки до корена (включително и самия корен), отговора на заявката е сравнението само между елементите, към които сочат индексите от върховете оцветени в оранжево (листата на обхождането).

Виждаме, че това е същият отговор като преди. В общия случай, въпреки, че изглежда, че може би ще трябва да обходим цялото дърво, всъщност едно от разклоненията на рекурсията винаги (освен при първото разклонение) завършва след една стъпка. Това е така, понеже единият наследник ще бъде с интервал, който изцяло се съдържа в интервала от заявката.

Например, ако $N = 37$ и заявката е 2 23, интервалите, които ще бъдат разгледани са следните (изобразено във Фигура 4):

- $[2, 23]$ в $[1, 37]$: десният отпада
- $[2, 23]$ в $[1, 32]$: първо разклонение
- $[2, 23]$ в $[1, 16]$: десният е с размер степен на 2
- $[2, 23]$ в $[17, 32]$: десният отпада
- $[2, 23]$ в $[1, 8]$: десният е с размер степен на 2
- $[2, 23]$ в $[9, 16]$: изцяло се съдържа
- $[2, 23]$ в $[1, 4]$: десният е с размер степен на 2
- $[2, 23]$ в $[5, 8]$: изцяло се съдържа
- $[2, 23]$ в $[1, 2]$: левият отпада
- $[2, 23]$ в $[3, 4]$: изцяло се съдържа
- $[2, 23]$ в $[2, 2]$: изцяло се съдържа
- $[2, 23]$ в $[17, 32]$: десният отпада
- $[2, 23]$ в $[17, 24]$: левият е с размер степен на 2
- $[2, 23]$ в $[17, 20]$: изцяло се съдържа
- $[2, 23]$ в $[21, 24]$: левият е с размер степен на 2
- $[2, 23]$ в $[21, 22]$: изцяло се съдържа
- $[2, 23]$ в $[23, 24]$: десният отпада
- $[2, 23]$ в $[23, 23]$: изцяло се съдържа



Фигура 4: Обхождане на сегментно дърво за N=37 и заявката 2 23. Жълтите върхове на дървото имат два наследника, които имат припокриване със заявката. Зелените върхове имат само по един. Оранжевите се съдържат изцяло в заявката. Оранжевите върхове връщат стойността, запазена в тях. Зелените върхове предават нагоре стойността от единствения си оцветен наследник. Жълтите върхове избират по-добрата стойност от двата си наследника. Отговора на заявката е стойността върната от корена.

С други думи, може да се докаже, че сложността на отговаряне на заявка е $\mathcal{O}(logN)$. Така, сложността на това решение е $\mathcal{O}(N)$ (за предварителното изчисление) $+\mathcal{O}(Q \cdot logN)$ (за отговаряне на заявките). С други думи, сложността на решението е $\mathcal{O}(N + Q \cdot logN)$. Паметта, която заема е $\Theta(N)$.

Това решение е реализирано във файла deliverly_68p.cpp.

За да решим задачата за 100 точки, трябва да приложим същия подход като при решението използващо динамично програмиране: да подготвим две дървета, едното от които предпочита левите си наследници, а другото десните. При отговаряне на заявки трябва също да се вземе предвид дали заявката е обърната. Ако е, трябва освен да се използва второто дърво, ”жълтите”върхове също да предпочитат десните си наследници.

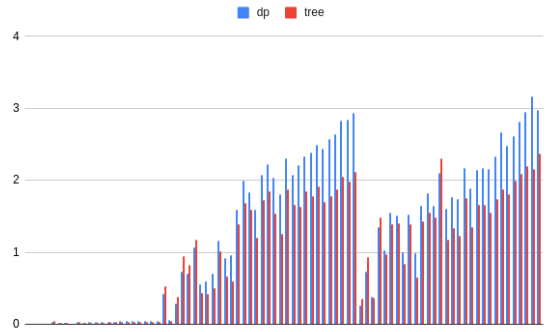
Това решение е реализирано във файла deliverly_100p.cpp. В author.cpp е същото решение, но с добавени подробни коментари с цел яснота.

Сравнение на двете ефикасни решения

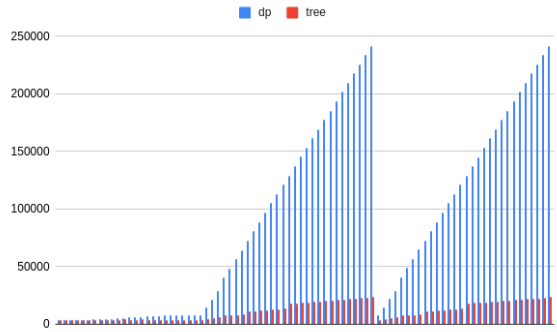
Така получихме две решения, които носят 100 точки. Едното е със сложност $\mathcal{O}(N \cdot logN + Q)$ и памет $\Theta(N \cdot logN)$, а другото със сложност $\mathcal{O}(N + Q \cdot logN)$ и памет $\Theta(N)$. не беше упоменато в условието, но за големите тестове Q винаги е по-голямо от N. (Генерирано е като произволно число между N и $max_N = 10^6$.) На пръв поглед бихме очаквали първото решение да е по-бързо от второто. Истината е, че сравнявайки времената им на входните тестове получаваме графиката от Фигура 5: второто решение е средно с 15% по-бързо.

Възможно обяснение за това е, че Q все пак е от порядъка на N ($Q = \Theta(N)$) в общия случай. Асимптотичната сложност на двата алгоритъма става еднаква, а именно $\mathcal{O}(N \cdot logN)$. И все пак, докато ДП решението винаги строи цялата $N \cdot logN$ таблица (с други думи, сложността е по-точно $\Theta(N \cdot logN)$), решението с дърво има вероятност да отговори за малък брой стъпки за определни заявки, т.е. времето там зависи от самите стойности на масива и е $\mathcal{O}(N \cdot logN)$. Друго възможно обяснение е, че използвайки по-малко памет, тя може да се събере в процесорния кеш и така достъпа до нея да е по-бърз.

бел. отг.: Имплементациите на други четири решения, базирани на двете идеи, описани от авторите на задачата, може да намерите във файловете deliverly_Lupov_rmq_100p.cpp и deliverly_Iliyan_rmq_100p.cpp (с разредена таблица) и deliverly_Lupov_tree_100p.cpp и deliverly_Iliyan_tree_100p.cpp (със сегментно дърво). Решенията със “sparse table” са с около 40% по-бързи от тези със “segment tree”. Това потвърждава асимптотичните разсъждения, но както е описано в анализа, понякога се наблюдават аномалии, свързани с конкретната имплементация.



*Фигура 5: Сравнение на времето за изпълнение за двете ефикасни решения върху входните примери. Подходът използващ динамично програмиране е в синьо, а този използващ сегментни дървета е в червено. Второто решение е **15% по-бързо** (сравнено тест по тест и претеглено средно геометрично).*



Фигура 6: Сравнение на паметта, нужна за изпълнението на двете ефикасни решения. Имайки предвид асимптотичния анализ, не е учудващо, че първото изисква много повече памет. Дървото изисква линейна памет спрямо входния масив, докато таблицата за реализация на динамично програмиране е с размер $\mathcal{O}(N \cdot logN)$.