Ezoteryczne Kartki
Dziel i zwyciężaj

Jakub Bachurski

wersja 1.0.0

1 Wstęp

Dziel i zwyciężaj (*Divide & Conquer*) to rodzina algorytmów, z natury rekurencyjnych, składających się z dwóch faz:

1. **Dziel**. Dzieli rozwiązywany problem na mniejsze podproblemy, które są rozwiązywane rekurencyjnie tym samym algorytmem.
2. **Zwyciężaj**. Łączy rozwiązania podproblemów, otrzymując rozwiązanie całego problemu.

2 Przykład zadania

Aby lepiej zrozumieć tok myślowy dziel i zwyciężaj, rozwiążmy najpierw proste zadanie: [Codeforces] Imbalanced Array.

Treść Dla ciągu liczb a definiujemy jego *niezrównoważenie* jako $\max(a) - \min(a)$. Policz sumę po *niezrównoważeniach* wszystkich spójnych podciągów a. Formalniej, policz sumę:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \max(a_i, a_{i+1}, ..., a_j) - \min(a_i, a_{i+1}, ..., a_j)$$

Rozwiązanie Zauważmy, że gdybyśmy znali algorytm znajdujący sumę po maksimum wszystkich podtablic a (oznaczmy ten algorytm $\mathcal{S}(a)$), to kwestia znalezienia sumy po minimum to wykonanie $\mathcal{S}$ na ciągu $a' = (-a_1, -a_2, ..., -a_n)$ i zanegowanie wyniku. Uprościliśmy zatem problem do liczenia:

$$\mathcal{S}(a) = \sum_{i=1}^{n} \sum_{j=i}^{n} \max(a_i, a_{i+1}, ..., a_j)$$

Policzenie tego brutalnie ma złożoność $O(n^2)$, więc wykorzystajmy dziel i zwyciężaj, by polepszyć tę złożoność. Pozostaje przejść do wymyślenia fazy dziel i ułożenia odpowiedniej fazy zwyciężaj.

Wprowadźmy oznaczenie podtablicy: $a[i..j] = (a_i, a_{i+1}, ..., a_j)$.

Dziel Skorzystajmy z najprostszego pomysłu: podzielmy a na dwie połowy i nazwijmy je a_L i a_R. Wyznaczmy punkt podziału $m = \lfloor \frac{n}{2} \rfloor$ i niech $a_L = a[1, m]$ i $a_R = a[m + 1, n]$.

Wszystkie podtablice $a[i..j]$ są teraz jednego z trzech rodzajów:

1. $a[i..j]$ znajduje się w całości w a_L: $j \leqslant m$.
2. $a[i..j]$ znajduje się w całości w a_R: $i > m$.
3. $a[i..j]$ znajduje się w obu podtablicach: $i \leqslant m < j$.

Zrobiliśmy założenie, że znamy już algorytm $\mathcal{S}$ rozwiązujący nasz problem. W takim razie, możemy wywołać go rekurencyjnie otrzymując $\mathcal{S}(a_L)$ oraz $\mathcal{S}(a_R)$, w ten sposób załatwiając podtablice pierwszego i drugiego typu. Pozostaje nam tylko policzyć sumę po maksimum podtablic typu trzeciego.

Zwyciężaj Zacznijmy od dodania do wyniku $\mathcal{S}(a_L) + \mathcal{S}(a_R)$. Przyjrzyjmy się teraz przykładowemu ciągowi a:

a_L					a_R				
5	3	1	2	2	0	3	2	6	4

Spójrzmy, jak wyglądają liczone maksima dla podtablic typu 3 (chcemy znaleźć ich sumę). Komórka w i-tym rzędzie i j-tej kolumnie to maksimum $a[m - i + 1..j]$ (liczby w nawiasach zostały już wcześniej policzone w $\mathcal{S}(a_L)$).

5	3	1	2	2	0	3	2	6	4
				(2)	2	3	3	6	6
			(2)	(2)	2	3	3	6	6
		(1)	(2)	(2)	2	3	3	6	6
	(3)	(3)	(3)	(3)	3	3	3	6	6
(5)	(5)	(5)	(5)	(5)	5	5	5	6	6

Czas na obserwację: są to maksima prefiksowe a_R, których prefiks został zmodyfikowany przez maksimum kolejnych sufiksów a_L. W tym wypadku maksima prefiksowe a_R wynosiły $(0, 3, 3, 6, 6)$, a maksima sufiksów $(2, 2, 2, 3, 5)$. Ponieważ oba te ciągi są monotoniczne, prefiks ten może się tylko zwiększać i możemy wykorzystać wskaźnik przesuwany tylko w prawą stronę. Da nam to amortyzowaną złożoność $O(n)$. Cała część fazy zwyciężaj może być wykonana w $O(n)$.

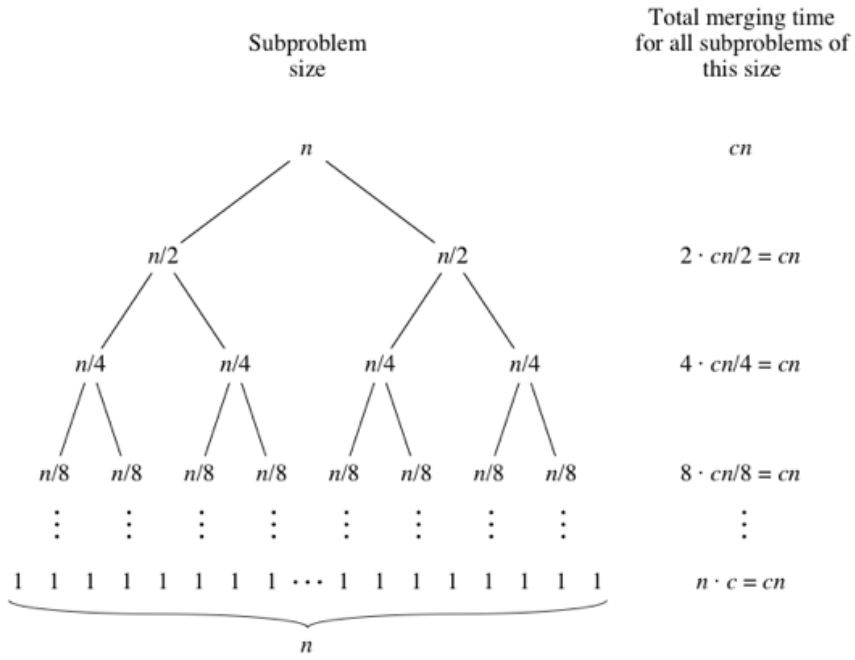
Przypadek bazowy Pozostaje tylko rozważyć przypadek bazowy rekurencji. Możemy za niego wybrać $n = 1$: rozwiązaniem jest wtedy jedyny element tablicy, czyli a_1. Rozważając więcej przypadków bazowych w prostszy sposób możemy polepszyć czynnik stały w złożoności rozwiązania.

3 Obliczanie złożoności

Istnieje kilka metod szacowania złożoności algorytmów dziel i zwyciężaj.

3.1 Drzewo rekurencji

Sposób ten jest najprostszy i opiera się na intuicji wizualnej reprezentacji wywołań rekurencyjnych. Po prostu rysujemy drzewo ilości danych wejściowych. Narysujmy takie drzewo dla przykładowego zadania[1].



Rysunek 1: Drzewo rekurencji

Na najwyższym poziomie rekurencji jest wywołanie dla a – składa się z n danych wejściowych. Kolejny poziom to wywołania dla a_L i a_R, odpowiednio $\frac{n}{2}$ danych. Możemy zauważyć, że na każdym poziomie rekurencji suma liczby danych to n (bo dane nie przepadają, a wszystkie połowy nie nachodziły na siebie). Natomiast samych poziomów jest $\log_2 n$, bo na każdym poziomie wielkość jednego wywołania maleje dwukrotnie. Wnioskujemy z tego, że złożoność to $O(n \log n)$.

[1] Grafika pochodzi z `https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort`.

3.2 Rozwiązywanie rekurencji

Bardziej analitycznym sposobem policzenia złożoności jest rozwiązanie funkcji rekurencyjnej złożoności czasowej od ilości danych wejściowych $T(n)$. W tym wypadku stwierdzamy, że $T(1) = O(1)$, czyli wywołanie dla jednej jednostki danych zajmuje stałą liczbę operacji. Natomiast w przypadku ogólnym:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Ponieważ w fazie dziel wykonujemy dwa wywołania na dwukrotnie mniejszej liczbie danych oraz w $O(n)$ wykonujemy fazę zwyciężaj.

Aby rozwiązać tę rekurencję (znaleźć wzór na $T(n)$ bez rekurencji) należy ją rozwijać, dopóki nie zauważymy schematu. Na koniec możemy udowodnić ją indukcyjnie.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + O(n) = \\
&= 2(2T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)) + O(n) = \\
&= 4T\left(\frac{n}{4}\right) + 2O(n) = \\
&= 4(2T\left(\frac{n}{8}\right) + O\left(\frac{n}{4}\right)) + 2O(n) = \\
&= 8T\left(\frac{n}{8}\right) + 3O(n) = \\
&\quad \ldots \\
&= 2^k T(n2^{-k}) + kO(n) = \\
&\overset{k=\log_2 n}{=} \log_2 O(n) = O(n \log n)
\end{aligned}$$

3.3 Twierdzenie o rekurencji uniwersalnej

Twierdzenie o rekurencji uniwersalnej *(Master Theorem)* to prosty sposób na oszacowanie rekurencji postaci:

$$T(n) = aT(n/b) + O(n^d)$$

Są trzy przypadki.

$$T(n) = \begin{cases} O(n^d), & d > log_b a \\ O(n^d \log n), & d = log_b a \\ O(n^{log_b a}), & d < log_b a \end{cases}$$

Jednak zwykle pozostałe dwa sposoby w zupełności nam wystarczą.

4 Algorytmy szybkie

Algorytmy *quicksort* i *quickselect* korzystają z dziel i zwyciężaj, by sortować ciąg oraz wybierać k-ty co do wielkości element. Pierwszy z nich jest jednym z najpopularniejszych algorytmów sortowania, dzięki swojemu dobremu średniemu czasowi działania. Oba algorytmy zostały opracowane przez Tony'ego Hoare'a.

4.1 Quicksort

Pomysł stojący za quicksortem jest bardzo prosty: w fazie dziel zamiast dzielić ciąg na dwie połowy, dzielimy go względem arbitralnie wybranego elementu p nazywanego *pivot*. Dokładniej, ciąg a podzielimy na ciąg $a_<$ zawierający elementy $a_i < p$, oraz $a_>$, do którego trafią $a_i > p$. Elementy równe p trafią do ciągu $a_=$. Następnie, w fazie zwyciężaj posortujemy quicksortem $a_<$ i $a_>$ ($a_=$ jest posortowane, bo zawiera tylko jeden rodzaj elementów). Na koniec wystarczy zwrócić konkatenację $a_<$, $a_=$, $a_>$.

Warto się zastanowić, jak efektywnie wybrać pivot p. Najlepszy czas działania osiągniemy jeżeli wybierzemy medianę (element który stoi w środku po posortowaniu całego ciągu), bo wtedy $a_<$ i $a_>$ są podobnej wielkości. W takim wypadku osiągniemy złożoność $O(n \log n)$. Niestety, znalezienie mediany jest stosunkowo kosztowne. Dlatego zwykle wybiera się pierwszy lub ostatni element a – jednak dokonując takiego wyboru istnieje łatwy do skonstruowania przypadek pesymistyczny (jaki?), w którym algorytm ma złożoność kwadratową. Zauważmy jednak, że wybierając pivot losowo *prawdopodobnie* będzie on stosunkowo blisko mediany – jeżeli sortowany ciąg byłby losowo przetasowany przed sortowaniem, to wybierając pierwszy lub ostatni element na pivot tak na prawdę go losujemy, więc osiągniemy średnią złożoność czasową $O(n \log n)$.

4.2 Quickselect

Quickselect rozwiązuje problem znalezienia w ciągu k-tego co do wielkości elementu. Ponownie wykorzystamy pomysł z pivotem p i podzielimy ciąg na $a_<$ i $a_\geqslant$. Nietrudno się domyślić, że jeżeli $k \leqslant |a_<|$, to odpowiedzią jest jeden z elementów ciągu $|a_<|$ – w takim razie wywołujemy się rekurencyjnie na $a_<$. W przeciwnym wypadku, odpowiedź musi znajdować się w $a_\geqslant$ – wtedy wywołujemy się quickselectem rekurencyjnie z parametrem $k - |a_<|$ (bo w tym nie znajduje się $|a_<|$ elementów mniejszych, w odróżnieniu od oryginalnego ciągu).

Rozważania na temat wyboru pivotu przebiegają bardzo podobnie również w wypadku tego algorytmu. Jeżeli pivot jest wybierany losowo, to algorytm ma średnią złożoność $n + \frac{n}{2} + \frac{n}{4} + ... + 1 = O(n)$. W przypadku pesymistycznym ma on jednak złożoność $O(n^2)$.

Pozostaje jeszcze kwestia tego, że quickselect nie do końca jest algorytmem dziel i zwyciężaj: w fazie zwyciężania nie łączymy wyników rekurencyjnych wywołań, a jedynie wybieramy jeden z podproblemów i tylko w nim wywołujemy się rekurencyjnie. Stąd określenie zmniejsz i zwyciężaj (*Decrease & Conquer*). Podobnie klasyfikowane jest wyszukiwanie binarne.

5 Gauss i mnożenie

Matematyk Carl Friedrich Gauss zauważył, że iloczyn dwóch liczb zespolonych[2]

$$(a+bi)(c+di) = (ac-bd) + (bc+ad)i$$

można otrzymać stosując zaledwie *trzy* mnożenia liczb rzeczywistych. Wystarczy policzyć ac, bd oraz $(a+b)(c+d)$, ponieważ:

$$bc + ad = (a+b)(c+d) - ac - bd$$

Wydaje się to niezbyt przydatne z naszej perspektywy. Możemy jednak wykorzystać to znalezisko w kompletnie inny sposób: rekurencyjnie, mnożąc wielomiany bądź liczby dowolnej precyzji.

5.1 Algorytm Karatsuby

Weźmy dwie liczby n-cyfrowe o podstawie β: A oraz B. Chcąc je normalnie pomnożyć, musimy wykonać $O(n^2)$ mnożeń cyfr. Spróbujmy zoptymalizować ten algorytm w stylu dziel i zwyciężaj inspirując się znaleziskiem Gaussa. W tym celu zdefiniujmy $m = \frac{n}{2}$ i podzielmy liczby A, B na połowy:

$$A = \beta^m A_\uparrow + A_\downarrow$$
$$B = \beta^m B_\uparrow + B_\downarrow$$

Dokonajmy tego podziału tak, że $A_\uparrow, A_\downarrow, B_\uparrow, B_\downarrow < \beta^m$.
Spójrzmy, jak teraz wygląda iloczyn tych liczb:

$$AB = (\beta^m A_\uparrow + A_\downarrow)(\beta^m B_\uparrow + B_\downarrow) = \beta^{2m} A_\uparrow B_\uparrow + \beta^m (A_\uparrow B_\downarrow + A_\downarrow B_\uparrow) + A_\downarrow B_\downarrow$$

Nietrudno sprawdzić, że rekurencyjnie wykonując te cztery iloczyny (liczb m-cyfrowych, które wykonujemy, by otrzymać iloczyn liczb $2m$-cyfrowych) otrzymamy $T(n) = 4T(\frac{n}{2}) + O(n)$, co daje $T(n) = O(n^2)$. Korzystając ze sposobu mnożenia Gaussa, spróbujmy to zrobić w trzech mnożeniach:

$$X = A_\uparrow B_\uparrow$$
$$Z = A_\downarrow B_\downarrow$$
$$Y = (A_\uparrow + A_\downarrow)(B_\uparrow + B_\downarrow) - X - Z$$

Jak się można łatwo przekonać, $Y = A_\uparrow B_\downarrow + A_\downarrow B_\uparrow$. W ten sposób otrzymaliśmy wszystkie składniki AB w trzech, a nie czterech, mnożeniach liczb m-cyfrowych:

$$AB = \beta^{2m} X + \beta^m Y + Z$$

[2]Czyli liczb postaci $a + bi$, gdzie i to taka liczba, że $i^2 = -1$.

Ponieważ przemnożenie przez podstawę systemu liczbowego jest operacją o wiele prostszą (to kwestia dodania odpowiedniej liczby zer), znaleźliśmy lepszy algorytm mnożenia od brutalnego.

Przeanalizujmy teraz złożoność, wykorzystując twierdzenie o rekurencji uniwersalnej. Wykonaliśmy trzy mnożenia liczb dwukrotnie mniejszych i kilka sumowań, czyli:

$$T(n) = 3T(n/2) + O(n)$$

$d = 1 < log_2 3 \approx 1.585$, więc na mocy twierdzenia o rekurencji uniwersalnej:

$$T(n) = O(n^{log_2 3}) \approx O(n^{1.585})$$

Opracowaliśmy w ten sposób algorytm Karatsuby. Można go także wykorzystywać do mnożenia wielomianów. Cieszy się on popularnością, ponieważ jego dobra implementacja ma stosunkowo małą stałą.

5.2 Algorytm Strassena

Dociekliwy czytelnik mógłby się zacząć zastanawiać, czy w podobny sposób można opracować szybszy algorytm mnożenia macierzy. Owszem, da się, ale jego wymyślenie jest o wiele trudniejsze. Dokonał tego Volker Strassen. Był to pierwszy algorytm wykonujący mnożenie macierzy szybciej niż w $O(n^3)$.

Więcej można przeczytać o nim np. na Wikipedii: `https://en.wikipedia.org/wiki/Strassen_algorithm`. W dużym skrócie, algorytm pozwala na mnożenie macierzy kwadratowych wielkości $2^n \times 2^n$, dzieląc każdą z nich na cztery macierze $2^{n-1} \times 2^{n-1}$. Następnie wykonuje 7 mnożeń (a nie 8, jak zrobiłoby to podejście brutalne), a następnie oblicza macierz wynikową w sumarycznie 8 dodawań i odejmowań. Można sprawdzić, że jego złożoność to

$$O(n^{log_2 7}) \approx O(n^{2.81})$$

Niestety, ze względu na bardzo duży czynnik stały nie ma zbyt dużego znaczenia praktycznego. Natomiast zapoczątkował on proces polepszania algorytmów mnożenia macierzy: obecnie, najszybszy algorytm ma złożoność $O(n^{2.373})$.

6 Meet in the middle

Meet in the middle jest ciekawą techniką służącą do optymalizacji algorytmów wykładniczych. W takich algorytmach często przeglądamy zbiory wszystkich możliwych rozwiązań poszukując rozwiązania pewnego problemu decyzyjnego (np. czy podzbiór sumuje się do pewnej liczby, czy dana permutacja ma pewną właściwość). Zamiast przeglądać wszystkie możliwe rozwiązania jednocześnie, możemy spróbować śledzić jedynie ich część, na przykład dotyczącą tylko jednej połowy. W odróżnieniu od klasycznych algorytmów dziel i zwyciężaj, w fazie dziel nie wywołujemy algorytmu rekurencyjnie.

6.1 Problem sumy podzbioru: [IX OI] Szyfr

Treść Dany jest (multi)zbiór liczb $w = \{w_1, w_2, ..., w_n\}$ ($1 \leqslant w_i \leqslant 10^9$, $1 \leqslant n \leqslant 40$). Czy istnieje ich podzbiór sumujący się do podanej liczby W? Jeżeli odpowiedź jest twierdząca, podaj ten podzbiór.

Rozwiązanie Problem ten jest NP-zupełny, więc nie możemy liczyć na rozwiązanie w czasie wielomianowym (można go rozwiązać podobnie jak problem plecakowy w czasie pseudowielomianowym). Zastanówmy się najpierw nad podejściem brutalnym i spróbujmy wykorzystać *meet in the middle*.

Najprostszym sposobem jest przejrzenie wszystkich podzbiorów w $O(2^n)$ i sprawdzenie, czy któryś z nich spełnia warunki zadania. Podzielmy zatem nasz zbiór na połowy W_0 i W_1 i policzmy sumy wszystkich ich podzbiorów S_0 i S_1. Zauważmy, że:

$$S = \{s + z : s \in S_0 \wedge z \in S_1\}$$

Sprowadziliśmy zatem problem do stwierdzenia, czy istnieje taka para elementów $s \in S_0$ i $z \in S_1$, że $s + z = W$ lub, równoważnie, $z = W - s$.

Rozwiązanie tak postawionego problemu jest możliwe w czasie wielomianowym (względem wielkości zbiorów S_0 i S_1). Wystarczy spamiętać w odpowiedniej strukturze reprezentującej zbiór wszystkie wartości S_1 i przeiterować się po elementach $s \in S_0$ sprawdzając, czy w S_1 znajduje się element $W - s$. Wykorzystując hashmapę można to wykonać w średnim czasie $O(|S_0| + |S_1|)$ (lub $O((|S_0| + |S_1|) \log |S_1|)$ sortując i wyszukując binarnie). Identyfikację wykorzystanych elementów można przeprowadzić przypisując każdej liczbie jej indeks z wejścia. Tym samym kończymy fazę zwyciężania.

Podsumujmy złożoność algorytmu. Wielkość $|S_0|$ i $|S_1|$ jest rzędu $2^{n/2}$, zatem sumarycznie otrzymaliśmy złożoność $O(2^{n/2})$, co mieści się w limitach zadania.

6.2 [ILOCAMP] (Treść) Polonez

Treść Mamy dany ciąg k permutacji („figur") $\pi = (\pi_1, \pi_2, ..., \pi_k)$ n liczb (tancerzy). Dla każdej permutacji τ ciągu figur, rozważamy złożenie tych permutacji w kolejności τ – w ten sposób powstaje permutacja p. Punktem stałym permutacji p jest taki indeks, dla którego $p(i) = i$. Chcemy policzyć, jaka jest suma po liczbie punktów stałych p dla każdej permutacji τ. $1 \leqslant n \leqslant 2000$, $1 \leqslant k \leqslant 11$.

Przykładowo, dla $\tau = ((1, 4, 2, 5, 3, 6), (2, 3, 1, 5, 4, 6), (3, 1, 5, 4, 2, 6))$ są dwa punkty stałe (1 i 6) złożenia kolejnych permutacji:

Permutacja	Ciąg po permutacji
–	1, 2, 3, 4, 5, 6
1, 4, 2, 5, 3, 6	1, 4, 2, 5, 3, 6
2, 3, 1, 5, 4, 6	2, 1, 4, 3, 5, 6
3, 1, 5, 4, 2, 6	**1**, 5, 2, 3, 4, **6**

Rozwiązanie Brutalne rozwiązanie działa w $O(nk!)$ i przegląda wszystkie permutacje figur i najzwyklej zlicza punkty stałe. Wykorzystajmy podejście meet in the middle i podzielmy figury na połowy, a następnie policzmy wszystkie permutacje powstałych połówek. Oznaczmy połowy L i R, a ich zbiory ich permutacji P_L i P_R. Spróbujmy parować powstałe permutacje z P_L i P_R ($\circ$ oznacza złożenie permutacji):

$$Q = \{\pi_L \circ \pi_R : \pi_L \in P_L \wedge \pi_R \in P_R\}$$

Jak nietrudno się domyślić, zbiór Q nie zawiera wszystkich możliwych permutacji. Ma on jedynie $(\frac{k}{2})!^2$ elementów, a wszystkich permutacji jest $k!$. Zastanówmy się, dlaczego nie rozważamy wszystkich permutacji: permutacje ze zbioru R nie mają szans znaleźć się na początku naszego złożenia w zbiorze Q. Musimy zatem z π wybrać wszystkie możliwe podzbiory figur L o $\frac{k}{2}$ elementach. Pozostałe figury trafiają do R. Dopiero w ten sposób rozważymy wszystkie permutacje figur. Możemy teraz przejść do kroku zwyciężania.

Zastanawiamy się teraz, ile jest punktów stałych wśród wszystkich permutacji $\pi_R \circ \pi_L$ ($\pi_L \in P_L, \pi_R \in P_R$). Zapiszmy to inaczej i skorzystajmy z tego, że każda permutacja ma swoją odwrotność:

$$\pi_R(\pi_L(i)) = i$$
$$\pi_L(i) = \pi_R^{-1}(i)$$

Rozbijając w ten sposób permutacje na dwie strony równości ułatwiliśmy sobie zadanie. Możemy teraz wybrać pewną permutację π_L oraz parametr i i po prostu policzyć $x = \pi_L(i)$. Teraz wystarczy po prostu spytać „ile jest takich permutacji π_R, że $\pi_R^{-1}(i) = x$?". Znacząco uprościliśmy problem.

Ustalmy parametr i – figurę, dla której chcemy zliczyć, ile ma punktów stałych. Przejdźmy teraz po wszystkich π_R^{-1} i dla każdego x zliczmy, ile razy $x = \pi_R^{-1}(i)$. Oznaczmy tę wartość jako $l(x)$. Teraz wystarczy dla każdego π_L dodać do wyniku wartość $l(\pi_L(i))$ – liczbę takich permutacji π_R, które możemy sparować z rozważanym π_L, aby $\pi_R(\pi_L(i)) = i$. Implementując ten krok w odpowiedni sposób, otrzymamy złożoność $O(n(\frac{k}{2})!)$.

Jeżeli rozważymy wszystkie możliwe podzbiory L i R, w ten sposób weźmiemy pod uwagę wszystkie punkty stałe.

Podsumujmy złożoność tego podejścia. Podzbiorów $\frac{k}{2}$-elementowych jest $\binom{k}{\frac{k}{2}}$. Dla każdego podzbioru figur tej wielkości rozważamy wszystkie permutacje – jest ich $(\frac{k}{2})!$ – i robimy to sumarycznie w $O(n(\frac{k}{2})!)$. To daje nam złożoność

$$O\left(\binom{k}{\frac{k}{2}}\left(\frac{k}{2}\right)! \cdot n\right) = \left(\frac{k!}{(\frac{k}{2})!} \cdot n\right)$$

Jest to złożoność o wiele lepsza od brutalnego $O(nk!)$.

7 Aplikacja aktualizacji

Rozwiązując zadanie z aktualizacjami i zapytaniami można w ciekawy sposób wykorzystać dziel i zwyciężaj. Skorzystamy z tego, że czasem wiele aktualizacji nie ma wzajemnego wpływu na swoje działanie. Dzięki temu sprowadzimy problem do najpierw rozwiązania zestawu aktualizacji, a następnie odpowiadania na zapytania. Rozważmy to na przykładzie zadania „Przekaźniki telekomunikacyjne".

7.1 [XXV OI] Przekaźniki telekomunikacyjne

Treść Dany jest ciąg w długości n (opisujący siłę sygnału w każdym punkcie, $2 \leqslant n \leqslant 300\ 000$), początkowo wypełniony zerami, oraz m operacji ($1 \leqslant m \leqslant 500\ 000$). Każda operacja jest jednego z trzech rodzajów:

- `P` – postaw w punkcie p przekaźnik o sile s i współczynniku a. Dla każdego indeksu x w ciągu w dodaj $\max(0, s - a \cdot |x - p|)$.
- `U` – usuń z punktu p przekaźnik (w tym samym punkcie nie będą stały dwa przekaźniki), tym samym odejmując dodaną przez niego siłę sygnału.
- `Z` – podaj średnią siłę sygnału w_x dla każdego indeksu x spełniającego $x_1 \leqslant x \leqslant x_2$. Średnia ta powinna zostać zaokrąglona w dół.

Rozwiązanie Po pierwsze, zapytanie można sprowadzić do zwykłych zapytań o sumę (sumę należy następnie podzielić przez długość przedziału). Natomiast aktualizacje można sprowadzić do dodawania na przedziałach w ciągów arytmetycznych. Jest to jedynie kwestia uważnej implementacji. Dlatego też większość zawodników zaimplementowała w tym zadaniu drzewo przedziałowe, które może być w odpowiedni sposób zmodyfikowane, by obsługiwać takie operacje. Od teraz zatem rozważajmy tylko jeden rodzaj aktualizacji:

- `I` – na indeksach $i, i+1, ..., j$ dodaj wartości s, $s+a$, $s+2a$, ..., $s+(j-i)a$.

Aby poprawnie obsługiwać usuwanie przekaźnika, wystarczy wczytując zapytania pamiętać, jakie parametry miał przekaźnik ostatnio postawiony w danym punkcie.

Zauważmy teraz, że moglibyśmy zastosować następujące (brutalne) podejście: dla każdej aktualizacji przejrzeć wszystkie następujące po niej zapytania, i dodać do wyniku tego zapytania wartość dodaną przez tą aktualizację. Oznacza to, że kolejność aktualizacji zapytań tak naprawdę nie ma znaczenia i możemy je rozważać oddzielnie. Korzystamy przy tym z tego, że zapytania są *offline*.

Możemy wziąć pewien podzbiór aktualizacji poprzedzających pewien podzbiór zapytań i aplikować wszystkie aktualizacje na zapytaniach, aktualizując ich wynik. Wykorzystajmy to teraz, aby wymyślić algorytm dziel i zwyciężaj.

Załóżmy, że mamy algorytm, który dla danego ciągu operacji zwraca wyniki dla wszystkich zapytań w tym ciągu, uwzględniając (tylko) aktualizacje, które się w nim znajdują, i nazwijmy go $\mathcal{A}$.

Dziel Weźmy ciąg operacji i podzielmy go na dwie połowy: L i R. Wykonajmy algorytm $\mathcal{A}$ na L oraz R, otrzymując odpowiedzi na L i R. Odpowiedzi na zapytania z R nie uwzględniają jednak aktualizacji z L, więc musimy je teraz rozważyć.

Zwyciężaj Wyniki zwrócone z $\mathcal{A}(L)$ nie ulegną zmianie, więc możemy je już zapisać. Musimy jedynie zmodyfikować $\mathcal{A}(R)$, aplikując aktualizacje z ciągu L. Wcześniej sprowadziliśmy je do jedynie `I`, więc jest to znacznie łatwiejsze.

Problem ten można rozwiązać bardzo łatwo wykorzystując kilka razy sumy prefiksowe, symulując cały ciąg długości n. Niestety, popsujemy sobie w ten sposób złożoność. Zamiast tego musimy policzyć ciąg skompresowany, który zawiera tylko elementy o indeksach interesujących dla nas w tym wywołaniu rekurencyjnym (czyli indeksach końców przedziałów). Elementy nieinteresujące ciągu powinny dodawać swoją wartość do następnika, który został uwzględniony. Przykładowo, kompresja może wyglądać o tak (indeksy wyróżnione zostały oznaczone jako interesujące):

4	2	0	0	1	4	5	4
→	6	0	→	→	6	→	9

Uważna implementacja pozwoli nam wykonać ten krok w $O(m \log m)$, gdzie m to liczba operacji w rozpatrywanym w tym wywołaniu ciągu.

Złożoność Otrzymujemy rekurencję:

$$T(m) = 2T(m/2) + O(m \log m)$$

Co daje nam złożoność $O(m \log^2 m)$. Dzięki dużym limitom czasowym jest ona wystarczająca.

Warto zauważyć, że uprościliśmy problem to rozwiązywania wyłącznie aktualizacji, a dopiero wtedy zapytań, narzucając jedynie czynnik $\log m$.