Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jakub Radoszewski
Nr albumu: 214565

Optymalna aproksymacja ciągów liczbowych ciągami monotonicznymi

Praca licencjacka
na kierunku MATEMATYKA

Praca wykonana pod kierunkiem
dra hab. Leszka Plaskoty
Instytut Matematyki Stosowanej i Mechaniki

Wrzesień 2007

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy przedstawiono algorytm znajdowania optymalnej aproksymacji n-elementowego ciągu rzeczywistego w zbiorze ciągów monotonicznych o złożoności czasowej $O(n^2 \log C \log p)$, gdzie C jest parametrem zależnym od maksymalnej wartości wyrazu danego ciągu i żądanej dokładności wyznaczenia wyniku. Za błąd aproksymacji przyjęto tutaj wartość p-tej normy odległości ciągów, traktowanych jako punkty z n-wymiarowej przestrzeni rzeczywistej. Dla konkretnych norm istnieją bardziej efektywne algorytmy takiej aproksymacji; w pracy zostały omówione algorytmy o złożoności czasowej $O(n)$ dla norm $||\cdot||_2$, $||\cdot||_4$ i $||\cdot||_\infty$, ogólny algorytm dla $||\cdot||_{2k}$ o złożoności czasowej $O(np^3 \log C)$ oraz algorytm o złożoności czasowej $O(n \log^2 n)$ dla normy $||\cdot||_1$.

Słowa kluczowe

optymalna aproksymacja, złożoność obliczeniowa

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.1 Matematyka

Klasyfikacja tematyczna

41. Approximations and expansions
41A29. Approximation with constraints

Tytuł pracy w języku angielskim

Optimal approximation of real sequences with monotonic sequences

Spis treści

Wprowadzenie

W niniejszej pracy rozwiązujemy w ogólności problem optymalnej aproksymacji ciągu liczbowego za pomocą ciągu monotonicznego. Błąd aproksymacji mierzymy w normie $||\cdot||_p$ dla całkowitego $p \geqslant 1$ lub $p = \infty$.

Praca składa się z czterech rozdziałów. W rozdziale 1 przedstawiono sformułowania problemu oraz założenia, które przyjęto za podstawę dalszych rozważań. Algorytm rozwiązujący rozważane zagadnienie w ogólnym przypadku jest szczegółowo opisany w rozdziale 2. Znaczącą jego część stanowi analiza szczególnego przypadku problemu, w którym ograniczamy się do poszukiwania optymalnej aproksymacji w zbiorze ciągów stałych. W pozostałej części pracy zaprezentowano ulepszenia ogólnego algorytmu dla pewnych konkretnych norm. W rozdziale 3 konstruujemy optymalny algorytm dla $||\cdot||_2$, na podstawie którego budujemy dość ogólny algorytm, działający w przypadku norm $||\cdot||_{2k}$, a także algorytm optymalny dla normy $||\cdot||_4$. Z kolei w rozdziale 4 rozpatrzone zostały normy $||\cdot||_\infty$ i $||\cdot||_1$. Dla pierwszej z nich zaprezentowano dwa optymalne algorytmy, dla drugiej natomiast przedstawiono algorytm o złożoności czasowej $O(n \log^2 n)$. Na końcu rozdziału znajduje się uzasadnienie, dlaczego główny pomysł pracy nie może być z powodzeniem wykorzystany do konstrukcji algorytmu o złożoności czasowej $O(n)$ dla przypadku $||\cdot||_1$.

Główne wyniki niniejszej pracy, dotyczące optymalnej aproksymacji ciągów ciągami monotonicznymi, jak i prezentowane algorytmy są oryginalne. W przypadku nieoryginalnych wyników podano ich źródła.

Rozdział 1

Postawienie problemu

Niech dany będzie ciąg liczb rzeczywistych $a = (a_1, a_2, \ldots, a_n)$. Poszukujemy takiego ciągu monotonicznego $b = (b_1, b_2, \ldots, b_n)$, który najlepiej aproksymuje ciąg a. Miarą błędu aproksymacji jest odległość ciągów a oraz b jako punktów w n-wymiarowej przestrzeni $\mathbb{R}^n$. Dokładniej, oznaczając ten błąd przez Δ mamy:

$$\Delta = ||a - b||_p = ||(a_1 - b_1, \ldots, a_n - b_n)||_p,$$

gdzie $|| \cdot ||_p$ jest normą p-tą, a p — liczbą całkowitą dodatnią lub nieskończonością. Przypomnijmy, że $||(x_1, \ldots, x_n)||_p = (|x_1|^p + \ldots + |x_n|^p)^{1/p}$ dla $p \neq \infty$ oraz $||(x_1, \ldots, x_n)||_\infty = \lim_{p\to\infty} ||(x_1, \ldots, x_n)||_p = \max(|x_1|, \ldots, |x_n|)$. Pewne konkretne normy będą nas szczególnie interesowały; będą to: $|| \cdot ||_1$, $|| \cdot ||_2$ oraz $|| \cdot ||_\infty$.

Powyższy problem można również sformułować równoważnie w następujący sposób. Dla danego zbioru punktów na płaszczyźnie $(1, a_1), \ldots, (n, a_n)$ poszukujemy funkcji monotonicznej $f : \{1, 2, \ldots, n\} \mapsto \mathbb{R}$, która najlepiej je przybliża (czyli takiej, że wartości $f(1), \ldots, f(n)$ są w wyżej opisanym sensie najbliższe wartościom $a_1, \ldots, a_n$). Różnych sformułowań problemu będziemy w dalszym ciągu używać zamiennie, w zależności od tego, które z nich będzie w danym momencie najwygodniejsze. W dalszej części pracy ograniczymy się do przypadku poszukiwania ciągu b w zbiorze ciągów niemalejących; przypadek ciągu nierosnącego b można wówczas rozwiązać tak samo, odwracając kolejność elementów w ciągu a albo rozpatrując ciąg $-a$.

W niniejszej pracy skupimy się na poszukiwaniu ogólnych algorytmów, które pozwalają znaleźć najmniejszy możliwy błąd aproksymacji w poszczególnych normach, a także skonstruować ciąg b realizujący to minimum. Jest to element najlepszej aproksymacji — oznaczany ENA — dla ciągu a, który będziemy również nazywać *elementem optymalnym*.

Lemat 1. *Dla dowolnego skończonego ciągu rzeczywistego a w zbiorze ciągów niemalejących istnieje element najlepszej aproksymacji w normie $|| \cdot ||_p$.*

Dowód. Infimum błędu w zbiorze wszystkich ciągów niemalejących jest dla ciągu a osiągane w zbiorze tych ciągów, dla których pierwszy element jest niemniejszy od najmniejszego w a, a ostatni niewiększy od największego w a. Zbiór ten jest w normie p-tej domknięty i ograniczony, a zatem zwarty. Funkcja błędu aproksymacji jako funkcja ciągła przyjmuje w tym zbiorze swoje kresy. □

Będziemy się przede wszystkim koncentrować na efektywności konstruowanych algorytmów; będziemy dążyć do uzyskania jak najlepszej złożoności czasowej. Przy opisie złożoności problem będziemy parametryzować długością n ciągu a oraz wartością p. Zakładamy, że koszt

wykonywania operacji arytmetycznych na wyrazach ciągu jest stały. W niektórych algorytmach w analizie złożoności będzie się także pojawiał parametr, zależny od żądanej dokładności wyznaczenia błędu aproksymacji oraz zakresu wartości ciągu a. Konieczność użycia tego parametru wyniknie z wykorzystania wyszukiwania ternarnego lub algorytmu bisekcji w zbiorze liczb rzeczywistych.

Rozdział 2

Ogólny algorytm dla $||\cdot||_p$

2.1. Przybliżanie ciągiem stałym

Rozważmy najpierw przypadek aproksymacji ciągów rzeczywistych ciągami stałymi. W tym celu zdefiniujmy funkcję

$$f(x) = ||a - (x, x, \ldots, x)||_p, \tag{2.1}$$

przyporządkowującą wartości ciągu stałego $(x, x, \ldots, x)$ błąd aproksymacji ciągu a. Dodatkowo oznaczmy $l_0 = \min(a_1, \ldots, a_n)$ oraz $r_0 = \max(a_1, \ldots, a_n)$. Zauważmy, że dla $x < l_0$ zachodzi $f(x) \geqslant f(l_0)$, co wynika stąd, że w tym przypadku $|a_i - x| > |a_i - l_0|$ dla wszystkich $i \in \{1, \ldots, n\}$. Podobnie dla $x > r_0$ zachodzi $f(x) \geqslant f(r_0)$. To pokazuje, że poszukiwania najlepiej aproksymującego ciąg a ciągu stałego możemy ograniczyć do przedziału $[l_0, r_0]$.

Lemat 2. *Funkcja f ma na przedziale $[l_0, r_0]$ następujący przebieg zmienności: jest najpierw ściśle malejąca, potem stała, a na koniec ściśle rosnąca. Każdy z tych trzech przedziałów może przy tym być pusty.*

Dowód. Dowód przeprowadzimy jedynie dla norm $||\cdot||_p$, gdzie $p \neq \infty$. Szczególny przypadek $p = \infty$ rozważony jest w rozdziale dedykowanym normie $||\cdot||_\infty$.

Na wstępie zauważmy, że dla dowolnego ciągu a w zbiorze ciągów stałych istnieje element najlepszej aproksymacji. Dowód tego spostrzeżenia jest analogiczny jak dowód Lematu 1, jeżeli zauważyć, że rozważany zbiór ma postać odcinka w przestrzeni $\mathbb{R}^n$, wyznaczonego przez prostą o równaniu $x_1 = \ldots = x_n$ oraz przez ograniczenie na współrzędne $l_0 \leqslant x_i \leqslant r_0$, zachodzące dla każdego $i \in \{1, \ldots, n\}$. A zatem zbiór ten jest domknięty i ograniczony, czyli zwarty, co wobec ciągłości f pokazuje, że przyjmuje ona na tym zbiorze swoje kresy.

Niech więc x_0 będzie szukaną wartością ciągu stałego, najlepiej aproksymującego ciąg a. Pokażemy, że jeżeli $x_0 < y_1 < y_2$, to błąd aproksymacji ciągu a za pomocą ciągu $c = (y_1, \ldots, y_1)$ jest niewiększy niż za pomocą ciągu $d = (y_2, \ldots, y_2)$. Niech $B(a, r)$ oznacza n-wymiarową kulę domkniętą w normie $||\cdot||_p$ o środku w punkcie a i promieniu r. Jeżeli dla pewnego $r \geqslant 0$ kula $B(a, r)$ zawiera d, to musi ona zawierać również $b = (x_0, \ldots, x_0)$, gdyż jest on elementem najlepszej aproksymacji. Stąd i z wypukłości kuli wiemy, że zawiera ona wówczas odcinek od b do d, a c oczywiście należy do tego odcinka. Innymi słowy pokazaliśmy, że dla dowolnego $r \geqslant 0$ jeżeli błąd aproksymacji ciągu a ciągiem d jest niewiększy niż r, to również błąd aproksymacji ciągu a ciągiem c jest niewiększy niż r, co kończy tę część dowodu. Analogicznie można pokazać, że dla $y_2 < y_1 < x_0$ błąd aproksymacji ciągu a za pomocą ciągu c jest niewiększy niż za pomocą ciągu d. To daje już pewne informacje na temat przebiegu zmienności funkcji f: jest ona najpierw nierosnąca, a potem niemalejąca.

Pozostało nam pokazać, że f może być stała na maksymalnie jednym podprzedziale przedziału $[l_0, r_0]$. Dalsze rozważania podzielimy na trzy przypadki.

Przypadek 1: p jest liczbą parzystą (a zatem $p \geqslant 2$). W takim razie $f^p(x) = \sum_{i=1}^{n}(a_i - x)^p$. Oznacza to, że f^p jest wielomianem zmiennej x o współczynniku n przy najwyższej potędze x (równej p). Wiadomo, że taki wielomian nie może być stały na żadnym przedziale dodatniej długości, co na mocy wcześniejszych rozważań pokazuje, że w tym przypadku funkcja f jest najpierw ściśle malejąca, a potem ściśle rosnąca.

Przypadek 2: p jest liczbą nieparzystą, niemniejszą niż 3. Zdefiniujmy ciąg $(c_i)_{i=1}^{n}$ jako posortowaną niemalejąco kopię ciągu a i podzielmy rozważany przedział $[l_0, r_0]$ na kawałki $[c_1, c_2], \ldots, [c_{n-1}, c_n]$, gdzie $c_1 = l_0$ i $c_n = r_0$. Dla j-tego z tych kawałków (przy $1 \leqslant j \leqslant n-1$) wzór funkcji f^p wygląda następująco: $f^p(x) = \sum_{i=1}^{j}(x - c_i)^p + \sum_{i=j+1}^{n}(c_i - x)^p$. Jeżeli więc $j \neq \lfloor \frac{n}{2} \rfloor$, to współczynnik przy x^p nie może być równy zeru, a zatem podobnie jak w przypadku pierwszym f nie może być stała. f^p mogłaby być stała na przedziale dodatniej długości jedynie wtedy, gdyby zachodziły wszystkie spośród warunków: $2|n$ oraz $j = \frac{n}{2}$ oraz $c_j < c_{j+1}$. W tym przypadku współczynnik przy x^{p-1} byłby jednak równy

$$-p \cdot \sum_{i=1}^{n/2} c_i + p \cdot \sum_{i=n/2+1}^{n} c_i.$$

Ponieważ jednak ciąg c jest niemalejący i ponadto $c_{n/2} < c_{n/2+1}$, to opisany współczynnik musi być dodatni, więc również w tym przypadku wielomian f^p nie może być stały. To ostatecznie pokazuje, że i tym razem f jest najpierw ściśle malejąca, a potem ściśle rosnąca.

Przypadek 3: $p = 1$. W tym przypadku możemy podobnie jak poprzednio określić ciąg c i przedstawić $f(x)$ w przedziale $[c_j, c_{j+1}]$ jako $\sum_{i=1}^{j}(x - c_i) + \sum_{i=j+1}^{n}(c_i - x)$. Łatwo widzimy, że f może być stała na przedziale $[c_j, c_{j+1}]$ wtedy i tylko wtedy, gdy $2|n$ i $j = \frac{n}{2}$ — tylko wówczas jednomian x występuje we wzorze f z zerowym współczynnikiem. Aby pokazać, że i tym razem teza zachodzi, wystarczy dowieść, że dla każdego $\epsilon > 0$, $f(c_{j+1} + \epsilon) > f(c_{j+1})$ i podobnie że $f(c_j - \epsilon) > f(c_j)$. Udowodnimy tylko pierwszą z tych nierówności, dowód drugiej jest analogiczny. Aby to uczynić, zauważmy że przesunięcie się w ramach jednego przedziału postaci $[c_i, c_{i+1}]$ o ϵ w kierunku większych wartości powoduje zmianę wartości f o przemnożoną przez ϵ różnicę między liczbą elementów ciągu c niemniejszych od c_{i+1} a liczbą elementów tego ciągu niewiększych niż c_i. To pokazuje, że takie przesunięcie powoduje wzrost wartości f, o ile pierwsza z tych liczb jest większa od drugiej, a taka właśnie sytuacja zachodzi w dowolnym fragmencie przesuwania się od c_{j+1} ku większym wartościom. To kończy dowód ostatniego z przypadków, a więc zarazem i całego lematu. □

Do wyszukiwania minimum funkcji o takim jak opisany w Lemacie 2 przebiegu zmienności można próbować wykorzystać tak zwane *wyszukiwanie ternarne* (patrz np. [Dahl], [Ral], [Sto] czy [Wiki]). Algorytm ten pozwala znaleźć poszukiwane minimum z dowolną, ale z góry zadaną dokładnością (określoną przez żądany błąd bezwzględny e wyniku). Oto pseudokod wyszukiwania ternarnego, w wersji odpowiedniej dla naszego przypadku:

```
function wyszukiwanieTernarne(f, l, r, e)
begin
  // założenie: punkt realizujący minimum f należy do przedziału [l, r].
  if (r-l < e) then
    return (l+r)/2;
  jednaTrzecia := (l*2+r)/3;
  dwieTrzecie  := (l+r*2)/3;
```

```
  if (f(jednaTrzecia) > f(dwieTrzecie)) then
    return wyszukiwanieTernarne(f, jednaTrzecia, r, e);
  else
    return wyszukiwanieTernarne(f, l, dwieTrzecie, e);
end;
```

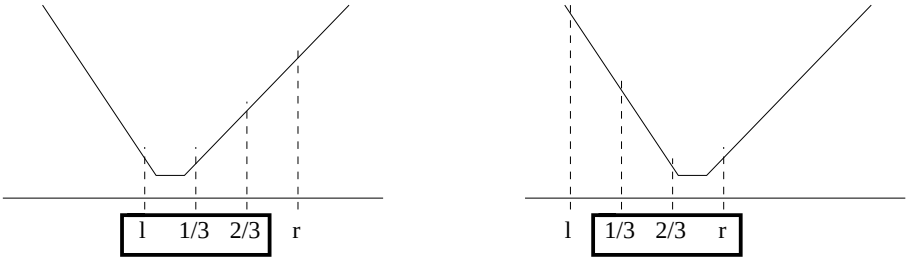
Zastanówmy się najpierw, jaka jest złożoność czasowa powyższego algorytmu, a kwestią jego poprawności zajmiemy się dalej. Na wejściu funkcji przedział zawierający jej minimum ma długość `r-l`. Zmienne pomocnicze `jednaTrzecia` i `dwieTrzecie` tworzą równoodległy podział odcinka `[l,r]`. Stąd w każdym z wywołań rekurencyjnych wewnątrz głównej instrukcji `if`, długość przekazywanego przedziału jest równa $\frac{2}{3}$ długości wyjściowego przedziału. W pierwszym wywołaniu funkcji `wyszukiwanieTernarne`, jako parametry `l` oraz `r` przekazujemy oczywiście wyżej zdefiniowane l_0 oraz r_0. Początkowa długość przedziału wyszukiwania to w takim razie $r_0 - l_0$, a końcowa to e, stąd początkowy przedział musi zostać zawężony $\frac{r_0-l_0}{e}$ razy. Podsumowując cały ten wywód mamy, że po $\lceil \log_{1.5} \left(\frac{r_0-l_0}{e} \right) \rceil$ wywołaniach rekurencyjnych (1.5 to odwrotność $\frac{2}{3}$) powyższy algorytm się zakończy. Dla uproszczenia powyższego zapisu wprowadźmy nowy parametr C, dla którego $\lceil \log_{1.5} \left(\frac{r_0-l_0}{e} \right) \rceil = O(\log C)$, a który zależy od zakresu wartości wyrazów wyjściowego ciągu oraz żądanej dokładności wyznaczenia wyniku.

Zastanówmy się na koniec, jaki jest koszt czasowy jednego wywołania rekurencyjnego funkcji `wyszukiwanieTernarne`. Nie jest to trudne pytanie; na koszt ten decydujący wpływ ma obliczanie wartości funkcji f w dwóch punktach, które sprowadza się do wyznaczenia wartości wyrażenia postaci $|a_1 - x|^p + \ldots + |a_n - x|^p$ (zauważmy, że możemy pominąć p-ty pierwiastek z tej wartości, gdyż interesują nas niekoniecznie dokładne wartości p-tej normy wyrażenia, a jedynie możliwość stwierdzenia relacji między nimi). Z kolei wartość takiego wyrażenia możemy wyznaczyć w złożoności czasowej $O(n \log (p+1))$ za pomocą algorytmu szybkiego potęgowania (dla normy $|| \cdot ||_\infty$ możemy przyjąć $p = 1$). Ostatecznie złożoność czasowa całego wyszukiwania ternarnego to w naszym przypadku $O(n \log C \log p)$.

Pozostało nam więc pokazać, że dla funkcji o przebiegu zmienności takim jak opisany w Lemacie 2 wyszukiwanie ternarne jest poprawne. W tym celu musimy pokazać następujący fakt.

Fakt 3. *Jeżeli wyjściowy przedział `[l,r]` zawierał minimum funkcji f, to także przedział przekazany do kolejnego wywołania funkcji `wyszukiwanieTernarne` spełnia tę własność.*

Dowód. Jeżeli jakikolwiek punkt realizujący minimum f należy do przedziału `[jednaTrzecia,dwieTrzecie]`, to punkt ten znajdzie się również w każdym z przedziałów w wywołaniach rekurencyjnych funkcji f. Jeżeli wszystkie poszukiwane punkty są mniejsze niż `jednaTrzecia`, to na podstawie Lematu 2 zachodzi `f(jednaTrzecia) < f(dwieTrzecie)` i zostanie wybrane wywołanie `wyszukiwanieTernarne(f, l, dwieTrzecie, e)`, czyli takie jak trzeba. Jeżeli wreszcie punkty realizujące minimum są wszystkie większe niż `dwieTrzecie`, to znów dzięki Lematowi 2 mamy `f(jednaTrzecia) > f(dwieTrzecie)` i również w tym przypadku wybór wywołania rekurencyjnego — `wyszukiwanieTernarne(f, jednaTrzecia, r, e)` — jest poprawny.



Rysunek 1: Przykładowe wykresy funkcji f wraz z informacjami o wyborze przedziału do kolejnego kroku wyszukiwania ternarnego.

□

Ostatecznie udało nam się udowodnić, że do rozwiązania problemu przybliżania ciągu ciągiem stałym możemy zastosować wyszukiwanie ternarne, co pozwala osiągnąć złożoność czasową $O(n \log C \log p)$, gdzie C jest parametrem zależnym od zakresu wartości elementów ciągu a oraz od żądanej dokładności wyznaczenia wyniku.

2.2. Algorytm dla problemu ogólnego

Przejdziemy teraz do konstrukcji rozwiązania dla właściwego problemu. Na początek pokażemy następujacą własność.

Fakt 4. *Niech dane będą dwa ciągi liczb rzeczywistych: $a = (a_1, \ldots, a_n)$ oraz ciąg niemalejący $b = (b_1, \ldots, b_n)$. Dalej, oznaczmy przez $\Delta(u, v)$ błąd aproksymacji n-elementowego ciągu u równolicznym mu ciągiem niemalejącym v, przy ustalonej normie $||\cdot||_p$. Wówczas dla dowolnego ciągu niemalejącego $c = (c_1, \ldots, c_n)$ oraz indeksu $k \in \{1, \ldots, n\}$, jeżeli*

$$\Delta((a_i)_{i=1}^k, (c_i)_{i=1}^k) \leqslant \Delta((a_i)_{i=1}^k, (b_i)_{i=1}^k)$$

oraz

$$\Delta((a_i)_{i=k+1}^n, (c_i)_{i=k+1}^n) \leqslant \Delta((a_i)_{i=k+1}^n, (b_i)_{i=k+1}^n)$$

to

$$\Delta((a_i)_{i=1}^n, (c_i)_{i=1}^n) \leqslant \Delta((a_i)_{i=1}^n, (b_i)_{i=1}^n).$$

Co więcej, jeżeli $p \neq \infty$ i którakolwiek z nierówności z założenia jest ostra, to również nierówność z tezy jest ostra.

Dowód. Jeżeli $p \neq \infty$, to $\Delta^p(u, v) = \sum_{i=1}^n |u_i - v_i|^p$. W takim razie z pierwszej nierówności z założenia, po podniesieniu do p-tej potęgi wynika, że $\sum_{i=1}^k |a_i - c_i|^p \leqslant \sum_{i=1}^k |a_i - b_i|^p$, natomiast z drugiej wynika, że $\sum_{i=k+1}^n |a_i - c_i|^p \leqslant \sum_{i=k+1}^n |a_i - b_i|^p$. Sumując te nierówności stronami otrzymujemy zatem, że

$$\Delta^p((a_i)_{i=1}^n, (c_i)_{i=1}^n) \leqslant \Delta^p((a_i)_{i=1}^n, (b_i)_{i=1}^n)$$

(w przypadku pierwszej części tezu faktu) bądź

$$\Delta^p((a_i)_{i=1}^n, (c_i)_{i=1}^n) < \Delta^p((a_i)_{i=1}^n, (b_i)_{i=1}^n)$$

(w przypadku drugiej części), co ostatecznie implikuje tezę dla $p \neq \infty$.

W przypadku $p = \infty$ jest jeszcze łatwiej, gdyż wówczas

$$\Delta((a_i)_{i=1}^n, (c_i)_{i=1}^n) = \max(\Delta((a_i)_{i=1}^k, (c_i)_{i=1}^k), \Delta((a_i)_{i=k+1}^n, (c_i)_{i=k+1}^n))$$

i podobnie

$$\Delta((a_i)_{i=1}^n, (b_i)_{i=1}^n) = \max(\Delta((a_i)_{i=1}^k, (b_i)_{i=1}^k), \Delta((a_i)_{i=k+1}^n, (b_i)_{i=k+1}^n)),$$

co wobec nierówności z założenia implikuje już tezę. □

Okazuje się, że w algorytmie rozwiązującym ogólny problem, w którym poszukujemy ciągu b w zbiorze ciągów niemalejących, często rozwiązywanym podproblemem jest znajdowanie ciągu stałego, najlepiej aproksymującego dany ciąg. Istotnie, mamy bowiem następujące twierdzenie.

Twierdzenie 5. *Jeżeli najlepszą aproksymacją w zbiorze ciągów niemalejących dla ciągu $(a_1, \ldots, a_k)$ jest ciąg stały $b_1 = \ldots = b_k = x$, a najlepszą aproksymacją dla ciągu $(a_{k+1}, \ldots, a_n)$ jest ciąg stały $b_{k+1} = \ldots = b_n = y$ i $x > y$, to istnieje najlepsza aproksymacja ciągu $(a_1, \ldots, a_n)$, która jest ciągiem stałym.*

Dowód. Niech $u = (u_1, \ldots, u_n)$ będzie najlepszą aproksymacją dla ciągu $(a_1, \ldots, a_n)$ i załóżmy, że u nie jest ciągiem stałym (w przeciwnym przypadku nie mamy czego dowodzić). Pokażemy najpierw, że możemy bez zwiększenia błędu aproksymacji przekształcić ciąg u do takiego ciągu v, że: $v_1 = \ldots = v_k \leqslant v_{k+1} = \ldots = v_n$. W dowodzie ograniczymy się do pokazania, że możemy przekształcić ciąg u do ciągu stałego w przedziale indeksów $[1, k]$; dowód dla przedziału $[k+1, n]$ jest analogiczny.

Niech $j < k$ będzie pierwszym indeksem takim, że $u_1 = \ldots = u_j < u_{j+1}$ (jeżeli szukany indeks nie istnieje, to nie mamy czego dowodzić). Jeżeli teraz $x \leqslant u_1$, to zamieniając prefiks $u_1, \ldots, u_k$ ciągu u na ciąg stały o wyrazach równych x, otrzymamy ciąg u', który dla indeksów $[1, k]$ przybliża ciąg a niegorzej niż u, a dla indeksów $[k+1, n]$ daje dokładnie taki sam błąd aproksymacji jak u. Stąd na podstawie pierwszej części Faktu 4 mamy, że u' przybliża ciąg a niegorzej niż u, co daje szukane sprowadzenie u do ciągu stałego dla indeksów $[1, k]$.

Rozważmy więc przypadek, kiedy $x > u_1$. W tej sytuacji wszystko zależy od tego, czy $\Delta((a_i)_{i=1}^j, (x)_{i=1}^j) \leqslant \Delta((a_i)_{i=1}^j, (u_i)_{i=1}^j)$.

1. Jeżeli tak, to na mocy Lematu 2, błąd aproksymacji ciągu $(a_1, \ldots, a_j)$ ciągiem stałym o elementach należących do przedziału $[u_1, x]$ jest niewiększy niż $\Delta((a_i)_{i=1}^j, (u_i)_{i=1}^j)$, co pozwala zamienić pierwsze j wyrazów ciągu u na równe $\min(u_{j+1}, x)$ bez zwiększenia błędu aproksymacji. Jeżeli $\min(u_{j+1}, x) = x$, to rozumując podobnie jak w przypadku $x \leqslant u_1$ możemy uzyskać szukany ciąg u'. W przeciwnym przypadku otrzymaliśmy ciąg, którego dłuższy niż poprzednio prefiks jest ciągiem stałym (ma długość co najmniej $j+1$). W takim wypadku całe opisane postępowanie kontynuujemy dalej. Wówczas po co najwyżej k krokach albo sprowadzimy prefiks ciągu u długości k do ciągu stałego (otrzymując szukany u'), albo natrafimy na jeden z pozostałych przypadków, które również pozwalają szukany ciąg u' otrzymać.

2. Jeżeli zaś $\Delta((a_i)_{i=1}^j, (x)_{i=1}^j) > \Delta((a_i)_{i=1}^j, (u_i)_{i=1}^j)$, to wszystko zależy od wartości parametru p. Jeżeli więc $p \neq \infty$, to zamieniając ciąg $b_1 = \ldots = b_k = x$ na ciąg $b'_1 = \ldots = b'_j = u_1$, $b'_{i+1} = \ldots = b'_k = x$ na mocy drugiej części Faktu 4 otrzymujemy lepsze przybliżenie ciągu $(a_1, \ldots, a_k)$ niż ciąg b, co przeczy założeniu. W przypadku $p = \infty$, opisana nierówność nie stanowi sprzeczności z założeniem, jeżeli tylko $\Delta((a_i)_{i=j+1}^k, (x)_{i=j+1}^k) \geqslant \Delta((a_i)_{i=1}^j, (x)_{i=1}^j)$, czyli innymi słowy na błąd aproksymacji ciągu $(a_1, \ldots, a_k)$ ciągiem b decydujący wpływ ma zakres indeksów $[j+1, k]$. W takim razie — ponownie wykorzystując Lemat 2 — dla każdego $s \in [u_1, x]$ mamy

$$\Delta((a_i)_{i=j+1}^k, (x)_{i=j+1}^k) \geqslant \Delta((a_i)_{i=1}^j, (x)_{i=1}^j) \geqslant \Delta((a_i)_{i=1}^j, (s)_{i=1}^j).$$

Ponieważ

$$\Delta((a_i)_{i=1}^k,(u_i)_{i=1}^k) \geqslant \Delta((a_i)_{i=1}^k,(x)_{i=1}^k) \geqslant \Delta((a_i)_{i=1}^j,(s)_{i=1}^j),$$

to zamieniając w ciągu u pierwsze j elementów na równe $\min(u_{j+1},x) \in [u_1,x]$ nie powiększamy błędu aproksymacji ciągu a. W zależności od wartości wyrażenia $\min(u_{j+1},x)$ znów mamy dwa przypadki, analogiczne do tych, które wystąpiły w pierwszym podpunkcie; w każdym z nich możemy sprowadzić ciąg u do poszukiwanego ciągu u' bez zwiększenia błędu aproksymacji.

Udało nam się pokazać sposób sprowadzenia ciągu u do ciągu u', dla którego $u'_1 = \ldots = u'_k$. Analogicznie postępując dla indeksów $[k+1,n]$ ostatecznie uzyskamy żądany ciąg v, którego elementy przyjmują co najwyżej dwie różne wartości.

Oznaczmy teraz $x' = v_1$, a $y' = v_{k+1}$. Jeżeli $x' = y'$, to otrzymaliśmy poszukiwane stałe optymalne przybliżenie ciągu a. Jeżeli nie, to musi zachodzić $x' < y'$, gdyż ciąg v jest niemalejący. Jeżeli $x' \leqslant x$, to możemy (bez zwiększenia błędu aproksymacji) cały k-elementowy prefiks ciągu v zamienić na prefiks stały równy x'' dla dowolnego $x'' \in [x',x]$. Wynika to z założenia, że ciąg stały równy x jest najlepszym przybliżeniem ciągu $a_1,\ldots,a_k$ oraz z połączenia tez Lematu 2 i Faktu 4. Podobnie argumentując wnioskujemy, że sufiks ciągu v długości $n-k$ możemy zamienić na ciąg stały równy y'' dla dowolnego $y'' \in [y,y']$, o ile tylko $y \leqslant y'$. Innymi słowy, prefiks v możemy „podciągnąć" ku x, a sufiks v możemy „obniżyć" ku y. Podciąganie i obniżanie traktujemy tu jako ciągłe przekształcenia, przy czym jeżeli któreś z nich nie może zostać wykonane, na przykład dlatego, że $x' > x$, to zakładamy, że jest to przekształcenie identycznościowe. Przed wykonaniem tych przekształceń zachodziło $x'_{pocz} < y'_{pocz}$. Na końcu jednoczesnego wykonywania tych przekształceń, nowa wartość x'_{kon} będzie niemniejsza od x (może być większa, jeżeli taka była na początku), a nowa wartość y'_{kon} niewiększa niż y, czyli dla nowych wartości zachodzi nierówność $x'_{kon} \geqslant x \geqslant y \geqslant y'_{kon}$. Ze względu na ciągłość przekształceń możemy stwierdzić, że w pewnym momencie ich wykonywania zachodzić musiało $x'_t = y'_t$. Ponieważ żadne z przekształceń w żadnym momencie nie powiększało całkowitego błędu aproksymacji, to otrzymany w trakcie ich wykonywania ciąg stały o wszystkich elementach równych x'_t stanowi element najlepszej aproksymacji ciągu a w zbiorze ciągów niemalejących. □

Twierdzenie 5 daje natychmiastowy pomysł na następujący iteracyjny algorytm rozwiązujący nasz problem. Na początku ciąg $a_1,\ldots,a_n$ traktujemy jako n jednoelementowych ciągów. Dla każdego z nich znamy najlepszą aproksymację w zbiorze ciągów niemalejących, bowiem dla ciągu jednoelementowego on sam jest właśnie poszukiwanym ciągiem. Z takich par postaci ciąg–aproksymacja stworzymy listę. Teraz w każdym kroku będziemy poszukiwać sąsiednich elementów listy, dla których wartość wyrazów aproksymacji pierwszego jest większa od wartości wyrazów aproksymacji drugiego. Po znalezieniu takiej pary skleimy (skonkatenujemy) ciągi, które odpowiadają tym elementom. Jako najlepszą aproksymację wynikowego ciągu, zgodnie z tezą Twierdzenia 5 możemy wybierać ciąg stały, którego wartość wyznaczymy za pomocą wyszukiwania ternarnego. Na koniec tego postępowania uzyskamy listę, w której kolejnym spójnym fragmentom ciągu odpowiadają najlepiej je aproksymujące ciągi stałe, które na dodatek będą ustawione w kolejności niemalejącej wartości wyrazów. Oznacza to, że wynikowa lista tworzy pewien ciąg niemalejący, aproksymujący ciąg a. Okazuje się, że otrzymana w ten sposób aproksymacja jest optymalna.

Stwierdzenie 6. *Lista uzyskana jako wynik działania opisanego algorytmu jest optymalną aproksymacją ciągu a.*

Dowód. Gdyby istniał inny ciąg v, który dawałby mniejszy błąd aproksymacji ciągu a, to na mocy Faktu 4 dla któregoś z fragmentów ciągu a spośród obecnych na skonstruowanej liście musiałby on stanowić lepszą aproksymację niż znaleziona, co przeczy wyżej opisanemu sposobowi konstrukcji listy. □

Pozostało nam jeszcze tylko w bardziej formalny sposób opisać poszczególne kroki naszego algorytmu, by móc oszacować jego złożoność czasową. Wprowadźmy w tym celu kilka oznaczeń. Opisana powyżej lista `L` zawiera w każdym momencie `rozmiar(L)` elementów, ponumerowanych od `1` do `rozmiar(L)`. Z każdym z nich jest związany przedział indeksów elementów ciągu `a`, który on reprezentuje (oznaczany przez `[l, u]`) oraz wartość elementów ciągu stałego, który najlepiej ten fragment ciągu `a` aproksymuje (oznaczany przez `ena`). Na liście możemy wykonywać wstawienia elementów na koniec (`wstaw`) oraz usuwanie elementów (`usuń`). Wreszcie w pseudokodzie dostęp do elementów ciągu `a` oraz listy `L` zapisujemy jak dostęp do tablic.

Jesteśmy już teraz gotowi do zapisania pseudokodu naszego algorytmu.

```
function aproksymacja(a, n, e) // e to max dopuszczalny błąd bezwzględny
                               // w kolejnych wywołaniach wysz. ternarnego
begin
  L := pustaLista;
  for i := 1 to n do
    wstaw(L, element([l = i, u = i], ena = a[i]))
  i := 1;
  while (i < rozmiar(L)) do
    if (L[i].ena > L[i+1].ena) then
    begin
      L[i].u := L[i+1].u;
      // g(l, u) to funkcja najlepszej aproksymacji dla ciągu a[l]..a[u]
      L[i].ena := wyszukiwanieTernarne(g(L[i].l, L[i].u),
                                       min(a[L[i].l], ..., a[L[i].u]),
                                       max(a[L[i].l], ..., a[L[i].u]), e);
      usuń(L, L[i+1]);
      if (i > 1) then
        i := i-1; // mogła się popsuć relacja między L[i-1].ena i L[i].ena!
    end
    else
      i := i+1;
  return L;
end;
```

Jedyną istotną różnicą między powyższym pseudokodem a opisem słownym algorytmu jest dużo sprytniejszy sposób przechodzenia po elementach listy `L`. Zamiast za każdym razem szukać kolidującej pary `(L[i], L[i+1])` od początku listy, poszukujemy jej od pierwszego naprawdę możliwego miejsca jej wystąpienia.

Pozostało nam przeanalizować złożoność czasową skonstruowanego algorytmu. Zapytajmy na początek, ile maksymalnie razy w opisanym algorytmie możemy wykonać scalenie dwóch sąsiednich fragmentów ciągu. Odpowiedź na to pytanie jest bardzo prosta: skoro każde scalenie zmniejsza `rozmiar(L)` o jeden, to możliwych jest co najwyżej $n-1$ takich scaleń. Każde scalenie wykonywane jest w złożoności czasowej $O(n \log C \log p)$, co daje ograniczenie na łączny koszt czasowy wszystkich scaleń rzędu $O(n^2 \log C \log p)$. Ze względu na użycie w opisie

złożoności czasowej pojedynczego scalenia parametru n zamiast rzeczywistej długości wynikowego ciągu, opisane szacowanie wydaje się być nieoptymalne. Tak jednak nie jest; dobierając w ciągu $(M, 1, 2, \ldots, n-1)$ odpowiednio dużą stałą M jesteśmy w stanie skonstruować przykład, dla którego liczba kroków algorytmu jest właśnie tego rzędu (w algorytmie $n-1$ razy wystąpi scalenie pierwszego elementu listy z drugim, przy czym ów pierwszy element będzie reprezentował kolejno prefiksy ciągu o długościach $1, \ldots, n-1$).

Zauważmy jednak, że nie w każdym kroku pętli `while` jest wykonywane scalenie. Dlatego przy liczeniu złożoności czasowej ogólnego algorytmu musimy uwzględnić łączną liczbę jej wykonań, a także udowodnić, że pętla się kiedykolwiek skończy! Aby to uczynić, przyjrzyjmy się wartościom wyrażenia `W=rozmiar(L)+rozmiar(L)-i` (celowo zapisanego w nieco skomplikowany sposób) po poszczególnych wykonaniach pętli. Przed pierwszym wykonaniem pętli `while` mamy `W=2*n-1`, a po ostatnim wykonaniu `W=rozmiar(L)>0`. Okazuje się, że w każdym wykonaniu pętli wartość `W` maleje o jeden. Faktycznie, jeżeli warunek głównej instrukcji `if` jest prawdziwy, to `rozmiar(L)` maleje o 1, a wartość wyrażenia `rozmiar(L)-i` się nie zmienia. W przeciwnym przypadku `rozmiar(L)` się nie zmienia, a `rozmiar(L)-i` maleje o 1. W ten sposób pokazaliśmy, że liczba wykonań pętli `while` jest niewiększa niż $2n$, czyli nie ma praktycznie żadnego wpływu na złożoność czasową algorytmu, którego wielkość danych wejściowych jest i tak $\Omega(n)$. To kończy dowód, że złożoność czasowa uzyskanego algorytmu to $O(n^2 \log C \log p)$.

Rozdział 3

Algorytmy dla $||\cdot||_{2k}$

3.1. Algorytm dla $||\cdot||_2$

W ogólnym algorytmie, działającym dla dowolnej p-tej normy, najbardziej nieefektywną częścią było wyznaczanie minimum funkcji (2.1). Usprawnienie tego elementu w istotny sposób wpływa na złożoność czasową całego algorytmu. Przyjrzyjmy się, jak pod tym względem wygląda sytuacja w przypadku $||\cdot||_2$. Dla danego ciągu a chcemy zminimalizować wartość wyrażenia $f(x) = \sqrt{(x-a_1)^2 + \ldots + (x-a_n)^2}$, lub — równoważnie — wyrażenia $f^2(x) = (x-a_1)^2+\ldots+(x-a_n)^2$. Przyjrzyjmy się jego pochodnej: $(f^2(x))' = 2(x-a_1)+\ldots+2(x-a_n) = 2nx - 2(a_1 + \ldots + a_n)$. Łatwo widzimy, że miejscem zerowym tej funkcji jest $x = \frac{a_1+\ldots+a_n}{n}$, czyli *średnia arytmetyczna* wyrazów ciągu. Ponieważ współczynnik przy x^2 w f^2 jest dodatni, to możemy stwierdzić, że zidentyfikowaliśmy w ten sposób jej minimum, czyli szukaną wartość elementów ciągu stałego, który najlepiej przybliża ciąg a.

Zastanówmy się, jak w już skonstruowanym algorytmie dobrze wykorzystać odkrytą postać odciętej minimum funkcji f; w tym celu przyjrzyjmy się fragmentowi pseudokodu, w którym scalane są dwa spójne fragmenty ciągu, reprezentowane przez `L[i]` i `L[i+1]`.

Stwierdzenie 7. *Wynikową wartość parametru* `L[i].ena` *możemy przedstawić jako*

`(L[i].ena*L[i].ile+L[i+1].ena*L[i+1].ile)/(L[i].ile+L[i+1].ile)`

gdzie parametr `ile` *reprezentuje liczbę elementów danego ciągu:*

`L[i].ile=L[i].u-L[i].l+1`.

Dowód. Wystarczy zauważyć, że wyrażenie postaci `L[i].ena*L[i].ile` to suma elementów fragmentu ciągu, reprezentowanego przez `L[i]`. W takim razie licznik powyższego wyrażenia reprezentuje sumę elementów fragmentu ciągu a, powstałego w wyniku scalenia, mianownik zaś liczbę elementów tego fragmentu. A zatem powyższy ułamek jest niczym innym jak średnią arytmetyczną całego fragmentu. □

Koszt czasowy jednego scalenia wykonanego w opisany wyżej sposób jest *stały*, a zatem łączny koszt wykonania wszystkich scaleń to $O(n)$. Dzięki temu, że — jak już wcześniej pokazaliśmy — liczba kroków pętli `while` może zostać oszacowana z góry przez $2n$, to ogólny algorytm, zmodyfikowany w opisany sposób, ma łączną złożoność czasową liniową względem n. W dodatku, w przeciwieństwie do algorytmu ogólnego, wykorzystującego wyszukiwanie ternarne, powyższy algorytm daje dokładne rozwiązanie. Udało nam się zatem skonstruować algorytm *optymalny* dla najlepszej aproksymacji ciągu ciągiem niemalejącym w $||\cdot||_2$.

3.2. Inne normy $||\cdot||_{2k}$

Można w tym miejscu postawić pytanie, dlaczego podobnej metody szukania minimum funkcji (2.1) nie zastosowaliśmy dla innych norm. Jeżeli p jest liczbą nieparzystą, to problemem jest to, że f nie jest wielomianem i dlatego nie jest łatwo szukać pierwiastków f' (co więcej f' nie wszędzie jest określona). Jeżeli $p \geqslant 4$ jest parzyste, to podobne podejście rzeczywiście działa: aby wyznaczyć minimum globalne f, wystarczy wziąć minimum z wartości tej funkcji w pierwiastkach f'. Jest tak dlatego, że przy $|x| \to \infty$ mamy $f(x) \to \infty$, a stąd minimum globalne wielomianu f musi być również minimum lokalnym (jest to prosty fakt z teorii wielomianów), które z kolei jest realizowane w jednym z pierwiastków f'. Odtąd będziemy się zatem koncentrować wyłącznie na poszukiwaniu pierwiastków wielomianu f'. Dla uproszczenia dalszych zapisów oznaczmy $g = f'$.

W przypadku $p = 4$ funkcja g jest wielomianem stopnia trzeciego i wyznaczanie jego pierwiastków można zrealizować w czasie stałym za pomocą wzorów Cardano. W tym przypadku, dla ciągu $a_l, \ldots, a_r$ mamy wzór:

$$g(y) = 4\left(y^3 \sum_{i=l}^{r} a_i^0 - 3y^2 \sum_{i=l}^{r} a_i + 3y \sum_{i=l}^{r} a_i^2 - \sum_{i=l}^{r} a_i^3\right).$$

Można teraz na wstępie stablicować sumy prefiksowe postaci $S_{r,j} = a_1^j + \ldots + a_r^j$ dla $0 \leqslant j \leqslant 3$ i $0 \leqslant r \leqslant n$, co wymaga złożoności czasowej $O(n)$ (dokładniejszy sposób wyznaczania takich sum jest opisany dalej). Wówczas żądane sumy występujące w powyższym wzorze na $g(y)$ można wyliczać jako różnice odpowiednich sum prefiksowych:

$$\sum_{i=l}^{r} a_i^j = S_{r,j} - S_{l-1,j}$$

w złożoności czasowej $O(1)$. Ostatecznie również dla $||\cdot||_4$ uzyskujemy algorytm o złożoności czasowej $O(n)$.

Dla $p \geqslant 6$, $p = 2k$ ogólne wzory na pierwiastki wielomianów stopnia $p-1$ nie istnieją. Można sobie w tym przypadku poradzić w inny sposób, na przykład numerycznie przybliżając pierwiastki funkcji g. Istnieje prosty algorytm, który może do tego służyć, opierający się na spostrzeżeniu, że pierwiastkiem g może być każdy z pierwiastków g' (jeśli tak to byłby to pierwiastek wielokrotny g), a także dla dowolnych dwóch kolejnych w posortowanym zbiorze pierwiastków g': x_1 i x_2, w przedziale (x_1, x_2) może istnieć co najwyżej jeden pierwiastek y funkcji g. Faktycznie, gdyby w przedziale (x_1, x_2) istniały dwa takie pierwiastki $x_1 < y_1 < y_2 < x_2$ funkcji g, to z twierdzenia Rolle'a wynikałoby, że w przedziale $[y_1, y_2]$ funkcja g' musi mieć pierwiastek, co jest sprzeczne z założeniem o wyborze wartości x_1 i x_2. Podobnie, w przedziałach $(-\infty, x_{min})$ i (x_{max}, ∞), gdzie x_{min} i x_{max} to odpowiednio najmniejszy i największy pierwiastek g', mogą istnieć pojedyncze pierwiastki g, ale dla uproszczenia te przedziały pomijamy w dalszej analizie.

Znalezienie wspomnianego jedynego pierwiastka w przedziale jest bardzo proste: jeżeli $g(x_1) \cdot g(x_2) \geqslant 0$, to on nie istnieje (gdyby taki pierwiastek istniał, to można pokazać, że w przedziale (x_1, x_2) funkcja g' miałaby jakiś pierwiastek, co — jak już wspominaliśmy — nie jest prawdą). W przeciwnym przypadku do jego identyfikacji można wykorzystać np. metodę bisekcji czy Newtona.

W naszym algorytmie znajdowania pierwiastków wielomianu stopnia $p-1$ w wywołaniu rekurencyjnym wyznaczamy pierwiastki wielomianu g' stopnia $p-2$, a następnie za ich pomocą w opisany w dowodzie sposób identyfikujemy pierwiastki funkcji g. Na samym początku trzeba jednak wyznaczyć funkcję g poprzez wyliczenie współczynników wielomianu stopnia

$p-1$, opisującego ją. Za pomocą współczynników g można z kolei łatwo wyznaczać współczynniki g' i kolejnych pochodnych g. Na początku głównego algorytmu wyznaczamy więc sumy prefiksowe $S_{r,j} = a_1^j + \ldots + a_r^j$ dla $1 \leqslant j \leqslant p-1$ i $0 \leqslant r \leqslant n$. Aby je wyliczyć, wyznaczamy najpierw wszystkie wartości postaci $a_i^j : i \in \{1, \ldots, n\}, j \in \{0, \ldots, p-1\}$ w kolejności wzrastających wykładników potęg, a następnie za ich pomocą wyliczamy sumy ze wzorów:

$$S_{0,j} = 0 \text{ dla } j \in \{0, \ldots, p-1\},$$

$$S_{r,j} = S_{r-1,j} + a_r^j \text{ dla } r \in \{1, \ldots, n\} \text{ oraz } j \in \{0, \ldots, p-1\}.$$

Złożoność czasowa tej fazy to $O(np)$. Dodatkowo należy wyznaczyć na wstępie wartości wszystkich niezerowych współczynników dwumianowych $\binom{x}{y}$ dla $x \leqslant p-1$ naturalnego (złożoność czasowa $O(p^2)$, za pomocą trójkąta Pascala). Łączna złożoność czasowa przedstawionych wstępnych obliczeń to $O((n+p)p)$.

Jeżeli teraz przy scalaniu elementów listy wynikowym fragmentem ciągu a jest przedział elementów od a_l do a_r, to możemy za pomocą wcześniej policzonych wartości pomocniczych w złożoności $O(p)$ wyznaczyć współczynniki wielomianu g:

$$g(x) = f'(x) = \sum_{i=0}^{p-1} p(x - a_i)^{p-1}.$$

Faktycznie, dla każdego $0 \leqslant i \leqslant p-1$ współczynnik przy x^i jest w nim równy

$$\binom{p-1}{i}(-1)^{p-1-i}\left(a_l^{p-1-i} + \ldots + a_r^{p-1-i}\right) = \binom{p-1}{i}(-1)^{p-1-i}(S_{r,p-1-i} - S_{l-1,p-1-i}).$$

Przypomnijmy, że w całym rozumowaniu zakładamy, że operacje na pojawiających się w trakcie działania liczbach zmiennoprzecinkowych możemy wykonywać w czasie stałym.

Zastanówmy się teraz, jaki jest koszt czasowy jednego wywołania rekurencyjnego wyżej opisanego algorytmu szukania pierwiastków wielomianu g, jeżeli mamy już obliczone jego współczynniki. Dla każdego z co najwyżej $p-1$ pierwiastków g' należy sprawdzić, czy jest on pierwiastkiem g (łączny czas $O(p^2)$, jeżeli dla każdego kandydata na pierwiastek skorzystamy ze schematu Hornera). Teraz na każdym z potencjalnie p przedziałów należy wyszukać pierwiastek funkcji g, co — przy zastosowaniu na przykład metody bisekcji — wymaga łącznie dla wszystkich przedziałów czasu $O(p^2 \log C)$, gdzie C jest stałą zdefiniowaną podobnie jak w ogólnym algorytmie, a dodatkowy czynnik p wynika z konieczności każdorazowego wyznaczania wartości g. Ponieważ w całym algorytmie mamy maksymalnie p wywołań rekurencyjnych, stąd złożoność czasowa całego algorytmu szukania pierwiastków może zostać oszacowana przez $O(p^3 \log C)$. Algorytm ten jest wykorzystywany przy każdym scaleniu elementów listy `L`, co wobec pesymistycznej liczby takich scaleń rzędu $O(n)$ daje łączną złożoność algorytmu $O((n+p)p)$ (koszt wstępnych obliczeń) + $O(np^3 \log C)$ (koszt wszystkich scaleń) = $O(np^3 \log C)$. Dla małych wartości p i długich ciągów algorytm ten może więc być znacznie szybszy niż opisana ogólna metoda. W tym miejscu nie będziemy się zajmować porównaniem pozostałych własności tych algorytmów, jak na przykład ich stabilności numerycznej, itp.

Rozdział 4

Algorytmy dla $||\cdot||_\infty$ i $||\cdot||_1$

4.1. Algorytmy dla $||\cdot||_\infty$

Podobnie jak w przypadku $||\cdot||_2$, i tym razem postaramy się przyspieszyć ogólny algorytm, pokazując jak znaleźć minimum funkcji (2.1). Jeżeli przez a_{min} i a_{max} oznaczymy odpowiednio najmniejszy i największy element ciągu a, to minimum funkcji f skonstruowanej dla ciągu a jest jak łatwo zauważyć równe dokładnie $\frac{a_{max}-a_{min}}{2}$, a jest ono realizowane dla ciągu stałego o wartości elementów $x = \frac{a_{min}+a_{max}}{2}$. Po poczynieniu tego spostrzeżenia możemy zauważyć, że zaległy dowód Lematu 2 dla przypadku $||\cdot||_\infty$ jest bardzo prosty.

Dowód. Jeżeli $y > x$, to $f(y)$ jest równe dokładnie $y - a_{min}$, co pokazuje, że wraz ze wzrostem y, $f(y)$ ściśle rośnie. Analogiczny wzór $f(y) = a_{max} - y$ pokazuje, że w przypadku $y < x$, $f(y)$ ściśle maleje, gdy y wzrasta. □

Zastanówmy się jak wykorzystać poznaną postać odciętej minimum funkcji f do skutecznej modyfikacji ogólnego algorytmu w przypadku $||\cdot||_\infty$. Jeżeli z każdym elementem listy `L` związać minimalny i maksymalny wyraz fragmentu ciągu, któremu on odpowiada, to przy scalaniu:

- jesteśmy w stanie wyznaczyć owe elementy — minimalny i maksymalny — dla wynikowego fragmentu ciągu, biorąc minimum i maksimum z odpowiednich wartości dla fragmentów scalanych,
- możemy obliczyć wartość x za pomocą średniej arytmetycznej wartości wyznaczonych w poprzednim kroku.

Ponieważ i tym razem udało nam się osiągnąć złożoność czasową pojedynczego scalenia $O(1)$, to ponownie złożoność czasowa całego algorytmu wynosi $O(n)$.

Jako ciekawostkę możemy dodać, że dla normy $||\cdot||_\infty$ istnieje łatwiejszy algorytm, znajdujący najlepszą niemalejącą aproksymację, również posiadający liniową złożoność czasową. Można mianowicie udowodnić, że najmniejszy błąd aproksymacji Δ, jaki można osiągnąć dla danego ciągu a wyraża się wzorem

$$\max(\frac{a_i - a_j}{2} : i \leqslant j).$$

Wyznaczenie tej wartości można zrealizować w złożoności czasowej $O(n)$ za pomocą algorytmu opisanego przez poniższy pseudokod.

```
function najmniejszyBlad(a, n)
begin
  maks := a[1]; wyn := 0;
  for i := 2 to n do
  begin
    maks := max(maks, a[i]);
    wyn  := max(wyn, (maks-a[i])/2);
    // Niezmienniki: maks = max(a[u] : 1 <= u <= i),
    //               wyn  = max((a[u]-a[v])/2 : 1 <= u <= v <= i).
  end;
  return wyn;
end;
```

Znając wartość Δ możemy wyznaczyć szukany ciąg b. Oczywiście $b_i \geqslant a_i - \Delta$, $1 \leqslant i \leqslant n$. W takim razie można pokazać, że prosty algorytm zachłanny, który za b_i przyjmuje najmniejszą możliwą (czyli $\geqslant b_{i-1}$) liczbę niemniejszą od $a_i - \Delta$ skonstruuje żądany ciąg w złożoności czasowej $O(n)$.

4.2. Algorytm dla $||\cdot||_1$ i dalsze rozważania

W przypadku $||\cdot||_1$ również istnieje możliwość usprawnienia ogólnego algorytmu. Zastanówmy się najpierw, jak w tym przypadku zidentyfikować minimum funkcji (2.1). Niech $c_1, \ldots, c_n$ będzie posortowaną niemalejąco kopią ciągu wyjściowego a, dla którego konstruujemy funkcję f. Jak już wcześniej uzasadnialiśmy, poszukiwana odcięta minimum f musi należeć do przedziału $[c_1, c_n]$. W poszukiwaniu minimum f wystartujemy zatem z punktu $y = c_1$ i będziemy się przemieszczać w kierunku c_n, zastanawiając się, jak zmienia się wartość funkcji f. Jeżeli przemieścimy się parametrem y w ramach przedziału $[c_1, c_2]$ o $\epsilon > 0$, to wartość $f(y)$ zmieni się o $\epsilon \cdot (l - u)$, gdzie l to liczba wyrazów ciągu c niewiększych od c_1, a u to liczba wyrazów ciągu niemniejszych niż c_2. Wstawiając $l = 1$ i $u = n - 1$ otrzymamy zmianę o $\epsilon \cdot (2 - n)$. Stale zwiększając wartość x, w pewnym momencie przejdziemy do przedziału $[c_2, c_3]$; jak łatwo policzyć, przemieszczenie się w ramach niego o ϵ powoduje zmianę wartości $f(y)$ o $\epsilon \cdot (2 - (n - 2)) = \epsilon \cdot (4 - n)$. Proste uogólnienie tego rozważania pozwala stwierdzić, że przemieszczenie się w ramach przedziału $[c_i, c_{i+1}]$ o ϵ powoduje zmianę $f(y)$ o $\epsilon \cdot (2i - n)$. Dopóki wartość $2i - n$ jest ujemna, to opłaca nam się zwiększać y, gdyż wtedy $f(y)$ maleje. $2i - n < 0$ wtedy i tylko wtedy, gdy $i < \frac{n}{2}$, co jest równoważne temu, że opłaca nam się zwiększać y do momentu, kiedy dotrzemy do $y = c_{\lceil \frac{n}{2} \rceil}$; za tym punktem wartość wyrażenia $2i-n$ zaczyna stopniowo wzrastać. Wspomniany element środkowy ciągu c jest niczym innym jak *medianą* ciągu a (zdefiniowaną właśnie jako środkowy element po posortowaniu).

Musimy zatem w jakiś sposób wzbogacić elementy listy `L`, tak aby w momencie scalania dało się łatwo wyznaczać medianę wynikowego ciągu. Zapytania o medianę nietrudno zrealizować, jeżeli posiłkujemy się zrównoważonymi drzewami poszukiań binarnych, na przykład typu AVL (patrz np. [BDR]) czy czerwono-czarnymi (patrz np. [Cormen]) do utrzymywania elementów fragmentów ciągu na liście. Drzewa tego typu można, odpowiednio wzbogacając dane utrzymywane w węzłach, wykorzystać do wyznaczania statystyk pozycyjnych ciągów w nich przechowywanych w złożoności czasowej $O(\log n)$ (po dokładniejszy opis sposobu takiego wzbogacenia tych drzew odsyłamy do wymienionej literatury). i-ta statystyka pozycyjna ciągu jest zdefiniowana przy tym jako wartość i-tego elementu posortowanej niemalejąco kopii rozważanego ciągu. W szczególności mediana ciągu długości n to jego statystyka pozy-

cyjna o numerze $\lfloor \frac{n}{2} \rfloor$, co pokazuje, że za pomocą drzew zrównoważonych możemy wyznaczać medianę ciągu w złożoności czasowej $O(\log n)$.

Problem w bezpośrednim zastosowaniu drzew zrównoważonych do rozwiązania rozważanego zagadnienia stanowi samo utrzymywanie wspomnianych tych struktur przypisanych do poszczególnych elementów listy, a dokładniej łączenie tych drzew przy scalaniu fragmentów ciągu. Okazuje się, że całkiem efektywne rozwiązanie powstaje, jeżeli łączyć drzewa każdorazowo przenosząc wszystkie elementy z *mniejszego* drzewa (czyli zawierającego mniej elementów ciągu) do większego. Przeniesienie elementu odpowiada usunięciu go z jednego drzewa i wstawieniu do drugiego. Przypomnijmy, że koszt czasowy wstawienia elementu do drzewa zrównoważonego czy też usunięcia elementu z drzewa to $O(\log n)$. W takim razie pojedyncze scalenie drzew może nawet wymagać czasu $O(n \log n)$, ale — jak dalej pokażemy — zamortyzowana złożoność czasowa algorytmu to zaledwie $O(n \log^2 n)$. Zastanówmy się mianowicie, ile razy dany element a_i może być w całym algorytmie przenoszony między drzewami. Zauważmy, że po wykonaniu scalenia, w którym wspomniane przeniesienie następuje, rozmiar drzewa AVL, do którego ów element należy wzrasta co najmniej dwukrotnie. Ponieważ maksymalny rozmiar drzewa, jakie może powstać w trakcie działania algorytmu to oczywiście n, to dla każdego i element a_i może zostać przeniesiony co najwyżej $\log n$ razy. Ponieważ elementów ciągu jest n, a koszt czasowy przeniesienia jednego elementu to $O(\log n)$ (usunięcie z mniejszego drzewa i wstawienie do większego drzewa), złożoność czasowa całego algorytmu wynosi $O(n \log^2 n)$.

Tak skonstruowany algorytm nie jest już bardzo odległy pod względem złożoności czasowej od liniowego, aczkolwiek rzeczywiście optymalny nie jest. Istnieje algorytm o złożoności czasowej $O(n \log n)$, który rozwiązuje problem aproksymacji ciągiem monotonicznym w normie $|| \cdot ||_1$. Jest on jednak oparty na pomyśle zupełnie innym niż algorytm ogólny. Najpierw przeskalowujemy w nim wszystkie elementy ciągu, tak by stały się liczbami całkowitymi z niedużego zakresu, a następnie optymalizujemy algorytm, który rozwiązuje taki problem za pomocą programowania dynamicznego, redukując jego złożoność czasową z $O(n^2)$ do $O(n \log n)$. Ze względu na niewielką korzyść z wprowadzenia tego algorytmu, niełatwy jego opis pomijamy.

Rodzi się jednak pytanie, czy rozważany problem da się rozwiązać w złożoności czasowej $O(n)$. Odpowiedź na to pytanie nie jest znana autorowi pracy. Można jednak pokazać, że jeżeli taki algorytm istnieje, to zapewne musi być oparty na idei innej niż ogólny algorytm opisany w pracy. Jest tak dlatego, że nasz algorytm w czasie działania wyznacza optymalne niemalejące aproksymacje dla wszystkich prefiksów ciągu a. Aby to zauważyć, można sobie wyobrazić sytuację, w której na początku nie umieszczamy na liście `L` wszystkich elementów ciągu a, a tylko pierwszy, a kolejne elementy dokładamy wtedy, kiedy pętla algorytmu dochodzi do końca listy. Zawartości listy w tych momentach to szukane najlepsze aproksymacje dla prefiksów ciągu. Gdyby taki algorytm działał w złożoności czasowej $o(n \log n)$, to — jak za chwilę pokażemy — za jego pomocą bylibyśmy w stanie w takiej właśnie złożoności sortować dowolne n-elementowe ciągi liczbowe. Zakładając model porównaniowy można jednakże pokazać, że sortowanie ciągów wymaga złożoności czasowej $\Omega(n \log n)$, zob. np. [BDR].

Pozostaje nam skonstruować sposób sortowania ciągów, oparty na naszym algorytmie znajdowania najlepszej aproksymacji. Niech dany będzie ciąg $a_1, \ldots, a_n$, który chcemy posortować. Skonstruujmy nowy ciąg $a'_1, \ldots, a'_{4n}$, zdefiniowany jak następuje:

- $a'_1 = \ldots = a'_n = \infty$,
- $a'_{n+1} = a_1$, …, $a'_{2n} = a_n$,
- $a'_{2n+1} = \ldots = a'_{4n} = -\infty$.

W powyższym zapisie przez ∞ została oznaczona stała $\max(|a_1|, \ldots, |a_n|) + 1$, większa co do wartości bezwzględnej od wszystkich elementów ciągu a. Okazuje się, że najlepszą aproksymacją każdego prefiksu ciągu a' jest ciąg stały. Można to udowodnić analizując przebieg działania naszego algorytmu dla tego ciągu. Dla uproszczenia przeanalizujemy jego równoważną wersję, w której scalenie elementów listy `L[i]` oraz `L[i+1]` wykonywane jest także wówczas, kiedy `L[i].ena=L[i+1].ena`. Każdy z n pierwszych prefiksów ma jednoznaczną najlepszą aproksymację, będącą ciągiem stałym o elementach równych ∞. Dla każdego z kolejnych n elementów ciągu a' (są to kolejne elementy ciągu a), tuż przed jego dodaniem do `L` lista ta będzie jednoelementowa i reprezentować będzie ciąg o medianie ∞; w takim razie nowy element zostanie każdorazowo scalony z `L[1]`. Ostatnie $2n$ elementów ciągu a' będzie zawsze mniejsze od mediany `L[1]`, a zatem i one zostaną scalone z tym elementem.

Zastanówmy się, jaka będzie wartość `L[1].ena` dla prefiksów a' o długościach $2n + 1, 2n + 3, \ldots, 4n - 1$. Łatwo widać, że będzie ona równa kolejno $c_n, c_{n-1}, \ldots, c_1$, gdzie ciąg $c = (c_1, \ldots, c_n)$ to posortowana niemalejąco kopia ciągu a (takie są właśnie mediany odpowiednich prefiksów ciągu a'). A zatem na podstawie wspomnianych wartości jesteśmy w stanie posortować ciąg a, uruchamiając nasz algorytm dla ciągu długości $4n = O(n)$.

Bibliografia

[BDR] L. Banachowski, K. Diks, W. Rytter „Algorytmy i struktury danych", WNT 1996.

[Cormen] T. Cormen, C. Leiserson, R. Rivest, C. Stein „Wprowadzenie do algorytmów", WNT 2004.

[Dahl] G. Dahlquist, A. Bjorck „Metody numeryczne", PWN 1983.

[Ral] A. Ralston „Wstęp do analizy numerycznej", PWN 1971.

[Sto] J. Stoer, R. Bulirsch „Wstęp do analizy numerycznej", PWN 1987.

[Wazn] http://wazniak.mimuw.edu.pl

[Wiki] http://en.wikipedia.org/wiki/Ternary_search