

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

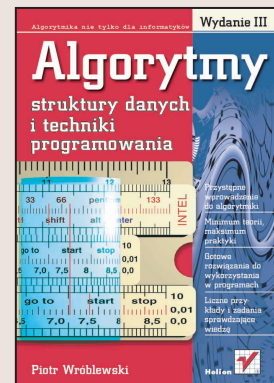
FRAGMENTY KSIĄŻEK ONLINE

Algorytmy, struktury danych i techniki programowania. Wydanie III

Autor: Piotr Wróblewski

ISBN: 83-7361-101-0

Format: B5, stron: 360



Algorytmika stanowi gałąź wiedzy, która w ciągu ostatnich kilkudziesięciu lat dostarczyła wielu efektywnych narzędzi wspomagających rozwiązywanie różnorodnych problemów za pomocą komputera. Teoria algorytmów i struktur danych jest jednym z podstawowych przedmiotów wykładanych na studiach informatycznych i pokrewnych.

Książka, której trzecie i poprawione wydanie trzymasz w ręku, od wielu lat stanowi podstawowy podręcznik z dziedziny algorytmiki. Różni się od klasycznych podręczników akademickich: skierowana jest nie tylko do adeptów informatyki. Dzięki naciskowi na praktyczną stronę prezentowanych zagadnień powinna zainteresować także osoby programujące hobbystycznie, jak również tych wszystkich, dla których programowanie jest działalnością ważną, lecz nie podstawową w pracy zawodowej. Jest to nowoczesny podręcznik dla wszystkich, którzy w codziennej pracy programistycznej odczuwają potrzebę szybkiego odzyskania pewnych informacji z dziedziny algorytmiki w celu zastosowania ich w swoich programach.

W książce opisano m.in.:

- Techniki rekurencyjne: co to jest rekurencja i jak ją stosować w praktyce?
- Sortowanie danych: najpopularniejsze procedury sortujące.
- Struktury danych: listy, kolejki, zbiory i drzewa w ujęciu praktycznym.
- Derekursywacja: jak zmienić program rekurencyjny (czasami bardzo czasochłonny) na wersję iteracyjną?
- Algorytmy przeszukiwania: przeszukiwanie liniowe, binarne i transformacja liniowa (ang. hashing).
- Przeszukiwanie tekstów: opis najbardziej znanych metod przeszukiwania tekstów (Boyer'a i Moore'a, Rabina i Karpa, brute-force, K-M-P).
- Zaawansowane techniki programowania: dziel i rządź, programowanie dynamiczne, algorytmy żarłoczne (ang. greedy).
- Algorytmika grafów: opis jednej z najciekawszych struktur danych występujących w informatyce.
- Sztuczna inteligencja: czy komputery mogą myśleć?
- Kodowanie i kompresja danych: opis najpopularniejszych metod kodowania i kompresji danych – systemu kryptograficznego z kluczem publicznym i metody Huffmana

W książce znajdziesz również liczne przykłady i zadania, które pomogą Ci sprawdzić swoją wiedzę. Kod źródłowy znajdziesz na dołączonej dyskietce.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl



Spis treści

Przedmowa	11
Rozdział 1. Zanim wystartujemy	19
1.1. Jak to wcześniej bywało, czyli wyjątki z historii maszyn algorytmicznych	21
1.2. Jak to się niedawno odbyło, czyli o tym, kto „wymyślił” metodologię programowania	23
1.3. Proces koncepcji programów	24
1.4. Poziomy abstrakcji opisu i wybór języka	25
1.5. Poprawność algorytmów	27
Rozdział 2. Rekurencja	29
2.1. Definicja rekurencji	29
2.2. Ilustracja pojęcia rekurencji	31
2.3. Jak wykonują się programy rekurencyjne?	32
2.4. Niebezpieczeństwa rekurencji	34
2.4.1. Ciąg Fibonacciego	34
2.4.2. Stack overflow!	36
2.5. Pułapek ciąg dalszy	37
2.5.1. Stąd do wieczności	37
2.5.2. Definicja poprawna, ale...	38
2.6. Typy programów rekurencyjnych	39
2.7. Myślenie rekurencyjne	41
2.7.1. Przykład 1.: Spirala	42
2.7.2. Przykład 2.: Kwadraty „parzyste”	44
2.8. Uwagi praktyczne na temat technik rekurencyjnych	45
2.9. Zadania	46
2.9.1. Zadanie 2.1	46
2.9.2. Zadanie 2.2	46
2.9.3. Zadanie 2.3	48
2.9.4. Zadanie 2.4	48
2.9.5. Zadanie 2.5	49
2.10. Rozwiązania i wskazówki do zadań	49
2.10.1. Zadanie 2.1	49
2.10.2. Zadanie 2.2	50
2.10.3. Zadanie 2.3	50
2.10.4. Zadanie 2.4	51
2.10.5. Zadanie 2.5	52

Rozdział 3. Analiza sprawności algorytmów	53
3.1. Dobrze samopoczucie użytkownika programu	54
3.2. Przykład 1: Jeszcze raz funkcja silnia	56
3.3. Przykład 2: Zerowanie fragmentu tablicy	60
3.4. Przykład 3: Wpadamy w pułapkę	62
3.5. Przykład 4: Różne typy złożoności obliczeniowej	63
3.6. Nowe zadanie: uprościć obliczenia!	66
3.7. Analiza programów rekurencyjnych	66
3.7.1. Terminologia	67
3.7.2. Ilustracja metody na przykładzie	68
3.7.3. Rozkład logarytmiczny	70
3.7.4. Zamiana dziedziny równania rekurencyjnego	71
3.7.5. Funkcja Ackermanna, czyli coś dla smakoszy	72
3.8. Zadania	73
3.8.1. Zadanie 3.1	73
3.8.2. Zadanie 3.2	74
3.8.3. Zadanie 3.3	74
3.8.4. Zadanie 3.4	74
3.9. Rozwiązania i wskazówki do zadań	75
3.9.1. Zadanie 3.2	75
3.9.2. Zadanie 3.4	76
Rozdział 4. Algorytmy sortowania	77
4.1. Sortowanie przez wstawianie, algorytm klasy $O(N^2)$	78
4.2. Sortowanie bąbelkowe, algorytm klasy $O(N^2)$	80
4.3. Quicksort, algorytm klasy $O(N \log N)$	82
4.4. Heap Sort — sortowanie przez kopcowanie	85
4.5. Sortowanie przez scalanie	88
4.6. Uwagi praktyczne	90
Rozdział 5. Struktury danych	93
5.1. Listy jednokierunkowe	94
5.1.1. Realizacja struktur danych listy jednokierunkowej	96
5.1.2. Tworzenie listy jednokierunkowej	97
5.1.3. Listy jednokierunkowe — teoria i rzeczywistość	106
5.2. Tablicowa implementacja list	119
5.2.1. Klasyczna reprezentacja tablicowa	119
5.2.2. Metoda tablic równoległych	121
5.2.3. Listy innych typów	123
5.3. Stos	125
5.3.1. Zasada działania stosu	125
5.4. Kolejki FIFO	129
5.5. Sterty i kolejki priorytetowe	132
5.6. Drzewa i ich reprezentacje	139
5.6.1. Drzewa binarne i wyrażenia arytmetyczne	142
5.7. Uniwersalna struktura słownikowa	147
5.8. Zbiory	152
5.9. Zadania	155
5.9.1. Zadanie 5.1	155
5.9.2. Zadanie 5.2	155
5.9.3. Zadanie 5.3	155
5.10. Rozwiązania zadań	155
5.10.1. Zadanie 5.1	155
5.10.2. Zadanie 5.3	156

Rozdział 6. Derekursywacja i optymalizacja algorytmów	157
6.1. Jak pracuje kompilator?	158
6.2. Odrobina formalizmu nie zaszkodzi!.....	160
6.3. Kilka przykładów derekursywacji algorytmów	162
6.4. Derekursywacja z wykorzystaniem stosu	165
6.4.1. Eliminacja zmiennych lokalnych.....	166
6.5. Metoda funkcji przeciwnych.....	168
6.6. Klasyczne schematy derekursywacji.....	170
6.6.1. Schemat typu while.....	171
6.6.2. Schemat typu if-else.....	173
6.6.3. Schemat z podwójnym wywołaniem rekurencyjnym	175
6.7. Podsumowanie	177
Rozdział 7. Algorytmy przeszukiwania	179
7.1. Przeszukiwanie liniowe.....	179
7.2. Przeszukiwanie binarne.....	180
7.3. Transformacja kluczowa (hashing).....	181
7.3.1. W poszukiwaniu funkcji H	183
7.3.2. Najbardziej znane funkcje H.....	184
7.3.3. Obsługa konfliktów dostępu	187
7.3.4. Powrót do źródeł	187
7.3.5. Jeszcze raz tablice!.....	188
7.3.6. Próbkowanie liniowe	189
7.3.7. Podwójne kluczowanie	191
7.3.8. Zastosowania transformacji kluczowej.....	193
7.3.9. Podsumowanie metod transformacji kluczowej	193
Rozdział 8. Przeszukiwanie tekstów.....	195
8.1. Algorytm typu brute-force	195
8.2. Nowe algorytmy poszukiwań.....	197
8.2.1. Algorytm K-M-P.....	198
8.2.2. Algorytm Boyera i Moore'a.....	203
8.2.3. Algorytm Rabina i Karpa.....	205
Rozdział 9. Zaawansowane techniki programowania.....	209
9.1. Programowanie typu „dziel i zwyciężaj”	210
9.1.1. Odszukiwanie minimum i maksimum w tablicy liczb.....	211
9.1.2. Mnożenie macierzy o rozmiarze $N \times N$	214
9.1.3. Mnożenie liczb całkowitych	217
9.1.4. Inne znane algorytmy „dziel i zwyciężaj”	218
9.2. Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas... ..	219
9.2.1. Problem plecakowy, czyli niełatwe jest życie turysty-piechura	220
9.3. Programowanie dynamiczne	223
9.4. Uwagi bibliograficzne	227
Rozdział 10. Elementy algorytmiki grafów	229
10.1. Definicje i pojęcia podstawowe	230
10.2. Sposoby reprezentacji grafów	232
10.3. Podstawowe operacje na grafach	234
10.3.1. Suma grafów	234
10.3.2. Kompozycja grafów	234
10.3.3. Potęga grafu	235
10.4. Algorytm Roy-Warshalla	236
10.5. Algorytm Floyd-Warshalla.....	239
10.6. Algorytm Dijkstry	243

10.7. Drzewo rozpinające minimalne.....	244
10.8. Przeszukiwanie grafów	245
10.8.1. Strategia „w głąb”	246
10.8.2. Strategia „wszerz”	247
10.9. Problem właściwego doboru	249
10.10. Podsumowanie	254
Rozdział 11. Algorytmy numeryczne.....	255
11.1. Poszukiwanie miejsc zerowych funkcji	255
11.2. Iteracyjne obliczanie wartości funkcji.....	257
11.3. Interpolacja funkcji metodą Lagrange’a	258
11.4. Różniczkowanie funkcji	260
11.5. Całkowanie funkcji metodą Simpsona.....	262
11.6. Rozwiązanie układów równań liniowych metodą Gaussa	263
11.7. Uwagi końcowe.....	266
Rozdział 12. Czy komputery mogą myśleć?	267
12.1. Przegląd obszarów zainteresowań Sztucznej Inteligencji	268
12.1.1. Systemy eksperckie.....	269
12.1.2. Sieci neuronowe.....	271
12.2. Reprezentacja problemów	272
12.2.1. Ćwiczenie 12.1.....	274
12.3. Gry dwuosobowe i drzewa gier.....	274
12.4. Algorytm mini-max.....	276
Rozdział 13. Kodowanie i kompresja danych	281
13.1. Kodowanie danych i arytmetyka dużych liczb.....	283
13.1.1. Kodowanie symetryczne.....	284
13.1.2. Kodowanie asymetryczne	285
13.1.3. Metody prymitywne.....	292
13.1.4. Łamanie szyfrów.....	293
13.2. Techniki kompresji danych	294
13.2.1. Kompresja poprzez modelowanie matematyczne.....	296
13.2.2. Kompresja metodą RLE.....	297
13.2.3. Kompresja danych metodą Huffmana	298
13.2.4. Kodowanie LZW	304
13.2.5. Idea kodowania słownikowego na przykładach	304
13.2.6. Opis formatu GIF	307
Rozdział 14. Zadania różne.....	311
14.1. Teksty zadań.....	311
14.1.1. Zadanie 14.1.....	311
14.1.2. Zadanie 14.2.....	312
14.1.3. Zadanie 14.3.....	312
14.1.4. Zadanie 14.4.....	312
14.1.5. Zadanie 14.5.....	312
14.1.6. Zadanie 14.6.....	312
14.1.7. Zadanie 14.7.....	312
14.1.8. Zadanie 14.8.....	313
14.1.9. Zadanie 14.9.....	313
14.1.10. Zadanie 14.10.....	313
14.1.11. Zadanie 14.11.....	313
14.1.12. Zadanie 14.12.....	313

14.2. Rozwiązania	313
14.2.1. Zadanie 14.1.....	313
14.2.2. Zadanie 14.3.....	314
14.2.3. Zadanie 14.4.....	315
14.2.4. Zadanie 14.10.....	315
14.2.5. Zadanie 14.12.....	316
Dodatek A Poznaj C++ w pięć minut!	319
Dodatek B Systemy obliczeniowe w pigułce.....	337
Literatura	343
Spis ilustracji	345
Spis tablic	349
Skorowidz.....	351

Rozdział 8.

Przeszukiwanie tekstów

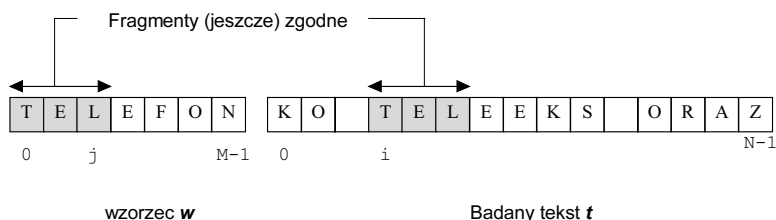
Zanim na dobre zanurzymy się w lekturę nowego rozdziału, należy wyjaśnić pewne nieporozumienie, które może towarzyszyć jego tytułowi. Otóż za *tekst* będziemy uważali ciąg znaków w sensie informatycznym. Nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną”! Tekstem będzie na przykład również ciąg bitów¹, który tylko przez umowność może być podzielony na równej wielkości porcje, którym przyporządkowano pewien kod liczbowy².

Okazuje się wszelako, że przyjęcie konwencji dotyczących interpretacji informacji ułatwia wiele operacji na niej. Dlatego też pozostaniemy przy ogólnikowym stwierdzeniu „tekst” wiedząc, że za tym określeniem może się kryć dość sporo znaczeń.

8.1. Algorytm typu brute-force

Zadaniem, które będziemy usiłowali wspólnie rozwiązać, jest poszukiwanie wzorca³ w o długości M znaków w tekście t o długości N . Z łatwością możemy zaproponować dość oczywisty algorytm rozwiązujący to zadanie, bazując na pomysłach symbolicznie przedstawionych na rysunku 8.1.

Rysunek 8.1.
Algorytm typu brute-force przeszukiwania tekstu



¹ Reprezentujący np. pamięć ekranu.

² Np. ASCII lub dowolny inny.

³ Ang. *pattern matching*.

Zarezerwujmy indeksy j i i do poruszania się odpowiednio we wzorcu i tekście podczas operacji porównywania znak po znaku zgodności wzorca z tekstem. Załóżmy, że w trakcie poszukiwań obszary objęte szarym kolorem na rysunku okazały się zgodne. Po stwierdzeniu tego faktu przesuwamy się zarówno we wzorcu, jak i w tekście o jedną pozycję do przodu ($i++$; $j++$).

Cóż się jednak powinno stać z indeksami i oraz j podczas stwierdzenia niezgodności znaków? W takiej sytuacji całe poszukiwanie kończy się porażką, co zmusza nas do anulowania „szarej strefy” zgodności. Czynimy to poprzez cofnięcie się w tekście o to, co było zgodne, czyli o $j-1$ znaków, wyzerowując przy okazji j . Omówmy jeszcze moment stwierdzenia całkowitej zgodności wzorca z tekstem. Kiedy to nastąpi? Otóż nietrudno zauważyć, że podczas stwierdzenia zgodności ostatniego znaku j powinno zrównać się z M . Możemy wówczas łatwo odtworzyć pozycję, od której wzorec startuje w badanym tekście: będzie to oczywiście $i-M$.

Tłumacząc powyższe sytuacje na C++, możemy łatwo dojść do następującej procedury:



txt-1.cpp

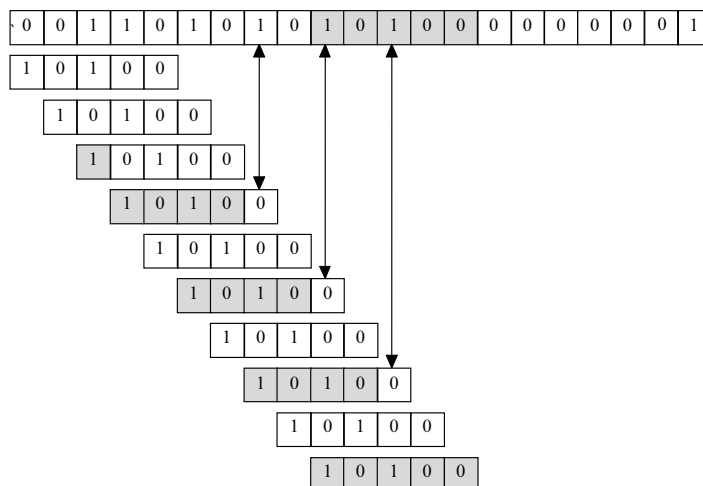
```
int szukaj(char *w, char *t)
{
    int i=0, j=0, M, N;
    M=strlen(w);           // długość wzorca
    N=strlen(t);           // długość tekstu
    while(j<M && i<N)
    {
        if(t[i]!=w[j])
        {
            // *
            i-=j-1;
            j=-1;
        }
        i++; j++;          // **
    }
    if(j==M)
        return i-M;
    else
        return -1;
}
```

Sposób korzystania z funkcji `szukaj` jest przedstawiony na przykładzie następującej funkcji `main`:

```
int main()
{
    char *b="abrakadabra", *a="rak";
    cout << szukaj(a,b) << endl;    // zwraca 2
}
```

Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorec, lub -1 w przypadku, gdy poszukiwany tekst nie został odnaleziony — jest to znana nam już doskonale konwencja. Przyjrzyjmy się dokładniej przykładowi poszukiwania wzorca 10100 w pewnym tekście binarnym (patrz rysunek 8.2).

Rysunek 8.2.
„Falszywe starty”
podczas poszukiwania



Rysunek jest nieco uproszczony: w istocie poziome przesuwanie się wzorca oznacza instrukcje zaznaczone na listingu jako (*), natomiast cała szara strefa o długości k oznacza k -krotne wykonanie (**).

Na podstawie zobrazonego przykładu możemy spróbować wymyślić taki najgorszy tekst i wzorec, dla których proces poszukiwania będzie trwał możliwie najdłużej. Chodzi oczywiście o zarówno tekst, jak i wzorec złożone z samych zer i zakończone jedynką⁴.

Spróbujmy obliczyć klasę tego algorytmu dla opisanego przed chwilą ekstremalnego najgorszego przypadku. Obliczenie nie należy do skomplikowanych czynności: zakładając, że restart algorytmu będzie konieczny $(N-1) - (M-2) = N-M+1$ razy, i wiedząc, że podczas każdego cyklu jest konieczne wykonanie M porównań, otrzymujemy natychmiast $M(N-M+1)$, czyli około⁵ $M \cdot N$.

Zaprezentowany w tym paragrafie algorytm wykorzystuje komputer jako bezmyślne, ale sprawne liczydło⁶. Jego złożoność obliczeniowa eliminuje go w praktyce z przeszukiwania tekstów binarnych, w których może wystąpić wiele niekorzystnych konfiguracji danych. Jedyną zaletą algorytmu jest jego prostota, co i tak nie czyni go na tyle atrakcyjnym, by dać się zamęczyć jego powolnym działaniem.

8.2. Nowe algorytmy poszukiwań

Algorytm, o którym będzie mowa w tym rozdziale, posiada ciekawą historię, którą w formie anegdoty warto przytoczyć. Otóż w 1970 roku S. A. Cook udowodnił teoretyczny rezultat dotyczący pewnej abstrakcyjnej maszyny. Wynikało z niego, że istniał

⁴ Zera i jedynki symbolizują tu dwa różne od siebie znaki.

⁵ Zwykle M będzie znacznie mniejsze niż N .

⁶ Termin *brute-force* jeden z moich znajomych ślicznie przetłumaczył jako „metodę mastodonta”.

algorytm poszukiwania wzorca w tekście, który działał w czasie proporcjonalnym do $M+N$ w najgorszym przypadku. Rezultat pracy Cooka wcale nie był przewidziany do praktycznych celów, niemniej D. E. Knuth i V. R. Pratt otrzymali na jego podstawie algorytm, który był łatwo implementowalny praktycznie — ukazując przy okazji, iż pomiędzy praktycznymi realizacjami a rozważaniami teoretycznymi nie istnieje wcale aż tak ogromna przepaść, jakby się to mogło wydawać. W tym samym czasie J. H. Morris odkrył dokładnie ten sam algorytm jako rozwiązanie problemu, który napotkał podczas praktycznej implementacji edytora tekstu. Algorytm *K-M-P* — bo tak będziemy go dalej zwali — jest jednym z przykładów dość częstych w nauce odkryć równoległych: z jakichś niewiadomych powodów nagle kilku pracujących osobno ludzi dochodzi do tego samego dobrego rezultatu. Prawda, że jest w tym coś niesamowitego i aż się prosi o jakieś metafizyczne hipotezy?

Knuth, Morris i Pratt opublikowali swój algorytm dopiero w 1976 roku. W międzyczasie pojawił się kolejny „cudowny” algorytm, tym razem autorstwa R. S. Boyera i J. S. Moore’a, który okazał się w pewnych zastosowaniach znacznie szybszy od algorytmu *K-M-P*. Został on również równolegle wynaleziony (odkryty?) przez R. W. Gospera. Oba te algorytmy są jednak dość trudne do zrozumienia bez pogłębionej analizy, co utrudniło ich rozpropagowanie.

W roku 1980 R. M. Karp i M. O. Rabin doszli do wniosku, że przeszukiwanie tekstów nie jest aż tak dalekie od standardowych metod przeszukiwania i wynaleźli algorytm, który — działając ciągle w czasie proporcjonalnym do $M+N$ — jest ideowo zbliżony do poznanego już przez nas algorytmu typu *brute-force*. Na dodatek, jest to algorytm, który można względnie łatwo uogólnić na przypadek poszukiwania w tablicach 2-wymiarowych, co czyni go potencjalnie użytecznym w obróbce obrazów.

W następnych trzech sekcjach szczegółowo omówimy sobie wspomniane w tym przeglądzie historycznym algorytmy.

8.2.1. Algorytm K-M-P

Wadą algorytmu *brute-force* jest jego czułość na konfigurację danych: fałszywe restarty są tu bardzo kosztowne; w analizie tekstu cofamy się o całą długość wzorca, zapominając po drodze wszystko, co przetestowaliśmy do tej pory. Narzuca się tu niejako chęć skorzystania z informacji, które już w pewien sposób posiadamy — przecież w następnym etapie będą wykonywane częściowo te same porównania, co poprzednio!

W pewnych szczególnych przypadkach przy znajomości struktury analizowanego tekstu możliwe jest ulepszenie algorytmu. Przykładowo: jeśli wiemy na pewno, iż w poszukiwanym wzorcu jego pierwszy znak nie pojawia się już w nim w ogóle⁷, to w razie restartu nie musimy cofać wskaźnika i o $j-1$ pozycji, jak to było poprzednio (patrz listing `txt-1.cpp`). W tym przypadku możemy po prostu zinkrementować i wiedząc, że ewentualne powtórzenie poszukiwań na pewno nic by już nie dało. Owszem, można się łatwo zgodzić z twierdzeniem, iż tak wyspecjalizowane teksty zdarzają się relatywnie rzadko, jednak powyższy przykład ukazuje, iż ewentualne manipulacje algo-

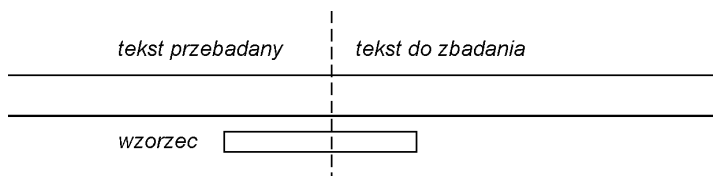
⁷ Przykład: „ABBBBBBB” — znak ‘A’ wystąpił tylko jeden raz.

rytmami poszukiwań są ciągle możliwe — wystarczy się tylko rozejrzeć. Idea algorytmu *K-M-P*. polega na wstępnym zbadaniu wzorca w celu obliczenia ilości pozycji, o które należy cofnąć wskaźnik i w przypadku stwierdzenia niezgodności badanego tekstu ze wzorcem. Oczywiście, można również rozumować w kategoriach przesuwania wzorca do przodu — rezultat będzie ten sam. To właśnie tę drugą konwencję będziemy stosować dalej. Wiemy już, że powinniśmy przesuwać się po badanym tekście nieco inteligentniej niż w poprzednim algorytmie. W przypadku zauważenia niezgodności na pewnej pozycji j wzorca⁸ należy zmodyfikować ten indeks, wykorzystując informację zawartą w już zbadanej „szarej strefie” zgodności.

Brzmi to wszystko (zapewne) niesłychanie tajemniczo, pora więc jak najszybciej wyjaśnić tę sprawę, aby uniknąć możliwych nieporozumień. Popatrzmy w tym celu na rysunek 8.3.

Rysunek 8.3.

Wyszukiwanie optymalnego przesunięcia w algorytmie K-M-P



Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz wzorec bardzo wolno w prawo, patrząc jednocześnie na już zbadany tekst — tak, aby obserwować ewentualne pokrycie się tej części wzorca, która znajduje się po lewej stronie przerywanej kreski, z tekstem, który umieszczony jest powyżej wzorca. W pewnym momencie może okazać się, że następuje pokrycie obu tych części. Zatrzymujemy wówczas przesuwanie i kontynuujemy testowanie (znak po znaku) zgodności obu części znajdujących się za kreską pionową.

Od czego zależy ewentualne pokrycie się oglądanych fragmentów tekstu i wzorca? Otóż, dość paradoksalnie badany tekst nie ma tu nic do powiedzenia — jeśli można to tak określić. Informacja o tym, jaki on był, jest ukryta w stwierdzeniu „ $j-1$ znaków było zgodnych” — w tym sensie można zupełnie o badanym tekście zapomnieć i analizując wyłącznie sam wzorec, odkryć poszukiwane optymalne przesunięcie. Na tym właśnie spostrzeżeniu opiera się idea algorytmu *K-M-P*. Okazuje się, że badając samą strukturę wzorca, można obliczyć, jak powinniśmy zmodyfikować indeks j w razie stwierdzenia niezgodności tekstu ze wzorcem na j -tej pozycji.

Zanim zagłębimy się w wyjaśnienia na temat obliczania owych przesunięć, popatrzmy na efekt ich działania na kilku kolejnych przykładach. Na rysunku 8.4 możemy dostrzec, iż na siódmej pozycji wzorca⁹ (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

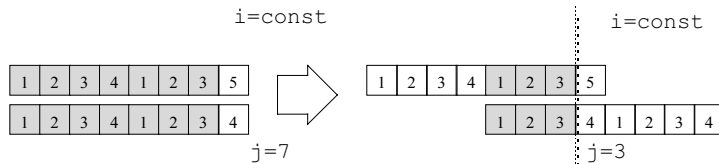
Jeśli zostawimy indeks i w spokoju, to — modyfikując wyłącznie j — możemy bez problemu kontynuować przeszukiwanie. Jakie jest optymalne przesunięcie wzorca? Ślizgając go wolno w prawo (patrz rysunek 8.4) doprowadzamy w pewnym momencie do nałożenia się ciągów 123 przed kreską — cała strefa niezgodności została wyprowadzona na prawo i ewentualne dalsze testowanie może być kontynuowane!

⁸ Lub i w przypadku badanego tekstu.

⁹ Licząc indeksy tablicy tradycyjnie od zera.

Rysunek 8.4.

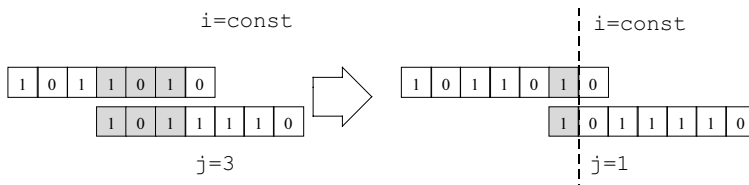
Przesuwanie się wzorca
w algorytmie K-M-P (1)



Analogiczny przykład znajduje się na rysunku 8.5.

Rysunek 8.5.

Przesuwanie się wzorca
w algorytmie K-M-P (2)



Tym razem niezgodność wystąpiła na pozycji $j=3$. Dokonując — podobnie jak poprzednio — przesuwania wzorca w prawo, zauważamy, iż jedyne możliwe nałożenie się znaków wystąpi po przesunięciu o dwie pozycje w prawo — czyli dla $j=1$. Dodatkowo okazuje się, że znaki za kreską też się pokryły, ale o tym algorytm dowie się dopiero podczas kolejnego testu zgodności na pozycji i .

Dla potrzeb algorytmu K-M-P konieczne okazuje się wprowadzenie tablicy przesunięć `int shift[M]`. Sposób jej zastosowania będzie następujący: jeśli na pozycji j wystąpiła niezgodność znaków, to kolejną wartością j będzie `shift[j]`. Nie wnikając chwilowo w sposób inicjalizacji tej tablicy (odmiennej oczywiście dla każdego wzorca), możemy natychmiast podać algorytm K-M-P, który w konstrukcji jest niemal dokładną kopią algorytmu typu *brute-force*:

**kmp.cpp**

```
int kmp(char *w, char *t)
{
    int i, j, N = strlen(t);
    for(i=0, j=0; i<N && j<M; i++, j++)
        while((j>=0) && (t[i] != w[j]))
            j = shift[j];
    if (j==M)
        return i-M;
    else
        return -1;
}
```

Szczególnym przypadkiem jest wystąpienie niezgodności na pozycji zerowej: z założenia niemożliwe jest tu przesuwanie wzorca w celu uzyskania nałożenia się znaków. Z tego powodu chcemy, aby indeks j pozostał niezmienny przy jednoczesnej progresji indeksu i . Jest to możliwe do uzyskania, jeśli umówimy się, że `shift[0]` zostanie zainicjowany wartością `-1`. Wówczas podczas kolejnej iteracji pętli `for` nastąpi inkrementacja i oraz j , co wyzeruje nam j .

Pozostaje do omówienia sposób konstrukcji tablicy `shift[M]`. Jej obliczenie powinno nastąpić przed wywołaniem funkcji `kmp`, co sugeruje, iż w przypadku wielokrotnego poszukiwania tego samego wzorca nie musimy już powtarzać inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna — jest ona praktycznie identyczna z `kmp` z tą tylko różnicą, iż algorytm sprawdza zgodność wzorca... z nim samym!

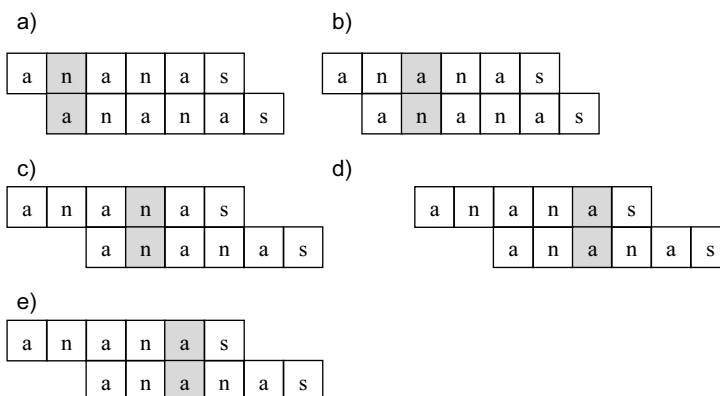
```
int shift[M];
void init_shifts(char *w)
{
    int i,j;
    shift[0]=-1;
    for(i=0,j=-1; i<M-1; i++, j++, shift[i]=j)
        while( (j>=0) && (w[i]!=w[j]) )
            j=shift[j];
}
```

Sens tego algorytmu jest następujący: tuż po inkrementacji `i` i `j` wiemy, że pierwsze `j` znaków wzorca jest zgodne ze znakami na pozycjach: `p[i-j-1]... p[i-1]` (ostatnie `j` pozycji w pierwszych `i` znakach wzorca). Ponieważ jest to największe `j` spełniające powyższy warunek, zatem, aby nie ominąć *potencjalnego* miejsca wykrycia wzorca w tekście, należy ustawić `shift[i]` na `j`.

Popatrzmy, jaki będzie efekt zadziałania funkcji `init_shifts` na słowie *ananas* (patrz rysunek 8.6). Zacieniowane litery oznaczają miejsca, w których wystąpiła niezgodność wzorca z tekstem. W każdym przypadku graficznie przedstawiono efekt przesunięcia wzorca — widać wyraźnie, które strefy pokrywają się przed strefą zacieniowaną (porównaj rysunek 8.5).

Rysunek 8.6.

Optymalne przesunięcia
wzorca „ananas”



Przypomnijmy jeszcze, że tablica `shift` zawiera nową wartość dla indeksu `j`, który przemieszcza się po wzorcu.

Algorytm *K-M-P* można zoptymalizować, jeśli znamy z góry wzorce, których będziemy poszukiwać. Przykładowo: jeśli bardzo często zdarza nam się szukać w tekstach słowa *ananas*, to w funkcję `kmp` można wbudować tablicę przesunięć:

**ananas.cpp**

```

int kmp_ananas(char *t)
{
    int i=-1;
start:
    i++;
et0: if (t[i]!='a') goto start;
    i++;
et1: if (t[i]!='n') goto et0;
    i++;
et2: if (t[i]!='a') goto et0;
    i++;
et3: if (t[i]!='n') goto et1;
    i++;
    if (t[i]!='a') goto et2;
    i++;
    if (t[i]!='s') goto et3;
    i++;
return i-6;
}

```

W celu właściwego odtworzenia etykiet należy oczywiście co najmniej raz wykonać funkcję `init_shifts` lub obliczyć samemu odpowiednie wartości. W każdym razie gra jest warta świeczki: powyższa funkcja charakteryzuje się bardzo zwięzłym kodem wynikowym asemblerowym, jest zatem bardzo szybka. Posiadacze kompilatorów, które umożliwiają generację kodu wynikowego jako tzw. „assembly output”¹⁰, mogą z łatwością sprawdzić różnice pomiędzy wersjami `kmp` i `kmp_ananas`! Dla przykładu mogę podać, że w przypadku wspomnianego kompilatora GNU klasyczna wersja procedury `kmp` (wraz z `init_shifts`) miała objętość około 170 linii kodu asemblerowego, natomiast `kmp_ananas` zmieściła się w ok. 100 liniach. (Patrz: pliki z rozszerzeniem `s` na dyskietce dla kompilatora GNU lub `asm` dla kompilatora Borland C++ 5.5).

Algorytm *K-M-P* działa w czasie proporcjonalnym do $M+N$ w najgorszym przypadku. Największy zauważalny zysk związany z jego użyciem dotyczy przypadku tekstów o wysokim stopniu samopowtarzalności — dość rzadko występujących w praktyce. Dla typowych tekstów zysk związany z wyborem metody *K-M-P* będzie zatem słabo zauważalny.

Użycie tego algorytmu jest jednak niezbędne w tych aplikacjach, w których następuje liniowe przeglądanie tekstu — bez buforowania. Jak łatwo zauważyć, wskaźnik `i` w funkcji `kmp` nigdy nie jest dekrementowany, co oznacza, że plik można przeglądać od początku do końca bez cofania się w nim. W niektórych systemach może to mieć istotne znaczenie praktyczne — przykładowo: mamy zamiar analizować bardzo długi plik tekstowy i charakter wykonywanych operacji nie pozwala na cofnięcie się w tej czynności (`i` w odczytywanym na bieżąco pliku).

¹⁰ W przypadku kompilatorów popularnej serii Borland C++ należy skompilować program ręcznie poprzez polecenie `bcc32 -S -lxxxplik.cpp`, gdzie `xxx` oznacza katalog z plikami typu `H`; identyczna opcja istnieje w kompilatorze GNU C++, należy wystukać: `c++ -Splik.cpp`.

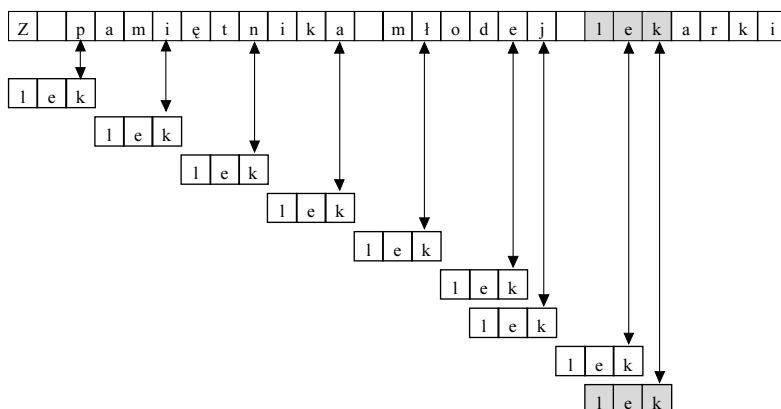
8.2.2. Algorytm Boyera i Moore'a

Kolejny algorytm, który będziemy omawiali, jest ideowo znacznie prostszy do zrozumienia niż algorytm *K-M-P*. W przeciwieństwie do metody *K-M-P* porównywaniu ulega ostatni znak wzorca. To niekonwencjonalne podejście niesie ze sobą kilka istotnych zalet:

- ♦ jeśli podczas porównywania okaże się, że rozpatrywany aktualnie znak nie wchodzi w ogóle w skład wzorca, wówczas możemy „skoczyć” w analizie tekstu o całą długość wzorca do przodu! Ciężar algorytmu przesunął się więc z analizy ewentualnych zgodności na badanie niezgodności — a te ostatnie są statystycznie znacznie częściej spotykane;
- ♦ skoki wzorca są zazwyczaj znacznie większe od 1 — porównaj z metodą *K-M-P*!

Zanim przejdziemy do szczegółowej prezentacji kodu, omówimy sobie na przykładzie jego działania. Spójrzmy w tym celu na rysunek 8.7, gdzie przedstawione jest poszukiwanie ciągu znaków „lek” w tekście „Z pamiętnika młodej lekarki”¹¹.

Rysunek 8.7.
Przeszukiwanie tekstu
metodą Boyera
i Moore'a



funkcji indeks, która zwraca w przypadku spacji liczbę zero — w pozostałych przypadkach numer litery w alfabecie. Poniższy przykład uwzględnia jedynie kilka polskich liter — Czytelnik uzupełni go z łatwością o brakujące znaki. Numer litery jest oczywiście zupełnie arbitralny i zależy od programisty. Ważne jest tylko, aby nie pominąć w tablicy żadnej litery, która może wystąpić w tekście. Jedna z możliwych wersji funkcji indeks jest przedstawiona poniżej:

```
const K=26*2+2*2+1;           // znaki ASCII+polskie litery+spacja
int shift[K];
int indeks(char c)
{
    switch(c)
    {
        case ' ':return 0;           // spacja=0
        case 'ę':return 53;
        case 'Ę':return 54;           // polskie litery
        case 'ł':return 55;
        case 'Ł':return 56;           // itd. dla pozostałych
        default:                       // polskich liter
            if(islower(c))             // 'c' jest małą literą?
                return c-'a'+1;
            else
                return c-'A'+27;
    }
}
```

Funkcja indeks ma jedynie charakter usługowy. Służy ona m.in. do właściwej inicjalizacji tablicy przesunięć. Mając za sobą analizę przykładu z rysunku 8.7, Czytelnik nie powinien być zbyt zdziwiony sposobem inicjalizacji:

```
void init_shifts(char *w)
{
    int i, M=strlen(w);
    for(i=0; i<K; i++)
        shift[i]=M;
    for(i=0; i<M; i++)
        shift[indeks(w[i])]=M-i-1;
}
```

Przejdźmy wreszcie do prezentacji samego listingu algorytmu, z przykładem wywołania:

```
int bm(char *w, char *t)
{
    init_shifts(w);
    int i, j, N=strlen(t), M=strlen(w);
    for(i=M-1, j=M-1; j>0; i--, j--)
        while(t[j]!=w[j])
        {
            int x=shift[indeks(t[i])];
            if(M-j>x)
                i+=M-j;
            else
                i+=x;
            if (i>=N)
                return -1;
            j=M-1;
        }
}
```

```
return i;
}

int main()
{
    char *t="Z pamiętnika młodej lekarki";
    cout << "Wynik poszukiwań=" << bm("lek",t) << endl;
}
```

Algorytm *Boyer'a* i *Moore'a*, podobnie jak i *K-M-P*, jest klasy $M+N$ — jednak jest on o tyle od niego lepszy, iż w przypadku krótkich wzorców i długiego alfabetu kończy się po około M/N porównaniach. W celu obliczenia optymalnych przesunień¹² autorzy algorytmu proponują skombinowanie powyższego algorytmu z tym zaproponowanym przez Knutha, Morrisa i Pratta. Celowość tego zabiegu wydaje się jednak wątpliwa, gdyż, optymalizując sam algorytm, można w bardzo łatwy sposób uczynić zbyt czasochłonnym sam proces prekompilacji wzorca.

8.2.3. Algorytm Rabina i Karpa

Ostatni algorytm do przeszukiwania tekstów, który będziemy analizowali, wymaga znajomości rozdziału 7. i terminologii, która została w nim przedstawiona. Algorytm *Rabina* i *Karpa* polega bowiem na dość przewrotnej idei:

- ♦ wzorec w (do odszukania) jest *kluczem* (patrz terminologia transformacji kluczowej w rozdziale 7.) o długości M znaków, charakteryzującym się pewną wartością wybranej przez nas funkcji H . Możemy zatem obliczyć jednokrotnie $H_w=H(w)$ i korzystać z tego wyliczenia w sposób ciągły.
- ♦ tekst wejściowy t (do przeszukania) może być w taki sposób odczytywany, aby na bieżąco znać M ostatnich znaków¹³. Z tych M znaków wyliczamy na bieżąco $H_t=H(t)$.

Gdy założymy jednoznaczność wybranej funkcji H , sprawdzenie zgodności wzorca z aktualnie badanym fragmentem tekstu sprowadza się do odpowiedzi na pytanie: czy H_w jest równe H_t ? Spostrzegawczy Czytelnik ma jednak prawo pokręcić w tym miejscu z powątpiewaniem głową: przecież to nie ma prawa działać szybko! Istotnie, pomysł wyliczenia dodatkowo funkcji H dla każdego słowa wejściowego o długości M wydaje się tak samo kosztowny — jak nie bardziej! — jak zwykle sprawdzanie tekstu znak po znaku (patrz algorytm *brute-force*, §8.1). Tym bardziej że jak do tej pory nie powiedzieliśmy ani słowa na temat funkcji H ! Z poprzedniego rozdziału pamiętamy zapewne, iż jej wybór wcale nie był taki oczywisty.

Omawiany algorytm jednak istnieje i na dodatek działa szybko! Zatem, aby to wszystko, co poprzednio zostało napisane, logicznie się ze sobą łączyło, potrzebny nam będzie zapewne jakiś trik. Sztuka polega na właściwym wyborze funkcji H . Robin i Karp wybrali

¹² Rozważ np. wielokrotne występowanie takich samych liter we wzorcu.

¹³ Na samym początku będzie to oczywiście M pierwszych znaków tekstu.

taką funkcję, która dzięki swym szczególnym właściwościom umożliwia dynamiczne wykorzystywanie wyników obliczeń dokonanych krok wcześniej, co znacząco potrafi uprościć obliczenia wykonywane w kroku bieżącym.

Założmy, że ciąg M znaków będziemy interpretować jako pewną liczbę całkowitą. Przyjmując za b — jako podstawę systemu — ilość wszystkich możliwych znaków, otrzymamy:

$$x = t[i]b^{M-1} + t[i+1]b^{M-2} + \dots + t[i+M-1].$$

Przesuńmy się teraz w tekście o jedną pozycję do przodu i zobaczmy, jak zmieni się wartość x :

$$x' = t[i+1]b^{M-1} + t[i+2]b^{M-2} + \dots + t[i+M].$$

Jeśli dobrze przyjrzymy się x i x' , to okaże się, że wartość x' jest w dużej części zbudowana z elementów tworzących x — pomnożonych przez b z uwagi na przesunięcie. Nietrudno jest wówczas wywnioskować, że:

$$x' = (x - t[i]b^{M-1}) + t[i+M].$$

Jako funkcji H użyjemy dobrze nam znanej z poprzedniego rozdziału $H(x) = x \% p$, gdzie p jest dużą liczbą pierwszą. Założmy, że dla danej pozycji i wartość $H(x)$ jest nam znana. Po przesunięciu się w tekście o jedną pozycję w prawo pojawia się konieczność wyliczenia wartości funkcji $H(x')$ dla tego „nowego” słowa. Czy istotnie zachodzi potrzeba powtarzania całego wyliczenia? Być może istnieje pewne ułatwienie bazujące na zależności, jaka istnieje pomiędzy x i x' ?

Na pomoc przychodzi nam tu własność funkcji modulo użytej w wyrażeniu arytmetycznym. Można oczywiście obliczyć modulo z wyniku końcowego, lecz to bywa czasami niewygodne z uwagi na przykład wielkość liczby, z którą mamy do czynienia — a poza tym, gdzie tu byłby zysk szybkości?! Jednak identyczny wynik otrzymuje się, aplikując funkcję modulo po każdej operacji cząstkowej i przenosząc otrzymaną wartość do następnego wyrażenia cząstkowego! Dla przykładu weźmy obliczenie:

$$(5 * 100 + 6 * 10 + 8) \% 7 = 568 \% 7 = 1.$$

Wynik ten jest oczywiście prawdziwy, co można łatwo sprawdzić z kalkulatorem. Identyczny rezultat da nam jednak następująca sekwencja obliczeń:

$$\begin{array}{ccccc} & \downarrow & & \downarrow & \\ 5 * 100 \% 7 = 3 & & (3 + 6 * 10) \% 7 = 0 & & (0 + 8) \% 7 = 1 \end{array}$$

co jest też łatwe do weryfikacji.

Implementacja algorytmu jest prosta, lecz zawiera kilka instrukcji wartych omówienia. Popatrzmy na listing:

**rk.cpp**

```
int rk(char w[],char t[])
{
    unsigned long i,bM_1=1,Hw=0,Ht=0,M,N;
    M=strlen(w),N=strlen(t);
    for(i=0;i<M;i++)
    {
        Hw=(Hw*b+indeks(w[i]))%p; // inicjacja funkcji H dla wzorca
        Ht=(Ht*b+indeks(t[i]))%p; // inicjacja funkcji H dla tekstu
    }
    for(i=1;i<M;i++) bM_1=(b*bM_1)%p;
    for(i=0;Hw!=Ht;i++) // przesuwanie się w tekście
    {
        Ht=(Ht+b*p-indeks(t[i])*bM_1)%p; // (*)
        Ht=(Ht*b+indeks(t[i+M]))%p;
        if (i>N-M)
            return -1; // porażka poszukiwań
    }
    return i;
}
```

Na pierwszym etapie następuje wyliczenie początkowych wartości H_t i H_w . Ponieważ ciągi znaków trzeba interpretować jako liczby, konieczne będzie zastosowanie znanej już nam doskonale funkcji `indeks` (patrz strona 204.). Wartość H_w jest niezmienna i nie wymaga uaktualniania. Nie dotyczy to jednak aktualnie badanego fragmentu tekstu — tutaj wartość H_t ulega zmianie podczas każdej inkrementacji zmiennej i . Do obliczenia $H(x')$ możemy wykorzystać omówioną wcześniej własność funkcji modulo — co jest dokonywane w trzeciej pętli `for`. Dodatkowego wyjaśnienia wymaga być może linia oznaczona (*). Otóż dodawanie wartości $b*p$ do H_t pozwala nam uniknąć przypadkowego wskoczenia w liczby ujemne. Gdyby istotnie tak się stało, przeniesiona do następnego wyrażenia arytmetycznego wartość modulo byłaby nieprawidłowa i sfałszowałaby końcowy wynik!

Kolejne uwagi należą się parametrom p i b . Zaleca się, aby p było dużą liczbą pierwszą¹⁴, jednakże nie można tu przesadzać z uwagi na możliwe przekroczenie zakresu pojemności użytych zmiennych. W przypadku wyboru dużego p zmniejszamy prawdopodobieństwo wystąpienia kolizji spowodowanej niejednoznacznością funkcji H . Ta możliwość — mimo iż mało prawdopodobna — ciągle istnieje i ostrożny programista powinien wykonać dodatkowy test zgodności w i $t[i] \dots t[i+M-1]$ po zwróceniu przez funkcję `rk` pewnego indeksu i .

Co zaś się tyczy wyboru podstawy systemu (oznaczonej w programie jako b), to warto jest wybrać liczbę nawet nieco za dużą, zawsze jednak będącą potęgą liczby 2. Możliwe jest wówczas zaimplementowanie operacji mnożenia przez b jako przesunięcia bitowego — wykonywanego przez komputer znacznie szybciej niż zwykłe mnożenie. Przykładowo: dla $b=64$ możemy zapisać mnożenie $b*p$ jako $p<<6$.

Gwoli formalności jeszcze można dodać, że gdy nie występuje kolizja (typowy przypadek!), algorytm *Robina* i *Karpa* wykonuje się w czasie proporcjonalnym do $M+N$.

¹⁴ W naszym przypadku jest to liczba 33554393.