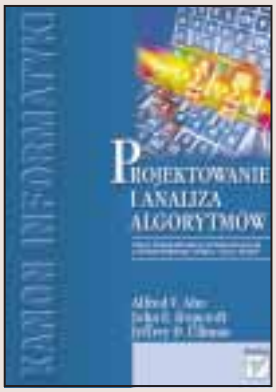
Projektowanie i analiza algorytmów



Autorzy: Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
Tłumaczenie: Wojciech Derechowski
ISBN: 83-7197-770-0
Tytuł oryginału: The Design and Analysis of Computer Algorithms
Format: B5, stron: 488

Badanie algorytmów leży w samym sercu nauk komputerowych. W ostatnich latach dokonano znaczących postępów w tej dziedzinie. Opracowano m.in. wiele efektywniejszych algorytmów (szybkie przekształcenie Fouriera), odkryto także istnienie pewnych naturalnych zadań, dla których wszystkie algorytmy są nieefektywne. Wyniki te powodują wzrost zainteresowania badaniami algorytmów, co przyczynia się do intensywnego rozwoju tej dziedziny wiedzy.

Książka jest podręcznikiem wstępnego kursu projektowania i analizy algorytmów. Autorzy położyli nacisk raczej na prezentacji najważniejszych idei i przystępności wykładu, niż na szczegółach realizacji i sztuczkach programistycznych. Autorzy przedstawiają na ogół nieformalne, intuicyjne objaśnienia zamiast długich i pracochłonnych dowodów. Książka nie wymaga żadnego szczególnego przygotowania z zakresu matematyki, czy języków programowania. Pożądana jest jednak pewna dojrzałość w stosowaniu pojęć matematycznych, ogólne obycie w językach programowania wysokiego poziomu, takich jak FORTRAN lub ALGOL, a także podstawowa znajomość algebry liniowej.

W książce omówiono m.in.:

- Podstawowe pojęcia i modele (w tym maszynę Turniga)
- Najważniejsze struktury danych, rekurencję, programowanie dynamiczne
- Algorytmy sortowania, operacje na zbiorach, drzewach i grafach
- Szybkie przekształcenie Fouriera z zastosowaniami
- Algorytmy arytmetyczne, operacje na wielomianach
- Algorytmy dopasowania wzorców
- Problemy NP-zupełne
- Dolne ograniczenia złożoności obliczeniowej

Ważnym uzupełnieniem treści książki są ćwiczenia o zróżnicowanych poziomach trudności. „Projektowanie i analiza algorytmów” to doskonały podręcznik dla studentów informatyki i kierunków pokrewnych, a także wspaniała pomoc dla osób prowadzących wykłady i ćwiczenia na tych kierunkach.

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

Spis treści

Rozdział 1.

Modele obliczania

Jak, mając dany problem, znajdziemy efektywny algorytm rozwiązania? Gdy znaleźliśmy algorytm, jak mamy porównać ten algorytm z innymi algorytmami, które rozwiązują ten sam problem? Jak powinniśmy oceniać jakość algorytmu? Pytania tego rodzaju są ciekawe zarówno dla programisty, jak i dla uczonego o teoretycznym nastawieniu do nauk komputerowych. W książce rozpatrujemy różne kierunki badań, które usiłują odpowiedzieć na takie pytania.

W tym rozdziale rozważamy kilka modeli komputera — maszynę o dostępie swobodnym, maszynę z zapamiętanym programem i maszynę Turinga. Porównujemy je co do tego, jak odzwierciedają złożoność algorytmu i wyprowadzamy z nich kilka wyspecjalizowanych modeli obliczeń: liniowe programy arytmetyczne, obliczenia na bitach, obliczenia na wektorach bitów i drzewa decyzji. Wreszcie, w ostatnim punkcie rozdziału wprowadzamy język do opisu algorytmów, zwany „Pidgin ALGOL".

1.1. Algorytmy i ich złożoność

Algorytmy mogą być oceniane na podstawie rozmaitych kryteriów. Najczęściej interesuje nas szybkość z jaką wzrastają czas lub pamięć potrzebne, by rozwiązać zadanie w coraz bardziej wymagających przypadkach. Zawsze będziemy przypisywać zadaniu liczbę całkowitą, zwaną *rozmiarem* zadania, która jest miarą wielkości danych. Na przykład rozmiarem zadania w przypadku mnożenia macierzy może być największy wymiar macierzy, które mamy pomnożyć. Rozmiarem zadania z grafem może być liczba krawędzi grafu.

Wymagany przez algorytm czas wyrażony jako funkcja rozmiaru zadania zwany jest *złożonością czasową* algorytmu. Zachowanie się tej złożoności w granicy, gdy rozmiar zadania wzrasta, nazywa się *asymptotyczną złożonością czasową*. Podobnie można zdefiniować *złożoność pamięciową* i *asymptotyczną złożoność pamięciową*.

Asymptotyczna złożoność algorytmu jest tym, co ostatecznie rozstrzyga o rozmiarze zadań, które mogą być rozwiązane przez ten algorytm. Jeżeli algorytm przetwarza dane o rozmiarze n w czasie cn^2 dla pewnej stałej c, to mówimy, że czasowa złożoność tego algorytmu jest $O(n^2)$, czytaj „rzędu n^2". Ściślej, funkcja $g(n)$ jest

Algorytm	Złożoność czasowa	Maksymalny rozmiar zadania		
		1 sek.	1 min.	1 godz.
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Rys. 1.1. Ograniczenia rozmiaru zadania spowodowane szybkością wzrostu złożoności

$O(f(n))$, jeżeli istnieje stała c taka, że $g(n) \leqslant cf(n)$ dla wszystkich nieujemnych wartości n prócz pewnego skończonego (być może pustego) zbioru tych wartości.

Można by przypuszczać, że ogromny wzrost szybkości obliczeń dzięki powstaniu maszyn cyfrowych obecnej generacji zmniejszy znaczenie efektywnych algorytmów. Jest jednak odwrotnie. Skoro komputery stają się szybsze i możemy przetwarzać coraz większe zadania, to o wzroście rozmiaru zadania, jaki można osiągnąć przez wzrost szybkości komputera, rozstrzyga złożoność algorytmu.

Załóżmy, że mamy pięć algorytmów $A_1 - A_5$ o podanych złożonościach czasowych:

Algorytm	Złożoność czasowa
A_1	n
A_2	$n \log n$ ([1])
A_3	n^2
A_4	n^3
A_5	2^n

Złożoność czasowa jest tu liczbą jednostek czasu potrzebnych do przetworzenia danych rozmiaru n. Zakładając, że jednostka czasu jest równa jednej milisekundzie, algorytm A_1 może przetworzyć w ciągu jednej sekundy dane o rozmiarze 1000, natomiast algorytm A_5 dane o rozmiarze co najwyżej 9. Rysunek 1.1 podaje rozmiary zadań, które mogą być rozwiązane przez każdy z tych pięciu algorytmów w ciągu jednej sekundy, jednej minuty i jednej godziny.

Przypuśćmy, że następna generacja komputerów będzie dziesięć razy szybsza niż obecna. Rysunek 1.2 pokazuje wzrost rozmiaru zadania, jakie można rozwiązać dzięki temu wzrostowi prędkości. Zauważmy, że z algorytmem A_5 dziesięciokrotny wzrost prędkości zwiększa tylko o trzy rozmiar zadania, które można rozwiązać, natomiast z algorytmem A_3 ten rozmiar wzrasta więcej niż trzykrotnie.

Zamiast wzrostu szybkości rozważmy skutek użycia bardziej efektywnego algorytmu. Popatrzmy raz jeszcze na rys. 1.1. Biorąc jedną minutę za podstawę porów-

[1] O ile nie zaznaczono inaczej, wszystkie logarytmy w tej książce mają podstawę 2.

Algorytm	Złożoność czasowa	Maksymalny rozmiar zadania przed przyspieszeniem	Maksymalny rozmiar zadania po przyspieszeniu
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	około $10s_2$ dla dużych s_2
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

Rys. 1.2. Skutek dziesięciokrotnego przyspieszenia

nania, można przez zastąpienie algorytmu A_4 algorytmem A_3 rozwiązać zadanie sześciokrotnie większe, a przez zastąpienie algorytmu A_4 algorytmem A_2, zadanie 125 razy większe. Wyniki te są znacznie bardziej przekonujące niż dwukrotna poprawa osiągnięta przez dziesięciokrotny wzrost szybkości. Jeżeli za podstawę porównania weźniemy godzinę, rożnice są jeszcze bardziej istotne. Wnioskujemy, że asymptotyczna złożoność algorytmu jest ważną miarą jakości algorytmu, miarą, która stanie się jeszcze ważniejsza w przyszłości, gdy szybkość obliczeń wzrośnie.

Mimo uwagi, którą poświęcamy temu, jak rośnie rząd wielkości, powinniśmy zdawać sobie sprawę, że algorytm o gwałtownym tempie wzrostu może mieć mniejszą stałą proporcjonalności niż algorytm o niższym. W takim przypadku szybko rosnący algorytm może być lepszy dla małych zadań, a może nawet dla wszystkich zadań, które mają rozmiar, jaki nas interesuje. Przypuśćmy na przykład, że złożonościami czasowymi algorytmów A_1, A_2, A_3, A_4 i A_5 są $1000n$, $100n \log n$, $10n^2$, n^3 i 2^n. Wtedy A_5 będzie najlepszy dla zadań o rozmiarze $2 \leqslant n \leqslant 9$, A_3 dla $10 \leqslant n \leqslant 58$, A_2 dla $59 \leqslant n \leqslant 1024$, a A_1 dla zadań o rozmiarze większym niż 1024.

Nim w rozważaniu algorytmów i ich złożoności pójdziemy dalej, musimy opisać model maszyny liczącej, która je wykonuje i określić, co rozumiemy przez krok w obliczeniach. Niestety nie istnieje model obliczeń, który pasowałby do wszystkich sytuacji. Jedną z głównych trudności jest długość słów maszynowych. Jeżeli na przykład założy się, że w słowie maszynowym można umieścić liczbę całkowitą dowolnej wielkości, całe zadanie można zakodować w postaci jednej liczby całkowitej w jednym słowie. Jeżeli założy się, że słowo maszynowe jest skończone, trzeba rozważyć trudność zapamiętania dowolnie dużych liczb i inne problemy pomijane, gdy zadania mają umiarkowany rozmiar. Dla problemu musimy wybrać model, który będzie odzwierciedlać czas obliczeń w rzeczywistym komputerze.

W następnych punktach tego rozdziału omówimy kilka podstawowych modeli maszyn liczących, przede wszystkim maszynę o dostępie swobodnym, maszynę o dostępie swobodnym z zapamiętanym programem i maszynę Turinga. Te trzy modele są równoważne pod względem mocy obliczeniowej, lecz nie szybkości.

Formalne modele obliczeń wzięły się głównie z pragnienia, by wydobyć na jaw istotną trudność obliczeniową różnych problemów. Chcemy podać dowody dolnych ograniczeń czasu obliczeń. Aby wykazać, że nie istnieje algorytm, który wykonuje dane zadanie w czasie krótszym niż pewien czas, potrzebujemy ścisłej i w wielu punktach bardzo sztywnej definicji tego, czym jest algorytm. Przykład takiej definicji stanowią maszyny Turinga (p. 1.6.).

W opisach i objaśnieniach algorytmów przyda się nam zapis prostszy i bardziej jasny niż program dla maszyny o dostępie swobodnym, maszyna z zapamiętanym programem, czy maszyna Turinga. Z tego powodu wprowadzimy język wysokiego poziomu, zwany Pidgin ALGOL. W całej książce opisujemy algorytmy w tym języku. Ale żeby rozumieć złożoność obliczeniową algorytmu opisanego przez Pidgin ALGOL, musimy pokazać, jak Pidgin ALGOL zależy od modeli bardziej formalnych. Zrobimy to w ostatnim punkcie tego rozdziału.

1.2. Maszyny o dostępie swobodnym

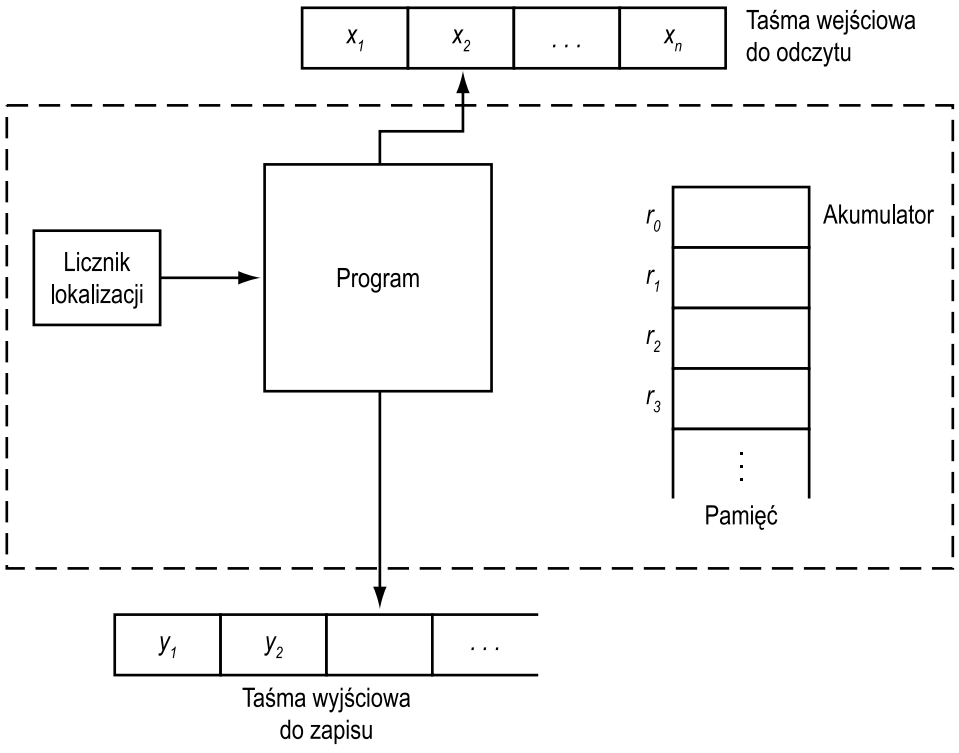
Maszyna (RAM, od *random access machine*) jest modelem komputera o jednym akumulatorze i instrukcjach, którym nie wolno się modyfikować.

Maszyna RAM składa się z taśmy wejściowej tylko do czytania, taśmy wyjściowej tylko do pisania, programu oraz pamięci (rys. 1.3). Taśma wejściowa jest ciągiem klatek, z których każda zawiera liczbę całkowitą (być może ujemną). Ilekroć z taśmy wejściowej czytany jest symbol, głowica taśmy wejściowej przesuwa się o jedną klatkę w prawo. Wyjściem jest taśma tylko do pisania, podzielona na klatki, które początkowo są puste. Gdy wykonywana jest instrukcja pisania, w klatce znajdującej się na taśmie wyjściowej pod głowicą taśmy wyjściowej drukowana jest liczba całkowita i głowica taśmy wyjściowej przesuwana jest na prawo. Gdy symbol wyjściowy zostanie zapisany, nie można go zmienić.

Pamięć składa się z ciągu rejestrów $r_0, r_1, \ldots, r_i, \ldots$, z których każdy może przechowywać liczbę całkowitą dowolnej wielkości. Na liczbę rejestrów, które mogą być użyte, nie nakładamy żadnego ograniczenia górnego. Abstrakcja tego rodzaju jest poprawna, w przypadkach gdy:

1. rozmiar zadania jest na tyle mały, że mieści się ono w pamięci komputera, oraz
2. liczby całkowite, użyte do obliczeń, są na tyle małe, że mieszczą się w pojedynczych słowach maszynowych.

Program dla maszyny RAM nie jest przechowywany w pamięci. A więc zakładamy, że program ten nie modyfikuje sam siebie. Program jest jedynie ciągiem instrukcji z (nieobowiązkowymi) etykietami. Ścisłe określenie instrukcji używanych w programie nie jest zbyt ważne, dopóki są podobne do instrukcji spotykanych w rzeczywistych komputerach. Zakładamy instrukcje arytmetyczne, instrukcje wejścia-wyjścia, instrukcje adresowania pośredniego (przykładowo w indeksowaniu do ta-



Rys. 1.3. Maszyna o dostępie swobodnym

blic) i instrukcje rozgałęzienia (*branching*).[2] Wszelkie obliczenia wykonywane są w rejestrze r_0, zwanym *akumulatorem*, który, jak wszystkie pozostałe rejestry pamięci, może pomieścić dowolną liczbę całkowitą. Przykład zbioru instrukcji dla maszyny RAM przedstawia rysunek 1.4. Każda instrukcja składa się z dwóch części — *kodu operacji* i *adresu*.

W zasadzie możemy uzupełnić ten zbiór o dowolne inne, znane z rzeczywistych komputerów instrukcje, takie jak operacje logiczne czy operacje na znakach, nie zmieniając przy tym rzędu złożoności zadań. Czytelnik wedle swego uznania może uważać zbiór instrukcji za uzupełniony w ten sposób. Operandum może mieć postać:

1. $=i$, co oznacza samą liczbę całkowitą i,
2. nieujemnej liczby całkowitej i, co oznacza zawartość rejestru i,
3. $*i$, co oznacza adresowanie pośrednie. A mianowicie, operandum jest zawartością rejestru j, gdzie j jest liczbą całkowitą, która znajduje się w rejestrze i. Jeżeli $j < 0$, to maszyna ulega zatrzymaniu.

[2] Oprócz instrukcji warunkowych (*Jump on Greater than Zero, Jump on Zero*, jak czytam JGTZ i JZERO), repertuar zawiera JUMP; por. rys. 1.5 (str. 17) — *przyp. tłum.*

Kod operacji		Adres
1.	LOAD	operandum
2.	STORE	operandum
3.	ADD	operandum
4.	SUB	operandum
5.	MULT	operandum
6.	DIV	operandum
7.	READ	operandum
8.	WRITE	operandum
9.	JUMP	etykieta
10.	JGTZ	etykieta
11.	JZERO	etykieta
12.	HALT	

Rys. 1.4. Tablica instrukcji RAM

Instrukcje te powinny być dobrze znane każdemu, kto programował asembler. Możemy teraz zdefiniować sens programu P za pomocą dwóch wielkości: przekształcenia c określonego na zbiorze nieujemnych liczb całkowitych o wartościach w zbiorze liczb całkowitych i „licznika lokalizacji", który ustala następną instrukcję do wykonania. Funkcja c jest *mapą pamięci*; $c(i)$ jest to liczba całkowita umieszczona w rejestrze i (*zawartość* rejestru i).

Początkowo $c(i) = 0$ dla każdego $i \geqslant 0$, licznik lokalizacji jest nastawiony na pierwszą instrukcję P, a taśma wyjściowa jest pusta. Po wykonaniu k-tej instrukcji P licznik lokalizacji jest automatycznie nastawiany na $k+1$ (tj. na następną instrukcję), chyba że k-tą instrukcją jest JUMP, HALT, JGTZ lub JZERO.

Aby określić sens instrukcji, definiujemy $v(a)$, *wartość operandum* a następująco:

$$\begin{aligned} v(=i) &= i, \\ v(i) &= c(i), \\ v(*i) &= c(c(i)). \end{aligned}$$

Tabela na rysuku 1.5 definiuje sens każdej instrukcji z rysunku 1.4. Instrukcje niezdefiniowane, takie jak STORE $=i$, można uważać za równoważne HALT. Podobnie zatrzymuje maszynę dzielenie przez zero.

Podczas wykonywania każdej z pierwszych ośmiu instrukcji licznik lokalizacji jest zwiększany o 1. Instrukcje są wykonywane w porządku, w którym występują w programie, aż do napotkania instrukcji JUMP, HALT, JGTZ przy zawartości akumulatora większej od zera, lub JZERO, przy zawartości akumulatora równej zero.

Ogólnie program RAM definiuje przekształcenie taśm wejściowych w taśmy wyjściowe. Skoro nie dla wszystkich taśm wejściowych program może się zatrzymać, przekształcenie jest częściowe (czyli może być nieokreślone dla pewnych danych

	Instrukcja	Sens
1.	LOAD a	$c(0) \leftarrow v(a)$
2.	STORE i	$c(i) \leftarrow c(0)$
	STORE $*i$	$c(c(i)) \leftarrow c(0)$
3.	ADD a	$c(0) \leftarrow c(0) + v(a)$
4.	SUB a	$c(0) \leftarrow c(0) - v(a)$
5.	MULT a	$c(0) \leftarrow c(0) \times v(a)$
6.	DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$ ([3])
7.	READ i	$c(i) \leftarrow$ bieżący symbol na wejściu.
	READ $*i$	$c(c(i)) \leftarrow$ bieżący symbol na wejściu. Głowica taśmy wejściowej przesuwa się o jedną klatkę w prawo w obu przypadkach.
8.	WRITE a	$v(a)$ jest drukowane w klatce, która na taśmie wyjściowej jest obecnie pod głowicą. Następnie głowica taśmy wyjściowej przesuwana jest o jedną klatkę w prawo.
9.	JUMP b	Licznik lokalizacji jest nastawiany na instrukcję z etykietą b.
10.	JGTZ b	Licznik lokalizacji jest nastawiany na instrukcję z etykietą b, jeżeli $c(0) > 0$; w przeciwnym razie licznik lokalizacji jest nastawiany na następną instrukcję.
11.	JZERO b	Licznik lokalizacji jest nastawiany na instrukcję z etykietą b, jeżeli $c(0) = 0$; w przeciwnym razie licznik lokalizacji jest nastawiany na następną instrukcję.
12.	HALT	Wykonanie ustaje.

[3] W tej książce $\lceil x \rceil$ (*ceiling* x) oznacza najmniejszą liczbę całkowitą, większą lub równą x, zaś $\lfloor x \rfloor$ (*floor* lub *część całkowita* x) oznacza największą liczbę całkowitą, mniejszą lub równą x.

Rys. 1.5. Sens instrukcji RAM. Operandum a jest tu $= i, i$, lub $*i$

wejściowych). Przekształcenie to można interpretować na różne sposoby. Dwiema istotnymi interpretacjami są funkcja, bądź język.

Przypuśćmy, że program P zawsze czyta n liczb całkowitych z taśmy wejściowej i pisze co najwyżej jedną liczbę całkowitą na taśmie wyjściowej. Jeżeli $x_1, x_2, \ldots, x_n$ są liczbami całkowitymi w pierwszych n klatkach taśmy wejściowej a P zapisuje y w pierwszej klatce taśmy wyjściowej i zatrzymuje się, to mówimy, że P oblicza funkcję $f(x_1, x_2, \ldots, x_n) = y$. Łatwo udowodnić, że RAM, jak każdy inny realistyczny model komputera, oblicza jedynie *funkcje częściowo rekurencyjne*. Otóż dla każdej częściowo rekurencyjnej funkcji f możemy zdefiniować program RAM, który oblicza f, i dla każdego programu RAM, równoważną funkcję częściowo rekurencyjną (patrz Davis [1958] lub Rogers [1967] odnośnie funkcji rekurencyjnych).

Program RAM można interpretować także jako akceptor języka. *Alfabetem* jest skończony zbiór symboli, a *językiem* zbiór napisów nad pewnym alfabetem. Symbole alfabetu mogą być reprezentowane przez liczby całkowite $1, 2, \ldots, k$ dla pewnego k. Maszyna RAM może akceptować język w następujący sposób. Umieszczamy

```
begin
    read r1;
    if r1 ⩽ 0 then write 0
    else
        begin
            r2 ← r1;
            r3 ← r1 − 1;
            while r3 > 0 do
                begin
                    r2 ← r2 ∗ r1;
                    r3 ← r3 − 1
                end;
            write r2
        end
end
```

Rys. 1.6. Program dla n^n w Pidgin ALGOLu

napis wejściowy $s = a_1 a_2 \cdots a_n$ na taśmie wejściowej: symbol a_1 w pierwszej klatce, symbol a_2 w drugiej, itd. Symbol 0, którego użyjemy jako znacznika końca, umieszczamy w klatce $(n+1)$, by oznaczyć koniec napisu wejściowego.

Napis wejściowy s jest *akceptowany* przez program P maszyny RAM, jeżeli P czyta cały napis s i znacznik końca, pisze 1 w pierwszej klatce taśmy wyjściowej i zatrzymuje się. *Język akceptowany przez* P jest zbiorem akceptowanych napisów wejściowych. Dla napisów wejściowych, które nie należą do języka akceptowanego przez P, P może drukować na taśmie wyjściowej symbol inny niż 1 i zatrzymywać się albo nawet nie zatrzymywać się. Łatwo udowodnić, że język jest akceptowany przez program RAM wtedy i tylko wtedy, gdy jest *rekurencyjnie przeliczalny*. Język jest akceptowany przez zatrzymującą się dla wszystkich danych maszynę RAM wtedy i tylko wtedy, gdy jest językiem *rekurencyjnym* (odnośnie języków rekurencyjnych i rekurencyjnie przeliczalnych, patrz Hopcroft i Ullman [1969]).

Rozważmy dwa przykłady programów RAM. Pierwszy definiuje funkcję, drugi akceptuje język.

Przykład 1.1. Rozważmy funkcję $f(n)$ daną wzorem:

$$f(n) = \begin{cases} n^n, & \text{gdy liczba całkowita } n \geqslant 1, \\ 0 & \text{w przeciwnym razie.} \end{cases}$$

Napisany w języku Pidgin ALGOL program, który oblicza $f(n)$ mnożąc n samo przez siebie $(n-1)$ razy, podaje rys. 1.6.[4] Odpowiedni program RAM to rys. 1.7. Zmienne r_1, r_2 i r_3 leżą w rejestrach 1, 2 i 3. Nie robimy pewnych oczywistych usprawnień, więc odpowiedniość między rysunkami 1.6 i 1.7 będzie jasna. □

[4] Patrz punkt 1.8. w sprawie opisu języka Pidgin ALGOL.

Program RAM			Odpowiednie instrukcje Pidgin ALGOLu
	READ	1	**read** $r1$
	LOAD	1	
	JGTZ	pos	**if** $r1 \leqslant 0$ **then write** 0
	WRITE	=0	
	JUMP	endif	
pos:	LOAD	1	$r2 \leftarrow r1$
	STORE	2	
	LOAD	1	
	SUB	=1	$r3 \leftarrow r1 - 1$
	STORE	3	
while:	LOAD	3	
	JGTZ	continue	**while** $r3 > 0$ **do**
	JUMP	endwhile	
continue:	LOAD	2	
	MULT	1	$r2 \leftarrow r2 * r1$
	STORE	2	
	LOAD	3	
	SUB	=1	$r3 \leftarrow r3 - 1$
	STORE	3	
	JUMP	while	
endwhile:	WRITE	2	**write** $r2$
endif:	HALT		

Rys. 1.7. Program RAM dla n^n

```
begin
    d ← 0;
    read x;
    while x ≠ 0 do
        begin
            if x ≠ 1 then d ← d − 1 else d ← d + 1;
            read x
        end;
    if d = 0 then write 1
end
```

Rys. 1.8. Rozpoznawanie napisów z równą liczbą jedynek i dwójek

Przykład 1.2. Rozważmy program RAM, który akceptuje złożony ze wszystkich napisów o tej samej liczbie jedynek i dwójek język nad alfabetem wejściowym $\{1, 2\}$. Program ten wczytuje każdy symbol wejściowy do rejestru 1, a w rejestrze 2 utrzy-

Program RAM			Odpowiednie instrukcje Pidgin ALGOLu
	LOAD	=0	$d \leftarrow 0$
	STORE	2	
	READ	1	**read** x
while:	LOAD	1	**while** $x \neq 0$ **do**
	JZERO	endwhile	
	LOAD	1	**if** $x \neq 1$
	SUB	=1	
	JZERO	one	
	LOAD	2	**then** $d \leftarrow d - 1$
	SUB	=1	
	STORE	2	
	JUMP	endif	
one:	LOAD	2	**else** $d \leftarrow d + 1$
	ADD	=1	
	STORE	2	
endif:	READ	1	**read** x
	JUMP	while	
endwhile:	LOAD	2	**if** $d = 0$ **then write** 1
	JZERO	output	
	HALT		
output:	WRITE	=1	
	HALT		

Rys. 1.9. Program RAM odpowiadający algorytmowi z rysunku 1.8

muje różnicę d pomiędzy liczbą jedynek i dwójek widzianych dotychczas. Po napotkaniu znacznika końca 0 sprawdza, czy różnica d jest równa zero i jeżeli tak jest, drukuje 1 i zatrzymuje się. Zakładamy, że 0, 1 i 2 są wszystkimi możliwymi symbolami wejściowymi.

Program z rysunku 1.8 zawiera istotne szczegóły tego algorytmu. Równoważny program RAM podaje rys. 1.9; x leży w rejestrze 1, a d w rejestrze 2. □

1.3. Złożoność obliczeniowa programów RAM

Dwie ważne miary algorytmu to jego złożoność czasowa i pamięciowa w funkcji rozmiaru danych. Jeżeli za złożoność, dla pewnego rozmiaru danch, wziąć złożoność maksymalną dla wszystkich danych tego rozmiaru, to złożoność tę nazywa się *złożonością najgorszego przypadku*. Jeżeli za złożoność wziąć „średnią" złożoność dla wszystkich danych pewnego rozmiaru, tę złożoność nazywana się *złożonością oczekiwaną*. Złożoność oczekiwana algorytmu jest zwykle trudniejsza do oszacowania

niż złożoność najgorszego przypadku. Konieczne jest jakieś założenie o rozkładzie danych, a założenia zgodne z rzeczywistością na ogół nie są łatwe (*tractable*) matematycznie. Położymy nacisk na złożoność najgorszego przypadku, ponieważ jest łatwiejsza do potraktowania i ma uniwersalne zastosowanie. Jednakże należy pamiętać, że algorytm o najlepszej złożoności najgorszego przypadku niekoniecznie musi mieć najlepszą złożoność oczekiwaną.

Złożoność czasowa najgorszego przypadku (bądź po prostu *złożoność czasowa*) programu RAM jest funkcją $f(n)$, która dla wszystkich danych rozmiaru n jest maksimum sumy opisującej „czas" zużywany przez każdą wykonywaną instrukcję. *Oczekiwana złożoność czasowa* jest średnią dla wszystkich danych rozmiaru n tej samej sumy. Odnośnie pamięci definiujemy podobne terminy, gdy za „«czas» zużywany przez każdą wykonywaną instrukcję" podstawiamy „«pamięć» zużywaną przez każdy wykorzystywany rejestr".

Aby ściśle określić złożoność czasową i pamięciową, musimy określić czas wymagany dla wykonania każdej instrukcji RAM i pamięć zajmowaną przez każdy rejestr. Rozważymy dwa takie kryteria kosztu dla programów RAM. Według *kryterium kosztu zuniformizowanego* każda instrukcja RAM wymaga jednej jednostki czasu, a każdy rejestr, jednej jednostki pamięci. O ile nie zaznaczymy inaczej, złożoność programu RAM będzie mierzona według kryterium kosztu zuniformizowanego.

Druga definicja, niejednokroć bardziej realistyczna, uwzględnia skończoną długość rzeczywistego słowa pamięciowego i nazywana jest *kryterium kosztu logarytmicznego*. Niech $l(i)$ będzie następującą *funkcją logarytmiczną* dla liczb całkowitych:

$$l(i) = \begin{cases} \lfloor \log |\, i \,| \rfloor + 1, & i \neq 0 \\ 1, & i = 0 \end{cases}$$

Tabela na rysunku 1.10 przedstawia koszt logarytmiczny $t(a)$ dla trzech możliwych postaci operandum a. Rysunek 1.11 przedstawia czas wymagany przez każdą z instrukcji.

W tym koszcie uwzględniony jest fakt, że reprezentacja liczby całkowitej n w rejestrze wymaga $\lfloor \log n \rfloor + 1$ bitów. Rejestry, jak pamiętamy, mogą zawierać dowolnie duże liczby całkowite.

Kryterium kosztu logarytmicznego opiera się na grubym założeniu, że koszt wykonania instrukcji jest proporcjonalny do długości operandów tych instrukcji. Rozważmy na przykład koszt instrukcji ADD $*i$. Po pierwsze musimy ustalić koszt

Operandum a	Koszt $t(a)$
$=i$	$l(i)$
i	$l(i) + l(c(i))$
$*i$	$l(i) + l(c(i)) + l(c(c(i)))$

Rys. 1.10. Logarytmiczny koszt operandum

	Instrukcja	Koszt
1.	LOAD a	$t(a)$
2.	STORE i	$l(c(0)) + l(i)$
	STORE $*i$	$l(c(0)) + l(i) + l(c(i))$
3.	ADD a	$l(c(0)) + t(a)$
4.	SUB a	$l(c(0)) + t(a)$
5.	MULT a	$l(c(0)) + t(a)$
6.	DIV a	$l(c(0)) + t(a)$
7.	READ i	$l(\text{input}) + l(i)$
	READ $*i$	$l(\text{input}) + l(i) + l(c(i))$
8.	WRITE a	$t(a)$
9.	JUMP b	1
10.	JGTZ b	$l(c(0))$
11.	JZERO b	$l(c(0))$
12.	HALT	1

Rys. 1.11. Logarytmiczny koszt instrukcji RAM, gdzie $t(a)$ jest kosztem operandum a, zaś b oznacza etykietę

dekodowania operandum reprezentowanego przez adres. Aby rozpoznać liczbę całkowitą i, trzeba czasu $l(i)$. Następnie, aby odczytać $c(i)$, zawartość rejestru i, oraz odszukać rejestr $c(i)$ potrzeba czasu $l(c(i))$. Wreszcie, czytanie zawartości rejestru $c(i)$ kosztuje $l(c(c(i)))$. Skoro instrukcja ADD $*i$ dodaje liczbę całkowitą $c(c(i))$ do $c(0)$, liczby całkowitej w akumulatorze, widzimy, że realistycznym kosztem, jaki należy przypisać instrukcji ADD $*i$, jest $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$.

Logarytmiczną złożoność pamięciową programu RAM definiujemy jako sumę $l(x_i)$ po wszystkich rejestrach z akumulatorem włącznie, gdzie x_i jest liczbą całkowitą o największej wielkości, umieszczoną w rejestrze i w dowolnej chwili obliczeń.

Jest rzeczą jasną, że dany program może mieć całkowicie różne złożoności czasowe zależnie od tego, czy użyje się kosztu zuniformizowanego, czy logarytmicznego. Jeżeli założenie, że każdą liczbę napotkaną w czasie obliczeń można umieścić w jednym słowie maszynowym, jest realistyczne, to właściwa jest funkcja kosztu zuniformizowanego. W przeciwnym razie dla realistycznej analizy złożoności bardziej właściwy może być koszt logarytmiczny.

Obliczmy złożoność czasową i pamięciową programu RAM, który wylicza wartości n^n w przykładzie 1.1. Złożoność czasowa tego programu jest zdominowana przez pętlę z instrukcją MULT. Za i-tym razem, gdy wykonywana jest istrukcja MULT, akumulator zawiera n^i, a rejestr 2 zawiera n. Wszystkich wykonywanych instrukcji MULT jest $n-1$. Zgodnie z kryterium kosztu zuniformizowanego każda z instrukcji MULT kosztuje jedną jednostkę czasu, stąd na wykonanie wszystkich instrukcji MULT zużywany jest czas $O(n)$. Zgodnie z kryterium kosztu logarytmicznego kosz-

tem wykonania i-tej instrukcji MULT jest $l(n^i) + l(n) \simeq (i+1)\log n$ i wobec tego kosztem wszystkich instrukcji MULT jest:

$$\sum_{i=1}^{n-1}(i+1)\log n,$$

który jest $O(n^2 \log n)$.

Złożoność pamięciową dyktują liczby całkowite, umieszczone w rejestrach od 0 do 3. Zgodnie z kosztem zuniformizowanym złożoność pamięciowa jest po prostu $O(1)$. Zgodnie z kosztem logarytmicznym złożoność pamięciowa jest $O(n \log n)$, gdyż największą liczbą całkowitą umieszczoną w dowolnym z rejestrów jest n^n, a $l(n^n) \simeq n \log n$. Wobec tego dla programu z przykładu 1.1 mamy następujące złożoności:

	Koszt zuniformizowany	Koszt logarytmiczny
Złożoność czasowa	$O(n)$	$O(n^2 \log n)$
Złożoność pamięciowa	$O(1)$	$O(n \log n)$

Koszt zuniformizowany jest dla tego programu realistyczny tylko wtedy, gdy pojedyncze słowo maszynowe może pomieścić liczbę całkowitą tak dużą, jak n^n. Jeżeli liczba n^n jest większa od tego, co można pomieścić w jednym słowie maszynowym, to nawet logarytmiczna złożoność czasowa jest nieco nierealistyczna, gdyż zakłada, że dwie liczby całkowite, i oraz j, mogą być pomnożone przez siebie w czasie $O(l(i)) + l(j))$, a nie wiadomo dotychczas, czy tak jest.

Dla programu RAM z przykładu 1.2, przy założeniu, że n jest długością napisu wejściowego, złożoności czasowe i pamięciowe są następujące:

	Koszt zuniformizowany	Koszt logarytmiczny
Złożoność czasowa	$O(n)$	$O(n \log n)$
Złożoność pamięciowa	$O(1)$	$O(\log n)$

Jeżeli n jest większe od tego, co można pomieścić w jednym słowie maszynowym, to koszt logarytmiczny dla tego programu jest dość realistyczny.

1.4. Model z zapamiętanym programem

Ponieważ program RAM nie jest przechowywany w pamięci maszyny, nie może modyfikować sam siebie. Teraz rozważymy inny model komputera, tzw. maszynę *o dostępie swobodnym z zapamiętanym programem* (RASP, od *random access stored program*), która jest podobna do maszyny RAM z tym, że program jest w pamięci i może modyfikować sam siebie.

Instrukcja		Kodowanie	Instrukcja		Kodowanie
LOAD	i	1	DIV	i	10
LOAD	$=i$	2	DIV	$=i$	11
STORE	i	3	READ	i	12
ADD	i	4	WRITE	i	13
ADD	$=i$	5	WRITE	$=i$	14
SUB	i	6	JUMP	i	15
SUB	$=i$	7	JGTZ	i	16
MULT	i	8	JZERO	i	17
MULT	$=i$	9	HALT		18

Rys. 1.12. Kody dla instrukcji RASP

Zbiór instrukcji RASP jest identyczny ze zbiorem instrukcji RAM prócz tego, że adresowanie pośrednie nie jest dozwolone, gdyż nie jest potrzebne. Jak zobaczymy, RASP może symulować adresowanie pośrednie przez modyfikacje instrukcji w czasie wykonania programu.

Ogólna struktura maszyny RASP jest także podobna do struktury RAM, ale zakłada się, że program RASP leży w rejestrach pamięci. Każda instrukcja RASP zajmuje dwa kolejne rejestry. Pierwszy z nich zawiera kod operacji; drugi — adres. Jeżeli adres jest w postaci $=i$, to pierwszy rejestr będzie kodować także fakt, że operandum jest literałem, a drugi rejestr będzie zawierać i. Do kodowania instrukcji służą liczby całkowite. Rysunek 1.12 pokazuje jeden z możliwych sposobów kodowania. Na przykład instrukcja LOAD=32 zostanie zapamiętana za pomocą 2 w jednym rejestrze i 32 w następnym.

Podobnie jak w przypadku RAM, stan RASP może być reprezentowany przez:

1. mapę pamięci c, gdzie $c(i)$ dla $i \geqslant 0$ jest zawartością rejestru i, oraz
2. licznik lokalizacji, wskazujący na pierwszy z dwóch kolejnych rejestrów pamięci, z których ma być pobrana bieżąca instrukcja.

Licznik lokalizacji jest nastawiony początkowo na pewien zadany rejestr. Początkowa zawartość rejestrów pamięci to z reguły nie wszędzie 0, gdyż na początku do pamięci pobierany jest program. Na początku jednak wszystkie prócz skończonej liczby rejestrów pamięci i akumulator muszą zawierać 0 Po wykonaniu każdej instrukcji licznik lokalizacji jest zwiększany o 2, z wyjątkiem przypadków JUMP i, JGTZ i (gdy akumulator jest dodatni), lub JZERO i (gdy akumulator zawiera 0), w których licznik lokalizacji jest nastawiany na i. Skutek każdej z instrukcji jest taki sam jak odpowiedniej instrukcji RAM.

Złożoność czasową programu RASP można zdefiniować bardzo podobnie, jak złożoność czasową programu RAM. Możemy użyć bądź kryterium kosztu zuniformizowanego, bądź logarytmicznego. Kosztem w tym ostatnim przypadku musimy jednak obciążyć nie tylko operandum, lecz także dostęp do samej instrukcji. Kosztem tego

dostępu jest $l(\mathrm{LC})$, gdzie LC oznacza zawartość licznika lokalizacji. Na przykład kosztem wykonania instrukcji ADD $=i$, umieszczonej w rejestrach j oraz $j+1$, jest $l(j)+l(c(0))+l(i)$[5]. Kosztem instrukcji ADD i, umieszczonej w rejestrach j oraz $j+1$, jest $l(j)+l(c(0))+l(i)+l(c(i))$.

Ciekawe jest pytanie, co różni złożoność programu RAM i odpowiedniego programu RASP. Odpowiedź nie jest zaskakująca. Dowolne przekształcenie wejścia na wyjście, które może być wykonane w czasie $T(n)$ przez jeden model, może być wykonane przez drugi w czasie $kT(n)$ dla pewnej stałej k, bez względu na to, czy weźmie się koszt zuniformizowany, czy logarytmiczny. Podobnie pamięć wykorzystywana przez te modele różni się tylko o stały czynnik przy obu miarach kosztu.

Dwa twierdzenia wyrażają te zależności w sposób formalny. Obydwu dowodzi się, pokazując algorytmy, na mocy których RAM może symulować RASP i odwrotnie.

> **Twierdzenie 1.1.** Jeżeli koszt instrukcji jest zuniformizowany lub logarytmiczny, to istnieje taka stała k, że dla każdego programu RAM o złożoności czasowej $T(n)$ istnieje równoważny program RASP o złożoności czasowej $kT(n)$.

Dowód. Pokazujemy, jak symulować program RAM P przez program RASP. Rejestr 1 RASP będzie służyć do tymczasowego przechowywania zawartości akumulatora RAM. Z programu P skonstruujemy program RASP P_S, który będzie zajmować następne $r-1$ rejestrów RASP. Stała r jest zdeterminowana przez program RAM P. Zawartość rejestru i RAM, $i \geqslant 1$, będzie przechowywana w rejestrze $r+i$ RASP, więc w programie RASP wszystkie odniesienia do pamięci mają adresy o r większe od odpowiednich odniesień w programie RAM.

Każda instrukcja RAM w P, niewymagająca adresowania pośredniego, jest kodowana bezpośrednio w postaci identycznej instrukcji RASP (z odpowiednio zwiększonymi adresami odniesień do pamięci). Każda instrukcja RAM w P, wymagająca adresowania pośredniego, jest przekształcana w sekwencję sześciu instrukcji RASP, która symuluje adresowanie pośrednie przez modyfikację instrukcji.

Aby objaśnić symulację adresowania pośredniego powinien wystarczyć przykład. By symulować instrukcję RAM SUB $*i$, gdzie i jest liczbą całkowitą dodatnią, tworzymy sekwencję instrukcji RASP, która:

1. umieszcza tymczasowo zawartość akumulatora w rejestrze 1,
2. pobiera zawartość rejestru $r+i$ do akumulatora (rejestr $r+i$ RASP odpowiada rejestrowi i RAM),
3. dodaje r do akumulatora,
4. umieszcza liczbę obliczoną w kroku 3. w polu adresu instrukcji SUB,
5. przywraca zawartość akumulatora z tymczasowego rejestru 1, i wreszcie
6. używa instrukcji SUB stworzonej w kroku 4., by wykonać odejmowanie.

[5] Można by doliczyć koszt czytania rejestru $j+1$, ale ten koszt nie może różnić się bardzo od $l(j)$. W tym rozdziale mamy na uwadze nie czynniki stałe, lecz raczej szybkość wzrostu funkcji. Zatem $l(j)+l(j+1)$ jest „w przybliżeniu" $l(j)$ z dokładnością co najwyżej do czynnika 3.

Rejestr	Zawartość	Sens		
100	3	STORE	1	
101	1			
102	1	LOAD	$r+i$	
103	$r+i$			
104	5	ADD	$=r$	
105	r			
106	3	STORE	111	
107	111			
108	1	LOAD	1	
109	1			
110	6	SUB	b	gdzie b jest zawartością rejestru i RAM
111	$-$			

Rys. 1.13. Symulacja SUB $*i$ przez RASP

Rejestr RASP	Instrukcja		Koszt
j	STORE	1	$l(j)+l(1)+l(c(0))$
$j+2$	LOAD	$r+1$	$l(j+2)+l(r+i)+l(c(i))$
$j+4$	ADD	$=r$	$l(j+4)+l(c(i))+l(r)$
$j+6$	STORE	$j+11$	$l(j+6)+l(j+11)+l(c(i)+r)$
$j+8$	LOAD	1	$l(j+8)+l(1)+l(c(0))$
$j+10$	SUB	$-$	$l(j+10)+l(c(i)+r)+l(c(0))$ $+l(c(c(i)))$

Rys. 1.14. Koszt instrukcji RASP

Na przykład, stosując kodowanie instrukcji RASP podane na rysunku 1.12, i zakładając, że sekwencja instrukcji RASP zaczyna się w rejestrze 100, możemy symulować SUB $*i$ za pomocą sekwencji pokazanej na rysunku 1.13. Przesunięcie r można określić, gdy znana jest liczba instrukcji w programie RASP P_S.

Stwierdzamy, że każda instrukcja RAM wymaga co najwyżej sześciu instrukcji RASP, zatem według kryterium kosztu zuniformizowanego złożonością czasową programu RASP jest co najwyżej $6T(n)$. (Zauważmy, że miara ta jest niezależna od sposobu, w jaki określa się „wielkość" danych.)

Według kryterium kosztu logarytmicznego stwierdzamy, że każda instrukcja RAM I należąca do P jest symulowana przez sekwencję S jednej lub sześciu instrukcji RASP w P_S. Możemy pokazać, iż istnieje taka stała k zależna od P, że koszt instrukcji należących do S jest nie większy niż k razy koszt instrukcji I.

Na przykład instrukcja RAM SUB $*i$ ma koszt:

$$M = l(c(0)) + l(i) + l(c(i)) + l(c(c(i))).$$

Sekwencja S, która symuluje tę instrukcję RAM, jest pokazana na rysunku 1.14. $c(0)$, $c(i)$, oraz $c(c(i))$ na rysunku 1.14 odnoszą się do zawartości rejestrów RAM. Ponieważ P_S zajmuje rejestry RASP od 2 do r, mamy $j \leqslant r-11$. Ponadto $l(x+y) \leqslant l(x) + l(y)$, więc koszt S jest na pewno mniejszy niż:

$$2l(1) + 4M + 11l(r) < (6 + 11l(r))M.$$

Wobec tego wnioskujemy, że istnieje stała $k = 6 + 11l(r)$ taka, że jeżeli P ma złożoność czasową $T(n)$, to P_S ma złożoność czasową co najwyżej $kT(n)$. □

Twierdzenie 1.2. Jeżeli koszt instrukcji jest zuniformizowany lub logarytmiczny, to istnieje taka stała k, że dla każdego programu RASP o złożoności czasowej $T(n)$ istnieje równoważny program RAM o złożoności czasowej co najwyżej $kT(n)$.

Dowód. Program RAM, który skonstruujemy, by symulować RASP, będzie używać adresowania pośredniego, żeby dekodować i symulować instrukcje RASP umieszczone w pamięci RAM. Pewne rejestry RAM będą mieć specjalne przeznaczenie:

rejestr 1 — używany w adresowaniu pośrednim,
rejestr 2 — licznik lokalizacji RASP,
rejestr 3 — pamięć do przechowywania akumulatora RASP.

Rejestr i RASP będzie umieszczony w rejestrze $i + 3$ RAM dla $i \geqslant 1$.

RAM rozpoczyna pracę z programem RASP o skończonej długości, który jest umieszczony w pamięci, poczynając od rejestru 4. Rejestr 2 — licznik lokalizacji, zawiera 4; rejestry 1 i 3 zawierają 0. Program RAM tworzy pętla symulacji, która zaczyna się od przeczytania (za pomocą instrukcji RAM LOAD $*2$) instrukcji RASP, dekodowania tej instrukcji i rozgałęzienia do jednego z 18 zestawów instrukcji, z których każdy służy do obsługi jednego typu instrukcji RASP. W razie niepoprawnego kodu operacji, RAM, jak i RASP zatrzymają się.

Operacje dekodowania i rozgałęzienia są jasne; jako model może służyć przykład 1.2 (chociaż tam dekodowany symbol był czytany z wejścia, a tu jest czytany z pamięci). Podamy przykład instrukcji RAM, które symulują instrukcję 6 RASP, tj. SUB i. Program ten, pokazany na rysunku 1.15, ulega wywołaniu, gdy $c(c(2)) = 6$, a więc gdy licznik lokalizacji wskazuje na rejestr, który zawiera 6, czyli kod SUB.

Pomijamy dalsze szczegóły budowy programu RAM. Jako ćwiczenie pozostawiamy dowód faktu, że według kryterium kosztu zuniformizowanego lub logarytmicznego, złożoność czasowa programu RAM jest co najwyżej pewną stałą w iloczynie ze złożonością czasową RASP. □

Z twierdzeń 1.1 i 1.2 wynika, że gdy chodzi o złożoność czasową (a także pamięciową, co pozostawiamy jako ćwiczenie) modele RAM i RASP są równoważne z dokładnością do czynnika stałego, tj. rząd ich złożoności jest ten sam dla tego samego algorytmu. Spośród tych dwóch modeli na ogół wykorzystujemy w książce model RAM, gdyż jest on nieco prostszy.

LOAD 2 ADD =1 STORE 2		Zwiększ licznik lokalizacji o 1, tak aby wskazywał na rejestr, który zawiera operandum i instrukcji SUB i.
LOAD *2 ADD =3 STORE 1		Pobierz i do akumulatora, dodaj 3, wynik umieść w rejestrze 1.
LOAD 3 SUB *1 STORE 3		Pobierz zawartość akumulatora RASP z rejestru 3. Odejmij zawartość rejestru $i+3$, wynik umieść z powrotem w rejestrze 3.
LOAD 2 ADD =1 STORE 2		Zwiększ licznik lokalizacji znów o 1, tak by wskazywał teraz na następną instrukcję RASP.
JUMP	a	Powróć na początek pętli symulacji (nazwany tutaj „a").

Rys. 1.15. Symulacja SUB i przez RAM

1.5. Abstrakcje RAM

W wielu sytuacjach nie są potrzebne tak skomplikowane modele obliczeń jak RAM i RASP. Wobec tego liczne modele definiuje się przez abstrakcję pewnych własności RAM, zaniedbując inne. Uzasadnieniem dla takich modeli jest fakt, że zaniedbywane instrukcje stanowią co najwyżej stały ułamek kosztu każdego efektywnego algorytu, rozwiązującego problemy, do których model jest stosowany.

i. Program liniowy

Pierwszym rozważanym przez nas modelem jest liniowy program (*stright-line program*). W wielu problemach wystarczy skupić uwagę na klasie programów RAM, gdzie instrukcje rozgałęzienia są używame tylko do powtarzania jakiejś sekwencji instrukcji pewną ilość razy, proporcjonalną do n — rozmiaru danych. W tym przypadku dla każdego rozmiaru n można program „rozwinąć", powielając odpowiednią ilość razy instrukcje, które mają być powtarzane. Daje to sekwencję liniowych (wolnych od pętli) i zapewne coraz dłuższych programów, po jednym dla każdego n.

Przykład 1.3. Rozważmy mnożenie dwóch macierzy wymiaru $n \times n$ o elementach ze zbioru liczb całkowitych. Zwykle można oczekiwać nie bez racji, że liczba powtórzeń pętli w programie RAM będzie niezależna od wielkosci elementów macierzy. Warto więc założyć dla uproszczenia, że dozwolone są tylko pętle z instrukcjami testu, w których wchodzi w grę wyłącznie n, rozmiar zadania. Oczywisty algorytm mnożenia macierzy zawiera pętle, które muszą być na przykład wykonane dokładnie n razy, gdyż wymaga instrukcji rozgałęzienia, które porównują indeks z n. □

Dzięki rozwinięciu programu do postaci liniowej obywamy się bez instrukcji rozgałęzienia. Uzasadnienie czerpiemy stąd, że w wielu zadaniach nie więcej niż stały

ułamek kosztu programu RAM jest przeznaczony na instrukcje rozgałęzienia, sterujące pętlami. Podobnie często możemy założyć, że instrukcje wejścia tworzą tylko stały ułamek kosztu programu i wykluczyć je, zakładając, że skończony zbiór wejść, wymagany przy pewnym n, znajduje się w pamięci, gdy program rozpoczyna pracę. Skutki adresowania pośredniego można oszacować przy ustalonym n, o ile rejestry, służące do adresowania pośredniego, zawierają wartości zależne tylko od n, a nie od wartości zmiennych wejściowych. Wobec tego zakładamy, że nasze programy liniowe są pozbawione adresowania pośredniego.

Ponadto skoro każdy z programów liniowych może zawierać odniesienia tylko do skończonej liczby rejestrów pamięci, wygodnie jest nazwać rejestry wykorzystywane przez program. Rejestry podlegają wobec tego raczej odnosieniom przez *adresy symboliczne* (symbole lub napisy złożone z liter), niż przez liczby całkowite.

Z repertuaru RAM po usunięciu wymagań co do READ, JUMP, JGTZ i JZERO pozostają nam LOAD, STORE, WRITE, HALT i operacje arytmetyczne. Nie potrzebujemy HALT, gdyż koniec programu musi oznaczać zatrzymanie. Możemy obyć się bez WRITE, wyróżniając pewne adresy symboliczne jako *zmienne wyjściowe*; informacją wyjścia programu są wartości tych zmiennych w chwili zakończenia.

Możemy wreszcie włączyć LOAD i STORE do operacji arytmetycznych, zastępując sekwencje, takie jak:

$$\begin{array}{ll} \text{LOAD} & a \\ \text{ADD} & b \\ \text{STORE} & c \end{array}$$

przez $c \leftarrow a+b$. Cały repertuar instrukcji programu liniowego jest więc następujący:

$$\begin{array}{l} x \leftarrow y + z \\ x \leftarrow y - z \\ z \leftarrow y * z \\ z \leftarrow y/z \\ x \leftarrow i \end{array}$$

gdzie x, y i z są adresami symbolicznymi (czyli *zmiennymi*), a i jest stałą. Łatwo zauważyć, że dowolna sekwencja LOAD, STORE i operacji arytmetycznych na akumulatorze może być zastąpiona pewną sekwencją pięciu powyższych instrukcji.

Programowi liniowemu są przyporządkowane dwa wyróżnione zbiory zmiennych: jego *wejścia* i *wyjścia*. Funkcja obliczana przez program liniowy jest zbiorem wartości zmiennych wyjściowych (w zadanym porządku), wyrażanych względem wartości zmiennych wejściowych.

Przykład 1.4. Rozważmy obliczanie wielomianu:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Zmiennymi wejściowymi są współczynniki $a_0, a_1, \ldots, a_n$ i symbol x. Zmienną wyjściową jest p. Według *reguły Hornera* $p(x)$ obliczamy jako:

$n=1$	$n=2$	$n=3$
$t \leftarrow a_1 * x$	$t \leftarrow a_2 * x$	$t \leftarrow a_3 * x$
$p \leftarrow t + a_0$	$t \leftarrow t + a_1$	$t \leftarrow t + a_2$
	$t \leftarrow t * x$	$t \leftarrow t * x$
	$p \leftarrow t + a_0$	$t \leftarrow t + a_1$
		$t \leftarrow t * x$
		$p \leftarrow t + a_0$

Rys. 1.16. Programy liniowe, odpowiadające regule Hornera

1. $a_1 x + a_0$ dla $n = 1$,
2. $(a_2 x + a_1)x + a_0$ dla $n = 2$,
3. $((a_3 x + a_2)x + a_1)x + a_0$ dla $n = 3$.

Wyrażeniom tym odpowiadają programy liniowe z rysunku. 1.16. Reguła Hornera dla dowolnego n powinna być jasna. Dla każdego n mamy program liniowy o $2n$ krokach, który oblicza wielomian n-tego stopnia. W rozdziale 12. pokażemy, że aby obliczyć wartość wielomianu n-tego stopnia, gdy współczynniki są dane jako wejście, konieczne jest n mnożeń i n dodawań. Reguła Hornera jest optymalna według modelu programu liniowego. □

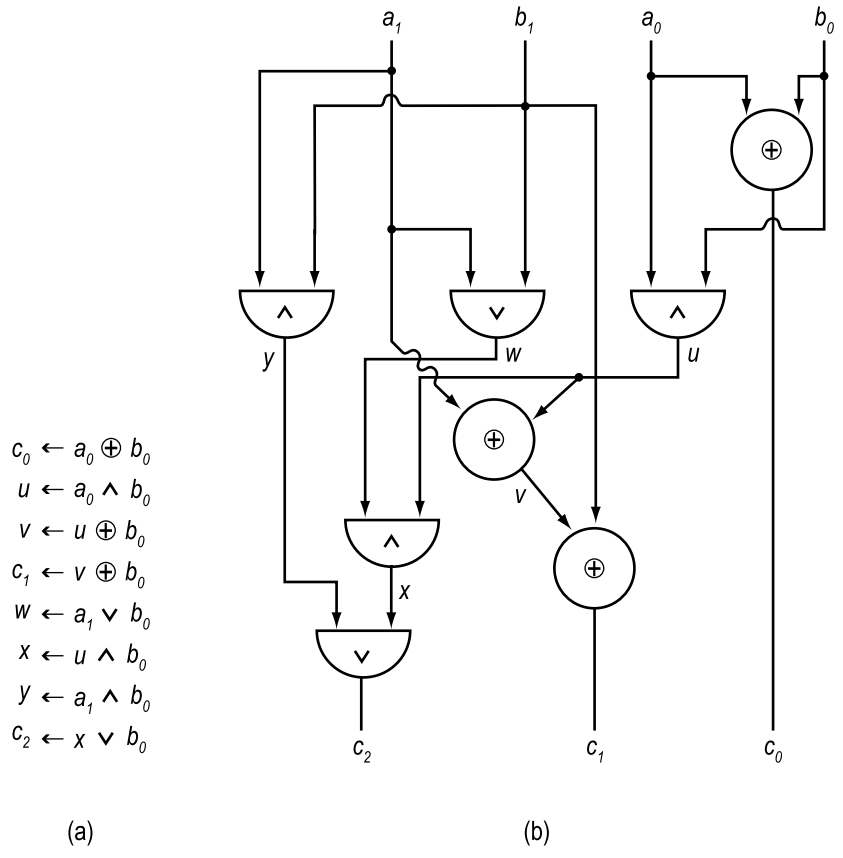
Według modelu programu liniowego obliczeń złożonością czasową ciągu programów jest liczba kroków n-tego programu jako funkcja n. Reguła Hornera na przykład daje ciąg o złożoności czasowej $2n$. Zauważmy, że mierzenie złożoności czasowej to tyle, co mierzenie liczby operacji arytmetycznych. Złożonością pamięciową ciągu programów jest liczba wymienionych zmiennych także jako funkcja n. Programy z przykładu 1.4 mają złożoność pamięciową $n + 4$.

> **Definicja.** Gdy chodzi o model programu liniowego, mówimy, że problem ma złożoność czasową lub pamięciową $O_{\mathrm{A}}(f(n))$, jeżeli istnieje ciąg programów, którego złożoność czasowa lub pamięciowa sięga co najwyżej $cf(n)$ dla pewnej stałej c. (Zapis $O_{\mathrm{A}}(f(n))$ oznacza „rząd $f(n)$ kroków, gdy modelem jest programu liniowy". Wskaźnik A oznacza „arytmetyczny", co jest główną cechą kodu liniowego.) Obliczanie wartości wielomianu ma złożoność czasową $O_{\mathrm{A}}(n)$, jak i pamięciową $O_{\mathrm{A}}(n)$.

ii. Obliczenia na bitach

Model programu liniowego opiera się oczywiście na funkcji kosztu zuniformizowanego. Jak wspomnieliśmy, koszt ten jest właściwy, gdy wszystkie obliczane wielkości są „rozsądne". Istnieje prosta modyfikacja modelu programu liniowego, która jest odbiciem funkcji kosztu logarytmicznego. Model ten, nazywamy przez nas *obliczaniami na bitach*, jest zasadniczo taki sam jak kod liniowy za wyjątkiem tego, że:

1. zakładamy, że wszystkie zmienne mają wartość 0 lub 1, tj. są bitami.



Rys. 1.17. (a)Program dodawania na bitach, (b) równoważny układ logiczny

2. używamy operacji logicznych, a nie arytmetycznych.[6] Piszemy $\wedge$ dla **i**, $\vee$ dla **lub**, $\oplus$ dla **rozłącznego lub** i $\neg$ dla **nie**.

Zgodnie z modelem bitowym operacje arytmetyczne na liczbach całkowitych i i j wymagają przynajmniej $l(i)+l(j)$ kroków, co jest odbiciem logarytmicznego kosztu operandów. Faktycznie, mnożenie i dzielenie według najlepszych znanych algorytmów wymaga wiecej niż $l(i) + l(j)$ kroków, by pomnożyć lub podzielić i przez j.

Na oznaczenie rzędu wielkości w modelu obliczeń na bitach stosujemy O_{B}. Model bitowy przydaje się, gdy chcemy mówić o podstawowych operacjach, jak operacje arytmetyczne, które są pierwotne w innych modelach. Na przykład w modelu programu liniowego mnożenie dwóch n-bitowych liczb całkowitych jest do wykonania w $O_{\mathrm{A}}(1)$ kroku, natomiast w modelu bitowym najlepszy znany wynik to $O_{\mathrm{B}}(n \log n \log\log n)$ kroków.

[6]Stąd zbiór instrukcji RAM musi zawierać te operacje.

Innym zastosowaniem modelu bitowego są układy logiczne. Programy liniowe z bitowymi wejściami i operacjami odpowiadają wzajemnie jednoznacznie logiczno-kombinatorycznym układom do obliczania układów funkcji boolowskich. Liczba kroków programu jest liczbą elemetów logicznych układu.

Przykład 1.5. Rysunek 1.17(a) przedstawia program dodawania dwóch dwubitowych liczb $[\,a_1a_0\,]$ i $[\,b_1b_0\,]$. Zmiennymi wyjściowymi są c_2, c_1 i c_0, takie że $[\,a_1a_0\,] + [\,b_1b_0\,] = [\,c_2c_1c_0\,]$. Program liniowy z rysunku 1.17(a) oblicza:

$$\begin{aligned} c_0 &= a_0 \oplus b_0, \\ c_1 &= ((a_0 \wedge b_0) \oplus a_1) \oplus b_1, \\ c_2 &= ((a_0 \wedge b_0) \wedge (a_1 \vee b_1)) \vee (a_1 \wedge b_1). \end{aligned}$$

Rys. 1.17(b) przedstawia odpowiedni układ logiczny. Dowód, że dodawanie dwu n-bitowych liczb można wykonać w $O_{\mathrm{B}}(n)$ krokach zostawiamy jako ćwiczenie. □

iii. Operacje na wektorach bitowych

Zamiast ograniczać wartość zmienej do 0 lub 1, można pójść w przeciwnym kierunku i pozwolić, by zmienne przybierały jako wartość dowolny wektor bitów. Faktycznie, wektory bitów o danej długości odpowiadają w oczywisty sposób liczbom całkowitym, więc nie wykraczamy istotnie poza model RAM, tj. w razie potrzeby wciąż zakładamy nieograniczoną wielkość rejestrów.

Jednakże, jak zobaczymy w tych kilku algorytmach, w których stosowany jest model z wektorami bitów, długość używanych wektorów znacznie przewyższa liczbę bitów potrzebnych do przedstawienia wielkości zadania. Wielkość liczb całkowitych używanych w algorytmie będzie na ogół tego samego rzędu co wielkość zadania. Na przykład, rozwiązując problemy dróg w grafie o 100 wierzchołkach, można by zastosować wektory bitów o długości 100 do wskazywania, czy istnieje droga z danego wierzchołka v do każdego z wierzchołków grafu; tzn. w wektorze dla wierzchołka v na i-tej pozycji jest 1 wtedy i tylko wtedy, gdy istnieje droga z v do v_i. W przypadku tego samego problemu można używać także liczb całkowitych (przykładowo do liczenia i indeksowania) i będą one mieć wielkość zapewne rzędu 100. Stąd dla liczb całkowitych będzie potrzebne 7 bitów, podczas gdy dla wektorów 100.

Różnica nie musi być jednak aż tak znaczna, ponieważ większość komputerów wykonuje operacje logiczne na wektorach bitów o długości pełnego słowa w cyklu jednej instrukcji. Zatem wektory bitów o długości 100 mogą podlegać manipulacjom w trzech lub czterech krokach, w porównaniu z jednym krokiem dla liczb całkowitych. Niemniej wyniki na temat czasowej i pamięciowej złożoności algorytmów dla modelu z wektorami bitów należy brać *cum grano salis*, gdyż wielkość zadania, przy której model ten staje się nierealistyczny jest znacznie mniejsza, niż dla modelu RAM i modelu kodu liniowego. Na oznaczenie rzędu wielkości w modelu z wektorami bitowymi stosujemy O_{BV}.

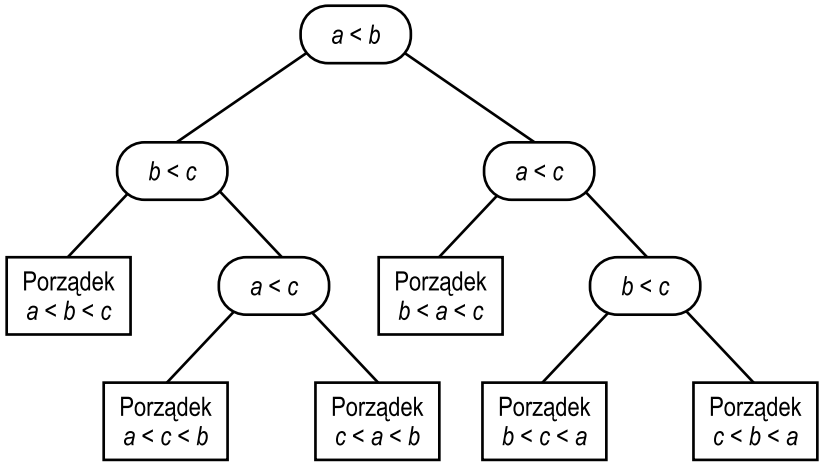
iv. Drzewa decyzji

Rozważyliśmy trzy abstrakcje RAM, które zaniedbywały instrukcje rozgałęzienia i obejmowały tylko kroki związane z obliczaniem. Istnieją pewne problemy, w których można realistycznie uznać liczbę instrukcji rozgałęzienia za podstawową miarę złożoności. W sortowaniu na przykład, wyjścia są identyczne z wejściami, wyjąwszy uporządkowanie. Rozsądnie jest więc rozważyć model, w którym wszystkie kroki są rozgałęzieniami od dwóch ramionach, i polegają na porównaniu dwóch wielkości.

Częstą reprezentacją programu z rozgałęzieniami jest drzewo binarne[7], zwane *drzewem decyzji*. Każdy wewnętrzny wierzchołek reprezentuje decyzję. Test reprezentowany przez korzeń jest wykonywany jako pierwszy, po czym zależnie od wyniku „sterowanie" przechodzi do jednego z synów. Ogólnie, sterowanie tak długo przechodzi od wierzchołka do jednego z synów, przy czym wybór zależy zawsze od testu na wierzchołku, aż dotrze do liścia. Wynik jest dostępny na tym liściu.

Przykład 1.6. Rys. 1.18 pokazuje drzewo decyzji dla programu, który sortuje trzy liczby a, b i c. Testy wskazują owale wokół porównań na wierzchołkach; sterowanie przechodzi na lewo, jeżeli test daje odpowiedź „tak", i na prawo, jeżeli „nie". □

Złożonością czasową drzewa decyzji jest jego wysokość jako funkcja rozmiaru zadania. Zwykle chcemy oszacować maksimum liczby porównań, które trzeba wykonać, by dojść z korzenia do liścia. Zakładając model drzewa decyzji (porównań), oznaczamy rząd wielkości przez O_{C}. Liczba wierzchołków może być znacznie większa od wysokości drzewa. Na przykład drzewo decyzji, które sortuje n liczb, musi mieć przynajmniej $n!$ liści, lecz wystarczy, że ma wysokość około $n \log n$.



Rys. 1.18. Drzewo decyzji

[7] W sprawie definicji dotyczących drzew patrz punkt 2.4.

1.6. Pierwotny model obliczania: maszyna Turinga

By udowodnić, że dana funkcja wymaga pewnego minimum czasu, potrzebujemy modelu, który jest równie ogólny, lecz bardziej pierwotny od rozpatrzonych. Repertuar instrukcji ma być jak najbardziej ograniczony, jednak model nie tylko musi obliczać to wszystko, co oblicza RAM, lecz czynić to niemal równie szybko. Według definicji, której użyjemy, „niemal" oznacza „równoważność wielomianową".

> **Definicja.** Mówimy, że funkcje $f_1(n)$ i $f_2(n)$ są *równoważne wielomianowo*, jeżeli istnieją wielomiany $p_1(x)$ i $p_2(x)$ takie, że dla wszystkich wartości n, $f_1(n) \leqslant p_1(f_2(n))$ i $f_2(n) \leqslant p_2(f_1(n))$.

Przykład 1.7. Funkcje $f_1(n) = 2n^2$ i $f_2(n) = n^5$ są równoważne wielomianowo; niech na przykład $p_1(x) = 2x$, skoro $2n^2 \leqslant 2n^5$, i $p_2(x) = x^3$, skoro $n^5 \leqslant (2n^2)^3$. Natomiast n^2 i 2^n nie są równoważne wielomianowo, gdyż nie istnieje wielomian $p(x)$, taki że dla każdego n, $p(n^2) \geqslant 2^n$. □

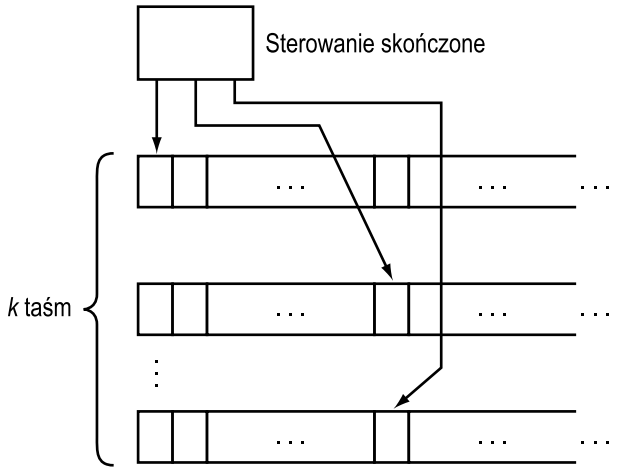
Obecnie jedynym zakresem, w którym do dowodu dolnych ograniczeń złożoności obliczeniowej możemy użyć ogólnych modeli, takich jak maszyna Turinga, jest „wyższy zakres". Na przykład w rozdziale 11. pokażemy, że pewne problemy wymagają wykładniczego czasu i pamięci. ($f(n)$ jest funkcją *wykładniczą*, jeżeli istnieją stałe $c_1 > 0$, $k_1 > 1$, $c_2 > 0$ i $k_2 > 1$ takie, że $c_1 k_1^n \leqslant f(n) \leqslant c_2 k_2^n$ dla wszystkich, prócz skończonej liczby wartości n.) W wykładniczym zakresie funkcje wielomianowo równoważne są zasadniczo tożsame, gdyż dowolna funkcja, która jest równoważna wielomianowo z funkcją wykładniczą, jest funkcją wykładniczą.

Jest więc powód, by używać pierwotnego modelu, w którym złożoność czasowa problemów jest równoważna wielomianowo ich złożoności w modelu RAM. Model, którego używamy — maszyna Turinga z wieloma taśmami — może wymagać czasu[8] ($[f(n)]^4$), lecz nie więcej, aby wykonać to, co RAM z funkcją kosztu logarytmicznego wykonuje w czasie $f(n)$. Złożoność czasowa z użyciem modelu RAM i maszyny Turinga będzie równoważna wielomianowo.

> **Definicja.** *Maszynę Turinga z wieloma taśmami* (TM) przedstawia rys. 1.19. Składa się ona z pewnej liczby k nieskończonych w prawo *taśm*. Każda taśma jest podzielona na *komórki*, a każda z nich zawiera jeden symbol spośród skończonej liczby *symboli taśm*. Jedna komórka na każdej taśmie jest czytana przez *głowicę taśmy*; głowica może czytać i pisać. Działanie maszyny Turinga jest określone przez pierwotny program, zwany *sterowaniem skończonym*. Sterowanie skończone jest zawsze w jednym ze skończonej liczby *stanów*, które można uznać pozycje w programie.

Jeden krok obliczeniowy maszyny Turinga zbudowany jest następująco. Zgodnie z bieżącym stanem sterowania skończonego i symbolami taśm, które ustawione

[8] Można udowodnić dokładniejsze ograniczenie: $O([f(n) \log f(n) \log\log f(n)]^2)$, lecz skoro nie rozważamy tu czynników wielomianowych, wynik z czwartą potęgą wystarczy (patrz p. 7.5).



Rys. 1.19. Maszyna Turinga z wieloma taśmami

są pod (są *czytane przez*) każdą z głowic taśm, maszyna Turinga może wykonać dowolną lub wszystkie z poniższych operacji.

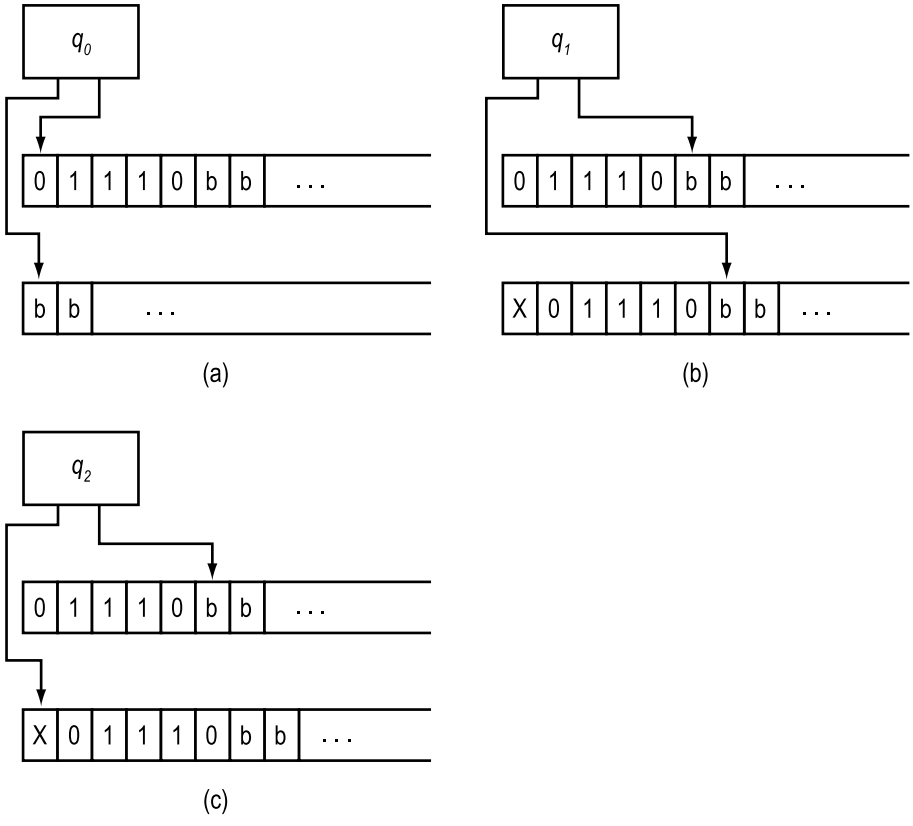
1. Zmienić stan sterowania skończonego.
2. Wydrukować nowe symbole taśm na bieżących symbolach w dowolnej lub każdej z komórkek pod głowicami taśm.
3. Przesunąć niezależnie dowolną lub każdą głowicę o jedną komórkę w lewo (L), lub w prawo (R), lub pozostawić głowice bez ruchu (S).

Formalnie oznaczamy k-taśmową maszynę Turinga przez siódemkę uporządkowaną:

$$(Q, T, I, \delta, b, q_0, q_f),$$

gdzie:

1. Q jest zbiorem *stanów*.
2. T jest zbiorem *symboli taśm*.
3. I jest zbiorem *symboli wejściowych*; $I \subseteq T$.
4. b, element $T - I$, jest *białym znakiem*.
5. q_0 jest *stanem początkowym*.
6. q_f jest *stanem końcowym* (lub *akceptującym*).
7. δ, *funkcja następnego ruchu*, odwzorowuje podzbiór $Q \times T^k$ w rodzinę podzbiorów $Q \times (T \times L, R, S)^k$. Tj. dla pewnych $(k+1)$-elementowych układów uporządkowanych, złożonych ze stanu i k symboli taśm daje nowy stan oraz k par uporządkowanych, złożonych z nowego symbolu taśm i kierunku dla głowicy. Załóżmy, że $\delta(q, a_1, a_2, \ldots, a_k) = (q', (a'_1, d_1), (a'_2, d_2), \ldots, (a'_k, d_k)$ i maszyna Turinga jest w stanie q i dla $1 \leqslant i \leqslant k$, i-ta głowica czyta symbol taśm a_i. Wtedy w jednym ruchu maszyna Turinga wchodzi w stan q', zmienia symbol a_i na a'_i i przesuwa i-tą głowicę w kierunku d_i dla $1 \leqslant i \leqslant k$.



Rys. 1.20. Maszyna Turinga przetwarzająca 01110

Maszyna Turinga może rozpoznawać język. Symbole taśm maszyny Turinga obejmują alfabet języka, zwany *symbolami wejściowymi*, specjalny *biały znak*, oznaczony przez b, i prócz tego być może inne symbole. Początkowo pierwsza taśma zawiera napis w symbolach wejściowych, po jednym symbolu w komórce, poczynając od komórki położonej najbardziej na lewo. Wszystkie komórki na prawo od komórek zawierających napis wejściowy są puste. Wszystkie inne taśmy są zupełnie puste. Napis w symbolach wejściowych jest *akceptowany* wtedy i tylko wtedy, gdy maszyna Turinga, zaczynając od wyróżnionego stanu początkowego ze wszystkimi głowicami na lewych końcach taśm wykonuje ciąg ruchów, w którym przechodzi kiedyś w stan akceptujący. *Język akceptowany* przez maszynę Turinga jest zbiorem akceptowanych w powyższym sensie napisów w symbolach wejściowych.

Przykład 1.8. Na rysunku 1.20 maszyna Turinga z dwiema taśmami rozpoznaje palindromy[9] nad alfabetem $\{0, 1\}$ w następujący sposób.

[9]Napis, który od tyłu można odczytać, tak jak do przodu, np. 0100010, nazywa się *palindromem*.

Stan bieżący	Symbol Taśma 1	Symbol Taśma 2	(Nowy symbol, ruch głowicy) Taśma 1	(Nowy symbol, ruch głowicy) Taśma 2	Nowy stan	Komentarze
q_0	0	b	0,S	X,R	q_1	Jeżeli dana nie jest pusta, drukuj X na taśmie 2 i przesuń głowicę w prawo; przejdź do stanu q_1. W przeciwnym razie przejdź do stanu q_5.
	1	b	1,S	X,R	q_1	
	b	b	b,S	b,S	q_5	
q_1	0	b	0,R	0,R	q_1	Pozostawaj w stanie q_1, kopiując taśmę 1 na 2, aż dotrzesz do b na taśmie 1. Wtedy przejdź do stanu q_2.
	1	b	1,R	1,R	q_1	
	b	b	b,S	b,L	q_2	
q_2	b	0	b,S	0,L	q_2	Pozostaw bez ruchu głowicę taśmy 1, a 2 przesuwaj w lewo, aż dotrzesz do X. Wtedy przejdź do stanu q_3.
	b	1	b,S	1,L	q_2	
	b	X	b,L	X,R	q_3	
q_3	0	0	0,S	0,R	q_4	Sterowanie na przemian w stanie q_3 i q_4. W q_3 porównaj symbole na obu taśmach, przesuń głowicę taśmy 2 w prawo i przejdź do q_4. W q_4 przejdź do q_5 i akceptuj, jeżeli głowica dotarła do b na taśmie 2. W przeciwnym razie przesuń głowicę taśmy 1 w lewo i wróć do q_3. Alternacja q_3, q_4 zapobiega przekroczeniu lewego końca taśmy przez głowicę wejściową.
	1	1	1,S	1,R	q_4	
q_4	0	0	0,L	0,S	q_3	
	0	1	0,L	1,S	q_3	
	1	0	1,L	0,S	q_3	
	1	1	1,L	1,S	q_3	
	0	b	0,S	b,S	q_5	
	1	b	1,S	b,S	q_5	
q_5						Akceptuj

Rys. 1.21. Funkcja następnego ruchu maszyny Turinga rozpoznającej palindromy

1. Pierwszą komórkę na taśmie 2 oznaczona specjalny symbol X; dane są kopiowane z taśmy 1, gdzie początkowo występują (patrz rys. 1.20a), na taśmę 2 (patrz rys. 1.20b).
2. Następnie głowica taśmy 2 jest przesuwana na X (rys. 1.20c),
3. Głowica taśmy 2 jest wielokrotnie przesuwana o jedną komórkę w prawo, głowica taśmy 1, o jedną komórkę w lewo, i porównywane są odpowiednie symbole. Jeżeli wszystkie symbole pasują, dane tworzą palindrom i maszyna wchodzi w stan akceptujący q_5. W przeciwnym razie maszyna Turinga nie będzie mogła w pewnej chwili zrobić żadnego poprawnego ruchu; zatrzyma się bez akceptowania.

Funkcję następnego ruchu tej maszynie Turinga podaje tablica z rysunku 1.21. □

Działanie maszyny Turinga można opisać formalnie za pomocą „chwilowych opisów". *Opis chwilowy* (ID, od *instantaneous description*) k-taśmowej maszyny Turinga M jest to k-elementowy układ uporządkowany $(\alpha_1, \alpha_2, \ldots, \alpha_k)$, gdzie α_i jest napisem postaci xqy takim, że xy jest napisem na i-tej taśmie M (pomijając koń-

$$
\begin{aligned}
(q_0010, q_0) &\vdash (q_0010,\ \mathrm{X}q_1)\\
&\vdash (0q_110,\ \mathrm{X}0q_1)\\
&\vdash (01q_10,\ \mathrm{X}01q_1)\\
&\vdash (010q_1,\ \mathrm{X}010q_1)\\
&\vdash (010q_2,\ \mathrm{X}01q_20)\\
&\vdash (010q_2,\ \mathrm{X}0q_210)\\
&\vdash (010q_2,\ \mathrm{X}q_2010)\\
&\vdash (010q_2,\ q_2\mathrm{X}010)\\
&\vdash (01q_30,\ \mathrm{X}q_3010)\\
&\vdash (01q_40,\ \mathrm{X}0q_410)\\
&\vdash (0q_310,\ \mathrm{X}0q_310)\\
&\vdash (0q_410,\ \mathrm{X}01q_40)\\
&\vdash (q_3010,\ \mathrm{X}01q_30)\\
&\vdash (q_4010,\ \mathrm{X}010q_4)\\
&\vdash (q_5010,\ \mathrm{X}010q_5)
\end{aligned}
$$

Rys. 1.22. Ciąg pewnych ID maszyny Turinga

cowe białe znaki), a q jest bieżącym stanem M. Symbol bezpośrednio na prawo od i-tego q jest symbolem obecnie czytanym na i-tej taśmie.

Gdy opis chwilowy D_1 staje się opisem chwilowym D_2 po jednym ruchu maszyny Turinga M, piszemy $D_1 \vdash_M D_2$. Jeżeli $D_1 \vdash_M D_2 \vdash_M \cdots \vdash_M D_n$ dla pewnego $n \geqslant 2$, piszemy $D_1 \vdash_M^+ D_n$. Jeżeli $D = D'$ lub $D \vdash_M^+ D'$, piszemy $D \vdash_M^* D'$. ($\vdash$ czytaj „przechodzi w".)

k-taśmowa maszyna Turinga $M = (Q, T, I, \delta, b, q_0, q_f)$ *akceptuje* napis $a_1a_2\cdots a_n$, gdzie a-ki należą do I, jeżeli $(q_0a_1a_2\cdots a_n, q_0, q_0, \ldots, q_0) \vdash_M^* (\alpha_1, \alpha_2, \ldots, \alpha_k)$, dla pewnych α_i, wśród których jest q_f.

Przykład 1.9. Ciąg opisów chwilowych, który wyznacza maszyna Turinga z rysunku 1.21, gdy otrzymuje dane 010, jest przedstawiony na rysunku 1.22. Skoro q_5 jest stanem końcowym, ta maszyna Turinga akceptuje 010. □

Oprócz naturalnej interpretacji, przy której maszyna Turinga akceptuje język, możliwa jest interpretacja, że jest to urządzenie do obliczania funkcji f. Argumenty tej funkcji są zakodowane na taśmie wejściowej jako napis x, przy czym rozgranicza je specjalny znacznik, taki jak #. Jeżeli maszyna Turinga zatrzymuje się z liczbą całkowitą y (wartość funkcji) zapisaną na taśmie, która jest wyróżniona jako *taśma wyjściowa*, mówimy, że $f(x) = y$. Stąd proces obliczania funkcji jest nieco inny niż akceptowania języka.

Złożoność czasowa $T(n)$ maszyny Turinga M jest to maksimum liczby ruchów, które wykonuje M przy przetwarzaniu dowolnych danych o długości n, dla wszystkich danych o długości n. Jeżeli dla pewnych danych o długości n maszyna Turinga nie zatrzymuje się, to $T(n)$ jest nieokreślona dla tej wartości n. *Złożoność pamięciowa* $S(n)$ maszyny Turinga jest to maksimum odległości od lewego końca taśmy, na którą dociera głowica taśmy przy przetwarzaniu dowolnych danych o długości n. Jeżeli głowica taśmy przesuwa się w prawo bez końca, $S(n)$ jest nieokreślona. Na oznaczenie rzędu wielkości, gdy modelem jest maszyna Turinga, używamy O_{TM}.

Przykład 1.10. Złożonością czasową maszyny Turinga z rysunku 1.21 jest $T(n) = 4n + 3$, a złożonością pamięciową $S(n) = n + 2$, jak można się przekonać, gdy dane faktycznie są palindromem. □

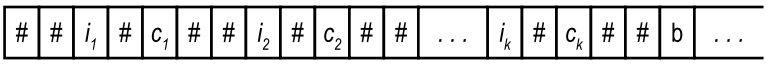
1.7. Związek pomiędzy maszyną Turinga i modelem RAM

Głównym zastosowaniem modelu maszyny Turinga (TM) jest wyznaczanie dolnych ograniczeń wiążących czas lub pamięć, które są konieczne do rozwiązania danego problemu. W większości przypadków możemy wyznaczyć dolne ograniczenia tylko z dokładnością do funkcji równoważnej wielomianowo. Wyprowadzenie dokładniejszych ograniczeń wymaga dalszych szczegółów danego modelu. Na szczęście obliczenia RAM i RASP są równoważne wielomianowo z obliczeniami TM.

Rozważmy związek pomiędzy modelami RAM i TM. Rzecz jasna RAM może symulować k-taśmową maszynę TM, przechowując jedną komórkę taśmy TM w każdym ze swoich rejestrów. W szczególności i-ta komórka taśmy j może być zapamiętana w rejestrze $ki + j + c$, gdzie c jest stałą dobraną tak, by zapewnić maszynie RAM pewną „pamięć roboczą". Pamięć robocza zawiera k rejestrów, w których przechowywane są pozycje k głowic TM. Komórki taśmy TM mogą być czytane przez RAM dzięki adresowaniu pośredniemu poprzez rejestry zawierające pozycje głowicy tej taśmy.

Załóżmy, że TM ma złożoność czasową $T(n) \geqslant n$. Wtedy RAM może przeczytać dane, umieścić je w rejestrach, które reprezentują pierwszą taśmę, i symulować TM w czasie $O(T(n))$, jeżeli używamy kryterium kosztu zuniformizowanego, lub $O(T(n) \log T(n))$, jeżeli używamy funkcji kosztu logarytmicznego. W obu przypadkach czas na maszynie RAM jest ograniczony od góry przez wielomianową funkcję czasu na TM, gdyż dowolna funkcja $O(T(n) \log T(n))$ jest z pewnością $O(T^2(n))$.

Odwrotne twierdzenie zachodzi tylko wtedy, gdy dla maszyn RAM obowiązuje koszt logarytmiczny. Gdy obowiązuje koszt zuniformizowany program RAM o n krokach bez wejścia może obliczać tak duże liczby, jak 2^{2^n}, co wymaga 2^n komórek TM już przy zapisie i odczycie. Wobec tego gdy obowiązuje koszt zuniformizowany, nie ma żadnej wyraźnej zależności wielomianowej pomiędzy maszynami RAM i TM (ćwiczenie 1.19).



Rys. 1.23. Reprezentacja RAM w TM

Chociaż ze względu na prostotę wolimy używać kosztu zuniformizowanego w analizie algorytmów, musimy go odrzucić w dowodach dolnych ograniczeń złożoności czasowej. Model RAM z kosztem zuniformizowanym jest zupełnie rozsądny, gdy liczby nie rosną nadmiernie wraz z rozmiarem zadania. Lecz, jak mówiliśmy wcześniej, model RAM „zamiata pod dywan" wielkość tych liczb i tylko wyjątkowo można otrzymać użyteczne ograniczenia dolne. Dla kosztu logarytmicznego mamy jednakże następujące twierdzenie.

> **Twierdzenie 1.3.** Niech L będzie językiem akceptowanym przez program RAM o złożoności czasowej $T(n)$ według kryterium kosztu logarytmicznego. Jeżeli ten program RAM nie wykonuje mnożenia i dzielenia, to L na maszynie Turinga z wieloma taśmami ma złożoność co najwyżej $O(T^2(n))$.

Dowód. Wszystkie rejestry RAM, które nie zawierają 0, reprezentujemy tak, jak na rysunku 1.23. Taśma zawiera ciąg par (i_j, c_j), zapisanych w postaci binarnej bez wiodących zer i rozgraniczonych znacznikami. Dla każdego j, c_j jest zawartością rejestru i_j RAM. Zawartość akumulatora leży w postaci binarnej na drugiej taśmie; trzecia taśma służy jako pamięć robocza. Dwie inne taśmy zawierają wejście i wyjście RAM. Krok programu RAM jest reprezentowany przez skończony zbiór stanów TM. Nie będziemy opisywać symulacji dowolnej instrukcji RAM, lecz rozważymy tylko instrukcje ADD $*20$ i STORE 30, co wiele wyjaśni. Dla ADD $*20$ możemy zbudować TM do wykonania następujących czynności.

1. Szukaj na taśmie 1 zapisu dla rejestru 20 RAM, tj. sekwencji ##10100#. Jeżeli jest taka, to następującą po tej sekwencji liczbę całkowitą, która musi być zawartością rejestru 20, umieść na taśmie 3. Jeżeli takiej nie ma, zatrzymaj się. Zawartością rejestru 20 jest 0, wobec czego adresowanie pośrednie jest niemożliwe.
2. Szukaj na taśmie 1 zapisu dla rejestru RAM, którego numer jest umieszczony na taśmie 3. Jeżeli jest taki zapis, kopiuj zawartość tego rejestru na taśmę 3. Jeżeli nie ma, umieść tam 0.
3. Dodaj liczbę umieszczoną na taśmie 3 w kroku 2. do zawartości akumulatora, który jest utrzymywany na taśmie 2.

Aby symulować instrukcję STORE 30, można zbudować TM do wykonania następujących czynności:

1. Szukaj zapisu dla rejestru 30 RAM, tj. ##11110#.
2. Jeżeli jest taki, kopiuj wszystko, co znajduje się na prawo od ##11110#, prócz liczby całkowitej bezpośrednio na prawo (stara zawartość rejestru 30), na taśmę 3. Następnie kopiuj zawartość akumulatora (taśma 2) bezpośrednio na prawo od ##11110# i dopisz do niej napis skopiowany na taśmę 3.

3. Jeżeli nie ma zapisu dla rejestru 30 na taśmie 1, przejdź w takim razie do białego znaku, położonego najbardziej na lewo, drukuj ##11110#, dopisz zawartość akumulatora, a następnie ##.

Po chwili namysłu powinno być jasne, że można zbudować TM do wiernej symulacji RAM. Musimy pokazać, że obliczenia RAM, które mają koszt logarytmiczny k, wymagają co najwyżej $O(k^2)$ kroków maszyny Turinga. Rozpoczynamy od stwierdzenia, że rejestr nie pojawia się na taśmie 1, chyba że wcześniej została w nim umieszczona jego bieżąca wartość. Kosztem zapamiętania c_j w rejestrze j jest $l(c_j)+l(i_j)$, co z dokładnością do stałej oznacza długość reprezentacji $\#\#i_j\#c_j\#\#$. Wnioskujemy, że długość niepustej części taśmy 1 jest $O(k)$.

Symulacja dowolnej innej instrukcji, różnej od STORE, jest rzędu długości taśmy 1, czyli $O(k)$, gdyż koszt dominujący stanowi szukanie na taśmie. Podobnie koszt STORE jest nie większy niż koszt szukania na taśmie 1 plus koszt jej kopiowania, oba $O(k)$. Stąd jedna instrukcja RAM (poza mnóż i dziel) może być symulowana w co najwyżej $O(k)$ krokach TM. Skoro instrukcja RAM według kryterium kosztu logarytmicznego kosztuje przynajmniej jedną jednostkę czasu, ogólny czas zużywany przez TM jest $O(k^2)$, co było do udowodnienia. □

Jeżeli program RAM zawiera instrukcje mnożenia i dzielenia, można napisać procedury TM, aby zaimplementować te instrukcje za pomocą dodawania i odejmowania. Dowód, że koszt logarytmiczny tych procedur jest nie większy niż kwadrat kosztu logarytmicznego instrukcji, które symulują, pozostawiamy Czytelnikowi. Nie trudno udowodnić następujące twierdzenie.

Twierdzenie 1.4. RAM i RASP z kosztem logarytmicznym oraz maszyna Turinga są modelami równoważnymi wielomianowo.

Dowód. Należy skorzystać z twierdzeń 1.1, 1.2 i 1.3, i własnej analizy procedur mnożenia i dzielenia. □

Analogiczne twierdzenie zachodzi dla złożoności pamięciowej, lecz wynik ten wydaje się mniej ciekawy.

1.8. Pidgin ALGOL — język wysokiego poziomu

Chociaż podstawowe miary złożoności zostały określone w sensie operacji RAM, RASP lub maszyny Turinga, to przeważnie nie chcemy opisywać algorytmów w sensie tak prymitywnych maszyn, ani nie jest to wcale konieczne. Do jaśniejszego opisu algorytmów użyjemy języka wysokiego poziomu, zwanego Pidgin ALGOL.

Program w języku Pidgin ALGOL może być łatwo przetłumaczony na program RAM lub RASP. W istocie jest to zadanie kompilatora Pidgin ALGOLu. Nie będziemy jednak zajmować się szczegółami tłumaczenia Pidgin ALGOLu na kod RAM lub RASP. Musimy uwzględnić dla naszych celów tylko czas i pamięć potrzebne do wykonania kodu, który odpowiada instrukcji języka Pidgin ALGOL.

Inaczej niż konwencjonalne języki programowania Pidgin ALGOL pozwala na użycie dowolnego wyrażenia matematycznego, o ile jego znaczenie jest jasne, a przekład na kod RAM lub RASP oczywisty. Język ten nie ma stałego zbioru typów danych. Zmienne mogą reprezentować liczby całkowite, napisy lub tablice. Typy danych, takie jak zbiory, grafy, listy i kolejki można wprowadzać w miarę potrzeb. Wszędzie, gdzie to możliwe, unika się formalnych deklaracji typów danych. Typ zmiennej i jej zasięg[10] powinien być jasny na podstawie jej nazwy lub kontekstu.

Pidgin ALGOL ma tradycyjne konstrukcje językowe, takie jak wyrażenia, warunki, instrukcje i procedury. Nieformalny opis niektórych podajemy poniżej. Próba ścisłej definicji wykraczałaby znacznie poza zakres tej książki. Trzeba zauważyć, że z łatwością można napisać programy, których sens będzie zależeć od szczegółów, jakich tutaj nie omawiamy, lecz tego należy unikać, jak to (miejmy nadzieję skutecznie) czynimy w tej książce.

W języku Pidgin ALGOL *program* jest instrukcją jednego z następujących typów:

1. zmienna ← wyrażenie
2. **if** warunek **then** instrukcja **else** instrukcja[11]

3a. **while** warunek **do** instrukcja

b. **repeat** instrukcja **until** warunek

4. **for** zmienna ← wartość początkowa **step** rozmiar kroku[12] **until** wartość końcowa **do** instrukcja
5. etykieta: instrukcja
6. **goto** etykieta
7. **begin**
   instrukcja
   instrukcja
   .
   .
   .
   instrukcja
   instrukcja
   end

8a. **procedure** nazwa (lista parametrów): instrukcja

b. **return** wyrażenie

c. nazwa procedury (argumenty)

9a. **read** zmienna

b. **write** wyrażenie

10. **comment** komentarz
11. różne inne instrukcje dodatkowe

[10] *Zasięg* zmiennej jest otoczeniem, w którym ona ma sens. Na przykład zasięg wskaźnika sumowania jest określony tylko wewnątrz sumowania i poza nim nie ma sensu.

[11] Część „**else** instrukcja" jest nieobowiązkowa. Opcja ta prowadzi do znanej wieloznaczności „chwiejnego" **else** (*dangling else*). Uciekniemy się do tradycyji i założymy, że **else** odpowiada najbliższemu **then** bez pary.

[12] Część „**step** rozmiar kroku" jest nieobowiązkowa, jeżeli rozmiarem kroku jest 1.

Przedstawimy krótki przegląd każdego z tych typów instrukcji.

1. Instrukcja przypisania

zmienna ← wyrażenie

powoduje, że wyrażenie po prawej stronie ← ulega obliczeniu i jego wartość jest przypisywana zmiennej po lewej stronie. Złożoność czasowa instrukcji przypisania jest czasem zużytym na to, by obliczyć wartość wyrażenia i przypisać tę wartość do zmiennej. Jeżeli wartość wyrażenia nie jest typu podstawowego, takiego jak typ całkowity, to w pewnych przypadkach można obniżyć koszt za pomocą wskaźników. Na przykład przypisanie $A \leftarrow B$, gdzie A i B są macierzami wymiaru $n \times n$, wymaga na ogół czasu $O(n^2)$. Jeżeli jednak nie używa się dłużej B, to można uzyskać czas skończony i niezależny od n, zwyczajnie zmieniając nazwę tablicy.

2. W instrukcji **if**

if warunek **then** instrukcja **else** instrukcja

warunkiem następującym po **if** może być dowolne wyrażenie, które ma wartość **true** lub **false**. Jeżeli warunek ma wartość **true**, wykonana będzie instrukcja następująca po **then**. W przeciwnym razie będzie wykonana instrukcja następująca po **else** (jeżeli występuje). Koszt instrukcji **if** jest to suma kosztów niezbędnych, by obliczyć i sprawdzić wartość wyrażenia, plus koszt instrukcji następującej po **then**, lub instrukcji następującej po **else**, zależnie od tego, która z nich faktycznie jest wykonana.

3. Celem instrukcji **while**

while warunek **do** instrukcja

oraz instrukcji **repeat**

repeat instrukcja **until** warunek

jest tworzenie pętli. W insrukcji **while** obliczana jest wartość warunku następującego po **while**. Jeżeli warunek ma wartość **true**, wykonywana jest instrukcja zadana po **do**. Proces ten jest powtarzany, aż wartością warunku stanie się **false**. Gdy warunek ma początkowo wartość **true**, to wykonanie zadanej instrukcji spowoduje kiedyś, że uzyska on wartość **false**, jeżeli wykonanie instrukcji **while** ma się zakończyć. Koszt instrukcji **while** jest to suma kosztów obliczania wartości warunku tylekroć, ilekroć obliczana jest ta wrtość, plus suma kosztów wykonania zadanej instrukcji tylekroć, ilekroć jest wykonywana.

Instrukcja **repeat** jest podobna, wyjąwszy to, że instrukcja następująca po **repeat** jest wykonywana zanim warunek będzie poddany obliczaniu.

4. W instrukcji **for**

for zmienna ← wartość początkowa **step** rozmiar kroku **until** wartość końcowa **do** instrukcja

wartość początkowa, rozmiar kroku i wartość końcowa są wyrażeniami. W przypadku, gdy rozmiar kroku jest dodatni, wymienionej zmiennej (zwanej *indeks*) nadaje się wartość równą wartości wyrażenia, wymienionego jako wartość początkowa. Jeżeli wartość ta przewyższa wartość końcową, to wykonanie ulega zakończeniu. W przeciwnym razie wykonywana jest instrukcja następująca po **do**, wartość zmiennej jest zwiększana o rozmiar kroku, a potem porównywana z wartością końcową. Proces ten jest powtarzany, aż wartość zmiennej przewyższy wartość końcową. W przypadku, w którym rozmiar kroku jest ujemny, dzieje się podobnie, lecz zakończenie następuje, gdy wartość zmiennej jest mniejsza niż wartość końcowa. Koszt instrukcji **for** powinien być oczywisty w świetle wcześniejszej analizy instrukcji **while**.

Powyższy opis pomija zupełnie takie szczegóły, jak to, kiedy obliczane są wartości wyrażeń na wartość początkową, rozmiar kroku i wartość końcową. Niewykluczone również, że wykonanie instrukcji następującej po **do** modyfikuje wartość wyrażenia na rozmiar kroku, a wtedy obliczanie wartości wyrażenia na rozmiar kroku za każdym razem, gdy zmienna jest zwiększana, wywiera inny efekt, niż obliczenie rozmiaru kroku raz na zawsze. Od obliczenia wartości dla rozmiaru kroku może również zależeć wartość wyrażenia na wartość końcową, a zmiana znaku rozmiaru kroku zmienia warunek zakończenia. [13] Problemy te rozwiązujemy, powstrzymując się od pisania programów, których sens może stać się niejasny wskutek takich zjawisk.

5. Przez poprzedzenie instrukcji etykietą, po której następuje dwukropek, można z każdej instrukcji utworzyć *instrukcję z etykietą*. Głównym zadaniem etykiety jest oznaczenie celu dla instrukcji **goto**. Z etykietą nie jest związany żaden koszt.

6. Instrukcja **goto**

goto etykieta

powoduje, że jako następna jest wykonywana instrukcja z daną etykietą. Oznaczonej tą etykietą instrukcji nie wolno być wewnątrz instrukcji-bloku (7), chyba że instrukcja **goto** należy do tej samej instrukcji-bloku. Kosztem instrukcji **goto** jest jeden. Instrukcje **goto** powinny być używane oszczędnie, ponieważ zwykle powodują, że programy są trudne do zrozumienia. Instrukcje **goto** służą przede wszystkim do wyskakiwania z instrukcji **while**.

7. Sekwencja instrukcji rozgraniczonych średnikami i osadzonych pomiędzy słowami kluczowymi **begin** i **end** jest instrukcją zwaną *blokiem*. Skoro blok jest instrukcją, może być używany wszędzie tam, gdzie można użyć instrukcji. Program na ogół będzie blokiem. Koszt bloku jest to suma kosztów instrukcji występujących wewnątrz bloku.

8. *Procedury*. W języku Pidgin ALGOL procedury mogą być definiowane, a następnie wywołane. Procedury są definiowane przez *intrukcję definicji procedury*, która ma postać:

[13] Ang. *test for termination*; (z.– (wk.)) × *sign*(rk.) — *przyp. tłum.*

procedure nazwa (lista parametrów): instrukcja

Lista parametrów jest to ciąg zwanych *parametrami formalnymi* nazw zmiennych. Przykładowo następująca instrukcja definiuje procedurę funkcji, nazwaną MIN:

```
procedure MIN(x, y):
if x > y then return y else return x
```

Argumenty x i y są parametrami formalnymi.

Procedury są używane na jeden z dwóch sposobów. Po pierwsze jako *funkcje*. Gdy procedura funkcji jest zdefiniowana, może być wywołana w wyrażeniu przez użycie jej nazwy z pożądanymi argumentami. W takim przypadku ostatnią instrukcją wykonaną w procedurze musi być instrukcja **return** 8(b). Instrukcja **return** powoduje obliczenie wartości wyrażenia następującego po słowie kluczowym **return** i zakończenie wykonania procedury. Wartością funkcji jest wartość tego wyrażenia. Na przykład:

$$A \leftarrow \text{MIN}(2 + 3, 7)$$

powoduje, że A otrzymuje wartość 5. Wyrażenia $2 + 3$ i 7 nazywane są *parametrami aktualnymi* tego wywołania procedury.

Druga metoda użycia procedury pozwala wywołać ją przez instrukcję wywołania procedury 8(c). Instrukcja ta jest tylko nazwą procedury, po której następuje lista parametrów aktualnych. Instrukcja wywołania procedury może modyfikować (i zwykle to czyni) dane wywołującego programu. Wywołana w ten sposób procedura nie wymaga instrukcji **return** w swej definicji. Dokończenie wykonania ostatniej instrukcji w procedurze kończy wykonanie instrukcji wywołania procedury. Przykładowo następująca instrukcja definiuje procedurę, nazwaną INTERCHANGE:

```
procedure INTERCHANGE(x, y):
begin
    t ← x;
    x ← y;
    y ← t
end
```

Aby wywołać tę procedurę, możemy napisać instrukcję wywołania procedury, taką jak:

$$\text{INTERCHANGE}(A[\,i\,],\ A[\,j\,])$$

Istnieją dwie metody, którymi procedura może komunikować się z innymi procedurami. Po pierwsze przez zmienne globalne. Zakładamy, że zmienne globalne są deklarowane domyślnie w pewnym uniwersalnym środowisku. W tym środowisku istnieje otoczenie (*subenvironment*), w której definiowane są procedury.

Drugą z metod komunikacji z procedurami jest użycie parametrów. ALGOL 60 posługuje się wywołaniem przez wartość i wywołaniem przez nazwę. W *wywołaniu przez wartość* parametry formalne procedury są traktowane jak zmienne lokalne, które inicjuje się wartościami parametrów aktualnych. W *wywołaniu przez nazwę* parametry formalne służą do oznaczania miejsc w programie, parametry aktualne podstawia się za każde wystąpienie odpowiednich parametrów formalnych. Dla prostoty odstąpimy od ALGOLu 60 i użyjemy wywołania przez odniesienie. W *wywołaniu przez odniesienie* parametry są przekazywane poprzez wskaźniki do parametrów aktualnych. Jeżeli parametr aktualny jest wyrażeniem (bądź stałą), to odpowiedni parametr formalny traktowany jest jak zmienna lokalna, inicjowana wartością wyrażenia. Wobec tego koszt funkcji lub procedury (*procedure-call*) w implementacji RAM lub RASP jest sumą kosztów wykonania instrukcji, które należą do definicji procedury. Koszt i implementacja procedury, która wywołuje inne procedury, bądź siebie samą, jest omówiony w rozdziale 2.

9. Instrukcja **read** oraz instrukcja **write** mają jasny sens. Instrukcja **read** ma koszt 1. Instrukcja **write** ma koszt jeden plus koszt obliczenia wartości wyrażenia następującego po słowie kluczowym **write**.

10. Instrukcja **comment** pozwala na wstawianie uwag, które mają pomóc w zrozumieniu programu i ma koszt zero.

11. Dodatkowo prócz konwencjonalnych instrukcji języka programowania dołączamy w punkcie „różne" każdą instrukcję, która pomaga zrozumieć algorytm lepiej, niż czyni to równoważna sekwencja instrukcji języka programowania. Instrukcje tego rodzaju używane są wtedy, gdy szczegóły implementacji są bądź nieistotne, bądź oczywiste, albo gdy pożądany jest wyższy poziom opisu. Przykładami często używanych instrukcji dodatkowych są:

a) niech a będzie najmniejszym elementem zbioru S
b) oznacz element a jako "stary"[14]
c) **without loss of generality (wlg)** załóż, że ... **otherwise** ... **in** instrukcja. Na przykład:

   wlg załóż, że $a \leqslant b$ **otherwise** zamień c i d **in** instrukcja

   znaczy, że jeżeli $a \leqslant b$, to należy wykonać instrukcję w zapisanej postaci. Jeżeli $a > b$, to należy wykonać instrukcję w postaci, w której role c i d uległy zamianie.

Implementacja tych instrukcji przez konwencjonalne instrukcje języka programowania albo kod RAM jest bezpośrednia, lecz pracochłonna. Przypisanie kosztu instrukcjom tego rodzaju zależy od kontekstu, w którym występują. Dalsze ich przykłady można znaleźć w programach Pidgin ALGOLu, które zawiera książka.

[14] Zakładamy tym samym, że istnieje tablica STATUS taka, że STATUS[a] jest 1, jeżeli a jest "stary", i 0, jeżeli a jest "nowy".

Ponieważ zmienne na ogół nie będą deklarowane, powinniśmy przyjąć pewną umowę, co do zasięgu zmiennych. W danym programie lub procedurze nie używamy tej samej nazwy dla dwóch różnych zmiennych. Stąd zwykle za zasięg zmiennej można wziąć całą procedurę lub program, w którym ta zmienna występuje.[15] Ważny wyjątek stanowi wspólna baza danych, na której operuje kilka procedur. W takim przypadku zmienne bazy danych uznawane są za globalne, natomiast zmienne wykorzystywane przez procedury jako pamięć tymczasowa przy operacjach na wspólnej bazie danych uznaje się za zmienne lokalne tych procedur. Jeśliby mogło wystąpić kiedyś nieporozumienie co do zasięgu zmiennych, zostanie dostarczona wyraźna deklaracja.

Ćwiczenia

1.1 Udowodnić, że $g(n)$ jest $O(f(n))$, jeżeli (a) $f(n) \geqslant \epsilon$ dla pewnego $\epsilon > 0$ i wszystkich n, prócz pewnego skończonego zbioru n, i (b) istnieją stałe $c_1 > 0$ i $c_2 > 0$ takie, że $g(n) \leqslant c_1 f(n) + c_2$ dla prawie wszystkich $n \geqslant 0$.

1.2 Piszemy $f(n) \preccurlyeq g(n)$, jeżeli istnieje dodatnia stała c taka, że $f(n) \leqslant cg(n)$ dla wszystkich n. Udowodnić, że jeżeli $f_1 \preccurlyeq g_1$ i $f_2 \preccurlyeq g_2$, to $f_1 + f_2 \preccurlyeq g_1 + g_2$. Jakie inne własności przysługują relacji $\preccurlyeq$?

1.3 Podać programy RAM, RASP i Pidgin ALGOLu dla następujących zadań.
a) Oblicz $n!$ dla danego wejścia n.
b) Czytaj n liczb całkowitych dodatnich, po których następuje znacznik końca (0), a następnie drukuj owe n liczb w posortowanym porządku (ang. *sorted order*).
c) Akceptuj wszystkie wejścia o postaci $1^n 2^{n^2} 0$.

1.4 Zbadać złożoność czasową i pamięciową swoich odpowiedzi w ćwiczeniu 1.3, jeżeli obowiązuje (a) koszt zuniformizowany (b) koszt logarytmiczny. Wyrazić swoją miarę „wielkości" danych.

***1.5** Napisać program RAM o złożoności czasowej $O(\log n)$ przy koszcie zuniformizowanym do obliczania n^n. Udowodnić, że program jest poprawny.

***1.6** Pokazać, że dla każdego programu RAM o złożoności czasowej $T(n)$ przy funkcji kosztu zuniformizowanego istnieje równoważny program RAM o złożoności czasowej $O(T^2(n))$, który nie zawiera instrukcji MULT i DIV. *Wskazówka*: Zasymulować MULT i DIV przez podprogramy, które na pamięć roboczą wykorzystują rejestry o numerach parzystych. Dla MULT pokazać, że jeżeli trzeba pomnożyć i przez j, to każdy z $l(j)$ iloczynów częściowych oraz ich sumę można obliczyć w $O(l(j))$ krokach, przy czym każdy krok wymaga czasu $O(l(i))$.

[15] Zachodzą pewne niezbyt istotne wyjątki od tego postanowienia. Na przykład procedura może mieć dwie niezagnieżdżone instrukcje **for**, obie z indeksem i. Mówiąc ściśle, zasięgiem indeksu instrukcji **for** jest instrukcja **for**, więc każde z tych i jest inną zmienną.

***1.7** Co stanie się z mocą obliczeniową RAM lub RASP, jeżeli MULT i ADD zostaną usunięte z repertuaru instrukcji? Jak wpłynie to na koszt obliczeń?

****1.8** Pokazać, że dowolny język, akceptowany przez RAM, może być akceptowany przez RAM bez adresowania pośredniego. *Wskazówka*: Pokazać, że całą taśmę TM można zakodować w postaci jednej liczby całkowitej. Zatem dowolna maszyna Turinga może być symulowana w skończonej liczbie rejestrów RAM.

1.9 Pokazać, że przy kosztcie (a) zuniformizowanym i (b) logarytmicznym, RAM i RASP są równoważne z dokładnością do czynnika stałego ze względu na złożoność pamięciową.

1.10 Znaleźć program liniowy, który oblicza wyznacznik macierzy wymiaru 3×3, dla danych, którymi jest dziewięć skalarnych elementów tej macierzy.

1.11 Napisać sekwencję operacji bitowych do obliczania iloczynu dwóch dwubitowych liczb całkowitych

1.12 Pokazać, że układ funkcji obliczanych przez dowolny program liniowy o n instrukcjach, z binarnymi wejściami i operatorami boolowskimi może być zrealizowany przez układ logiczno-kombinatoryczny z n elementami boolowskimi.

1.13 Pokazać, że każdą funkcję boolowską oblicza pewien program liniowy.

***1.14** Załóżmy, że graf o n wierzchołkach jest reprezentowany przez zbiór wektorów bitowych $\mathbf{v}_i$, gdzie $\mathbf{v}_i$ ma j-ty element 1 wtedy i tylko wtedy, gdy istnieje krawędź prowadząca od wierzchołka i do wierzchołka j. Znaleźć algorytm $O_{\mathrm{BV}}(n)$ wyznaczania wektora $\mathbf{p}_1$, który ma 1 na pozycji j wtedy i tylko wtedy, gdy istnieje droga łącząca 1 z wierzchołkiem j. Można używać bitowych operacji logicznych na wektorach bitów, operacji arytmetycznych (na zmiennych, które są „typu całkowitego"), instrukcji, które ustawiają w pewnych bitach pewnych wektorów 0 lub 1, oraz instrukcji, która przypisuje j do zmiennej a, jeżeli bit 1 położny najbardziej na lewo w wektorze $\mathbf{v}$, znajduje się na pozycji j, i ustawia $a = 0$, jeżeli $\mathbf{v}$ składa się z samych 0.

***1.15** Opisać maszynę Turinga, która, mając dane dwie binarne liczby całkowite na taśmach 1 i 2, drukuje sumę tych liczb na taśmie 3. Można założyć, że lewe końce taśm są oznaczone specjalnym symbolem #.

1.16 Podać ciąg konfiguracji, w który wchodzi TM z rysunku 1.21 (str. 37), otrzymując dane (a) 0010, (b) 01110.

***1.17** Podać TM, która:
a) drukuje 0^{n^2} na taśmie 2, jeżeli zaczyna działanie z 0^n na taśmie 1,
b) akceptuje dane o postaci $0^n 10^{n^2}$.

1.18 Podać zbiór stanów TM i funkcję następnego ruchu, które pozwolą TM symulować instrukcję RAM LOAD 3 tak, jak w dowodzie twierdzenia 1.3.

1.19 Podać program RAM w $O(n)$ krokach, który oblicza 2^{2^n} dla danego n. Jaki jest koszt (a) zuniformizowany i (b) logarytmiczny tego programu?

***1.20** Definiujemy $g(m,n)$ przez $g(0,n) = n$ i $g(m,n) = 2^{g(m-1,n)}$ dla $m > 0$. Podać program RAM do obliczania $g(n,n)$ dla danego n. Jak mają się do siebie koszt zuniformizowany i logarytmiczny tego programu?

1.21 Wykonać procedurę INTERCHANGE z punktu 1.8 z parametrami aktualnymi i, $A[i]$, używając wywołania przez nazwę, a następnie przez odniesienie. Czy wyniki są takie same?

Problem badawczy

1.22 Czy górne ograniczenie $O(T^2(n))$ czasu wymaganego przez maszynę Turinga do symulacji RAM, jak w twierdzeniu 1.3, można poprawić?

Noty bibliograficzne

RAM i RASP znalazły ujęcie formalne w pracach Shepherdson i Sturgis [1963], Elgot i Robinson [1964], oraz Hartmanis [1971]. Większość przedstawionych tutaj wyników, dotyczących maszyn RAM i RASP, jest wzorowana na pracy Cook i Reckshow [1973].

Maszynę Turinga zawdzięczamy pracy Turing [1936]. Bardziej szczegółowy wykład tego pojęcia można znaleźć w pracach Minsky [1967], oraz Hopcroft i Ullman [1969], tak samo jak odpowiedź do ćwiczenia 1.8. Złożoność czasowa maszyn Turinga była po raz pierwszy badana w pracy Hartmanis i Stearns [1965], a złożoność pamięciowa w pracach Hartmanis, Lewis i Stearns [1965] oraz Lewis, Stearns i Hartmanis [1965]. Pojęciu złożoności obliczeniowej poświęcono wiele badań teoretycznych, poczynając od pracy Blum [1967]. Przegląd można znaleźć w pracach Hartmanis i Hopcroft [1971], oraz Borodin [1973a]

Praca Rabin [1972] przedstawia interesujące rozszerzenie obliczeniowego modelu drzewa decyzji.