2
Czy komputery mogą być nieobliczalne?

Na pierwszy rzut oka termin „nieobliczalność" budzi skojarzenia typowo ludzkie. Wszak nieobliczalnymi nazywa się ludzi nieprzewidywalnych, a bywa, że – ze względu na ową nieprzewidywalność właśnie – groźnych dla otoczenia. Nieobliczalny zatem jest każdy szaleniec, a mówiąc nieco mniej drastycznie, każdy osobnik ekstrawagancki.

Interesujący nas termin ma jednak inne jeszcze znaczenie – dobrze znane z podręczników matematyki i jej informatycznych zastosowań. Otóż nieobliczalnymi zwie się tam szczególnego rodzaju funkcje, których wartości (przynajmniej niektóre) nie dają się uzyskać algorytmicznie, a więc w sposób całkowicie określony i możliwy do powierzenia maszynie (w domyśle: maszynie spełniającej pewne założenia teoretyczne, wyłuszczone po raz pierwszy przez Alana Turinga; ▷zob. esej 10, §2).

I tu właśnie – na polu komputerowej realizacji złożonych zadań – znaczenie pierwsze, czyli ludzkie, spotyka się ze znaczeniem drugim, komputerowym. Skoro bowiem istnieją funkcje nieobliczalne – a wraz z nimi problemy, dla których rozwiązania, trzeba wyznaczać wartości tychże funkcji – to nie wszystkie zadania komputer może wykonać niezawodnie. Jeśli zatem zlecimy maszynie pewne „nieobliczalne zadanie specjalne" (przykłady takich zadań podamy dalej), to musimy liczyć się z tym, że próbująca je wykonać maszyna nie będzie działać w sposób do końca przewidywalny. Tym samym zaś musimy wziąć w rachubę to, że maszyna stanie się nieobliczalna „po ludzku"...

§1. Co zrobią – trudno przewidzieć

1.1. O tym, że komputery bywają nieprzewidywalne, przekonuje aż nadto praktyka ich użytkowania. Weźmy przykład pierwszy z brzegu: wirusy, robaki, konie trojańskie i inne komputerowe *szkodniki*. Za ich istnienie odpowiadają, jak wiadomo, ludzie.

To właśnie komputerowi programiści wymyślili – prawdopodobnie dla żartu lub z chęci oderwania się od zawodowej monotonni – maleńkie, samokopiujące się programy, które potrafią dołączać się do innych i w tej zakamuflowanej postaci przenosić się z komputera na komputer. Kiedyś wędrowały na dyskietkach, dziś naturalnym środowiskiem ich ekspansji stał się Internet. W czym takie programy szkodzą? Czasami w niczym: wyświetlają po prostu jakiś żarcik lub wydają dziwaczne dźwięki. Czasami jednak są naprawdę złośliwe: niszczą dane, sieją spustoszenie w systemie, czy – jak to ma miejsce w przypadku tzw. koni trojańskich – umożliwiają włamania do komputerów z zewnątrz.

Mimo tego wszystkiego dla większości użytkowników spotkanie z wirtualnym szkodnikiem to zdarzenie albo całkiem niegroźne, albo wręcz „odświętne" (do czego przyczynia się w głównej mierze instalowane na komputerach, już niemal obowiązkowo, oprogramowanie „antyszkodnicze").

1.2. O wiele bardziej powszechne są takie oto sytuacje. Przynosimy do domu płytkę ze świeżo zakupionym programem, wkładamy ją do stacji CD-ROM i uruchamiamy komputer. Po kilkunastu sekundach system pyta nas, czy życzymy sobie zainstalować nowy produkt. Odpowiadamy twierdząco, po czym wykonujemy szereg czynności sugerowanych przez system. Po pewnym czasie na ekranie pojawia się krzepiący napis: „Instalacja została zakończona pomyślnie". Wtedy rozsiadamy się wygodnie w fotelu i wywołujemy program. Ale co to? Jedna funkcja działa, druga nie działa, nie wyświetlają się polskie litery... Próbujemy tego, owego – nic. Zamykamy program i uruchamiamy go ponownie. Gdy i to nie pomaga, próbujemy coś zmienić w systemie. I wtedy następuje totalny klops: myszka przestaje reagować, klawiatura nie odpowiada, nie możemy zrobić absolutnie nic. Cóż się stało? Ano, wersja naszego programu jest po prostu *niekompatybilna*, czyli niezgodna, z użytkowaną aktualnie wersją systemu operacyjnego.

Podobne sytuacje nie są tylko i wyłącznie zmorą komputerowych żółtodziobów. Przydarzają się nader często najbardziej prestiżowym instytucjom i najbardziej znanym firmom softwarowym. Wtedy jednak skala możliwych strat jest znacznie, znacznie większa. Przypomnijmy zdarzenie z roku 1997, gdy na powierzchni Marsa wylądował Pathfinder – automatyczny łazik, sterowany z Ziemi za pomocą wyrafinowanego systemu komputerowego. Niedługo po osiągnięciu celu sonda przestała odpowiadać. Mimo wielu prób nawiązania łączności i poprawiania ustawień systemu, problem udało się rozwiązać w sposób najbardziej toporny – poprzez zresetowanie systemu i uruchomienie go na nowo. W tym wypadku wszystko skończyło się szczęśliwie, bywały jednak awarie bardziej brzemienne w skutki. Na przykład, na początku lat sześćdziesiątych z powodu niedoskonałości oprogramowania zaginął jeden z amerykańskich statków kosmicznych podążających w kierunku Wenus. Wtedy straty sięgnęły milionów dolarów.

Widzimy więc, na czym może polegać nieobliczalność komputerów: są one po prostu *nieprzewidywalne*. Sterujące nimi programy mają taką złożoność, że mimo najlepszych chęci nawet ich autorzy (a zwykle są to całe zespoły autorów) nie są w stanie przewidzieć wszystkich ich możliwych zachowań we wszelkich możliwych sytuacjach.

§2. Złożoność czasowa algorytmów

Nieprzewidywalność komputerów może wydać się niektórym złudna. Wszak nie chodzi1 tu o jakąś fundamentalną niewydolność maszyn czy słabość narzędzi programistycznych, lecz o niedociągnięcia, a czasami wręcz złośliwe intencje, ich twórców. Innymi słowy: to nie maszyna jest nieprzewidywalna, ale człowiek, który ją konstruuje i programuje.

2.1. W odpowiedzi na tak radykalną samokrytykę mamy dla czytelnika dobre wieści. Są one dobre, bo uwalniają nas od poczucia winy; w ogólności jednak są bardzo, bardzo złe. Otóż istnieją problemy, dla których rozwiązania nie można wymyślić albo żadnych w ogóle algorytmów, albo algorytmów wystarczająco sprawnych. W ich przypadku wina nie leży ani po stronie człowieka, ani po stronie programowanego przezeń komputera. To same problemy są nieobliczalne. Kluczem do zrozumienia tej frapującej kwestii jest pojęcie *złożoności czasowej* algorytmów. Omówimy je stopniowo, zaczynając od przywołania treści pewnej bajeczki – bodajże starochińskiej.

> *Oto przed władcą silnego i zasobnego państwa stanął pewnego razu bystry wieśniak, który znał niezwykle cenną dla władcy tajemnicę. Nie bacząc na swój niski stan społeczny, odezwał się w te słowa: „Panie, wyjawię Ci ów cenny sekret, jeśli Ty spełnisz moją skromną prośbę". Tu wyjął zza pazuchy planszę do gry w szachy i kontynuował: „Położysz Panie na pierwszym polu szachownicy jedno ziarnko zboża, na drugim dwa ziarenka, na trzecim cztery i tak na każdym następnym polu dwa razy więcej niż na poprzednim. Dasz mi tyle ziaren, ile musiałbyś położyć na ostatnim, 64-tym polu szachownicy". Głupota wieśniaka rozbawiła króla setnie – wszak przeliczał on tajemnice wagi państwowej na jakieś nędzne ziarenka. Zgodził się więc bez wahania. Gdybyż wiedział, że liczba ziaren, której zażądał wieśniak, 2^{64}, przewyższała setki razy to, co byłby w stanie zgromadzić w spichlerzach przez dziesiątki lat.*

O tym, czy zgoda króla doprowadziła kraj do ruiny, czy też, co bardziej prawdopodobne, rezolutny chłop skończył w ciemnym lochu, nasza pamięć milczy. Zostaje jednak ważny, matematyczny morał: oto istnieją funkcje, nazywane wykładniczymi, które rosną niewiarygodnie szybko. Jedną z nich jest 2^n.

W tym miejscu warto przytoczyć pewną autentyczną historię, która uwspółcześni nieco bajeczkę o wieśniaku i władcy. Otóż nie więcej niż 29 lat temu, w roku 1982, Donald Knuth – autor systemu TEX, przy użyciu którego była składana ta książka – obmyślił sprytną strategię doskonalenia swojego systemu. Była to strategia wykładnicza. Ogłosił, że każdy, kto wykryje dowolny błąd w systemie, otrzyma gratyfikację pieniężną. Za błąd wykryty w pierwszym roku był gotów zapłacić aż 256 centów, za każdą usterkę wykrytą w roku drugim – 512 centów, w trzecim roku – dwa razy więcej, w czwartym – dwa razy więcej niż w trzecim itd., itd... nie w nieskończoność jednak, bo jako górną granicę (swoisty wentyl bezpieczeństwa) określono kwotę 327,68$. Choć można byłoby przypuścić, że system rosnących wykładniczo wypłat doprowadzi Donalda Knutha do ruiny, to tak się nie stało. Innymi słowy: program TeX nie okazał się bublem. W pierwszym roku testów wykryto 300 błędów, w drugim już tylko około 30, a w kolejnych po kilka raptem. Ostatnią usterkę zgłoszono aż 9 lat temu (!), w roku 2001. Dociekliwy Czytelnik zechce ustalić, czy wypłata za jej odkrycie wyniosła 327,68$, czy może mniej[1].

2.2. Każdy, kto liznął choć trochę teorii i praktyki programowania, wie że dla rozwiązania tych samych problemów można wymyślić algorytmy mniej lub bardziej *sprawne*. Dla przykładu: proste zadanie posortowania listy nazwisk dopuszcza kilka istotnie różnych rozwiązań. Gdy postawi się je przed początkującym, zwykle wpada on na pomysł słabiutkiej metody „sortowania bąbelkowego", profesjonalista wybierze algorytm *QuickSort* albo tzw. „sortowanie przez scalanie" (por. [Wirth 1989]).

Wspomniany profesjonalista wiedziałby zapewne, że najważniejszy wyznacznik sprawności programu to szybkość jego wykonywania. Mierzy się ją w ilości przeprowadzanych operacji elementarnych, takich jak porównywanie wartości, dodawanie wartości czy skoki do innych miejsc programu. Oczywiście im większy rozmiar danych, na których pracuje program, tym więcej operacji elementarnych trzeba wykonać. Na przykład, gdy program sortuje listę 1000 nazwisk, musi wykonać o wiele więcej działań, niż gdyby lista liczyła nazwisk 100.

Dla większości algorytmów daje się oszacować ścisłą, matematyczną zależność liczby operacji od rozmiaru danych. Nazywa się ją *złożonością czasową*. Parametr ten obrazuje, jak widać, nie czas realizowania przez komputer konkretnego zadania, ale to, jak czas ów wzrasta wraz ze wzrostem rozmiaru danych. Mówi zatem, czy algorytm jest „dobry" i można go stosować dla dowolnie dużych danych (mała złożoność), czy też jest „słaby" i sprawdza się tylko dla małych danych (duża złożoność). Złożoność czasową algorytmów wyraża się za pomocą funkcji matematycznych, takich jak n, n^2, $\log_2 n$ i 2^n (gdzie n symbolizuje rozmiar danych). Mówi się na przykład, że „algorytm jest rzędu n^2", co znaczy, że czas jego wykonywania rośnie proporcjonalnie do kwadratu rozmiaru danych. A więc, gdy za pomocą algorytmu takiego

[1] Przytoczoną historyjkę zawdzięczamy p. Jarosławowi Sokołowskiemu, który składał książkę właśnie za pomocą programu TEX (jak widać nie tylko składał, lecz wczuwał się w jej treść). Serdecznie dziękujemy.

sortujemy listę pięciu nazwisk, musimy wykonać 25 operacji, dla listy 6 elementów 36 operacji itd.

§3. Problemy łatwe, trudne i najtrudniejsze

3.1. Widzimy więc, że złożoności algorytmów bywają różne. Elementarna znajomość funkcji matematycznych podpowiada, że najlepsze (tj. najszybsze) są algorytmy rzędu $\log_a n$, trochę gorsze rzędu n, n^2 (i ogólnie n^k), a najgorsze rzędu 2^n (czy ogólnie a^n). To właśnie algorytmy o złożoności *wykładniczej* (a więc np. 2^n) stawiają komputery w sytuacji podobnej do króla z przytoczonej wcześniej bajeczki. Gdy rozmiar danych wejściowych rośnie, a przecież najbardziej interesujące są problemy o skomplikowanych danych, liczba koniecznych do wykonania operacji-ziarenek namnaża się w tempie iście zawrotnym. Matematyka przekonuje, że nic tu nie da nawet kilkusetkrotne przyspieszenie procesorów. Przy dostatecznie dużych danych na rozwiązanie trzeba by czekać miliony lat.

Czytelnik zadaje sobie zapewne pytanie: jakież to problemy są tak trudne, iż nie istnieją dla nich algorytmy lepsze niż wykładnicze. Spieszymy z odpowiedzią: jest ich bez liku. Wymieńmy dla przykładu trzy. Pierwszy z nich to prawdziwa zmora pracowników administracyjnych różnego rodzaju szkół, a mianowicie układanie *planów zajęć*. Mając do dyspozycji takie dane, jak wymiary godzinowe poszczególnych przedmiotów, dostępność sal w poszczególnych terminach, planowane obciążenia nauczycieli i ich życzenia co do prowadzenia zajęć w określonych godzinach, należy opracować algorytm, który przypisywałby poszczególnym przedmiotom godziny, sale i nazwiska nauczycieli. Drugi przykład to z kolei utrapienie niektórych młodych adeptów logiki. Chodzi o sprawdzanie prawdziwości formuł rachunku zdań metodą *zero-jedynkową*. Okazuje się, że problem ten ma złożoność wykładniczą, a więc dla długich formuł, zawierających bardzo dużo zdań atomowych, jest praktycznie nierozwiązywalny. Trzeci przykład reprezentuje dość szeroki krąg trudnych zagadnień projektowania sieci połączeń, np. komunikacyjnych lub energetycznych. Nazywa się go *problemem komiwojażera*. Oto dane jest n miast połączonych ze sobą drogami o różnych długościach. Od algorytmu żąda się znalezienia najkrótszej trasy objazdu tychże miast, przy założeniu że każde z nich musi zostać odwiedzone przynajmniej raz. Wykazano, że ten problem również nie należy do łatwych (łatwo-obliczalnych), bo ma złożoność wykładniczą (por. [Harel 2000]).

3.2. Problemy opisane wyżej można nazwać praktycznie nieobliczalnymi, bo choć istnieją algorytmy do ich rozwiązywania, to ich złożoność jest (niestety) co najmniej wykładnicza. To zaś uniemożliwia w praktyce komputerową realizację tychże zadań dla „dużych danych". Gdyby ktoś o podejściu bardziej teo-

retycznym pocieszał się jednak, że w przypadku zagadnień, o których mowa, istnieją jednak pewne precyzyjne (choć niekiedy niewykonalne w praktyce) przepisy postępowania, to ponownie mamy dla niego złe wieści. Otóż istnieją zagadnienia, które są *prawdziwie* (czyli w ogóle) *nieobliczalne*.

Informatycy skatalogowali ich całkiem sporo. Najbardziej wyrafinowane dotykają granic informatyki, a obracają się wokół fascynujących pytań. Dla przykładu: „Czy wszystkie problemy matematyczne, a więc ściśle zdefiniowane, mają rozwiązania algorytmiczne?", „Czy daje się napisać program, który dla dowolnego danego problemu odpowiadałby, czy ma on jakieś rozwiązanie algorytmiczne?". Albo nieco konkretniej, choć bardziej technicznie: „Czy istnieje super-program komputerowy, który otrzymując na wejściu kod dowolnego innego programu, jest w stanie odpowiedzieć jednoznacznie i w skończonym czasie, że program ten zakończy pracę?". Odpowiedzi na każde z nich są zdecydowanie negatywne. Dopowiedzmy nadto, że matematycznego uzasadnienia owej tezy negatywnej dokonano u progu rewolucji informatycznej, a stało się to za sprawą osób, o których więcej w esejach 10, p.t. *„Czy umysł jest liczbą?"*, oraz 19, p.t. *„Dynamika umysłu w perspektywie gödlowskiej"*.

Nie chcąc pozostać na poziomie tak ogólnym, wypada przedstawić czytelnikowi przykład bardziej konkretny. Oto i on: załóżmy, że mamy dwa języki programowania ze ściśle określonymi instrukcjami i regułami budowy z tych instrukcji złożonych procedur. Chcemy napisać inny program, który dla dwóch takich dowolnych języków sprawdzałby, czy są one *równoważne składniowo*, a więc, czy z ich instrukcji podstawowych daje się zbudować takie same zbiory instrukcji złożonych. Okazuje się, że programu takiego napisać się nie daje.

§4. Trzy poziomy trudności

Spróbujmy podsumować i uzupełnić to, co powiedziano. Za kanwę takiego resumé niech posłuży znakomita książka Davida Harela, *Algorytmika. Rzecz o istocie informatyki* (polecamy ją szczerze wszystkim miłośnikom komputerów). Wieńczące ją post scriptum skłania do następujących refleksji.

4.1. Przed twórcami programów komputerowych i projektantami złożonych systemów piętrzą się trudności trojakiego rodzaju. Pierwsze z nich zwykło się nazywać – nieco technicznie i być może myląco – *obliczeniowymi*. Chodzi o to, że istnieją pewne problemy podstawowe, które daje się opisać dla potrzeb rozwiązań algorytmicznych, ale nie opracowano dla nich algorytmów wystarczająco sprawnych. Sprawnych, to znaczy wykonywanych szybko, najlepiej w czasie „podążającym wielomianowo" za rosnącym rozmiarem danych, wymagających małej ilości zasobów itp. Dla niektórych problemów (tych, które

zwie się nieobliczalnymi) udowodniono wręcz, że nie istnieją żadne w ogóle rozwiązujące je algorytmy.

Trudności drugiego rodzaju można nazwać *zachowaniowymi*, a dotyczą one nie problemów podstawowych, lecz złożonych zadań zawierających w sobie wiele problemów (które wzięte z osobna nie muszą być nieobliczalne ani efektywnie nieobliczalne). To właśnie w ich przypadku grożą nam złości i przykrości, o których wspominaliśmy na początku. Zachowania systemów przeznaczonych do uporania się z takimi zadaniami, w tym systemów antywirusowych, są po prostu trudno przewidywalne. Dlatego lwia część pieniędzy łożonych na ich projektowanie dotyczy nie algorytmów, nie konkretnych rozwiązań, ale zwykłych testów. Dopiero po serii długotrwałych sprawdzianów zyskujemy gwarancję, że system będzie działał mniej więcej tak, jak zaplanowali to jego twórcy.

Trzeci rodzaj trudności, o których wyżej nie wspominaliśmy, to kłopoty – rzec by można – *poznawcze*. Wiążą się one z próbami skomputeryzowania rzeczywistych procesów przetwarzania informacji – rzeczywistych, a więc wykonywanych przez ludzi. Zbytnim optymistom w tej materii, np. orędownikom tzw. sztucznej inteligencji (▷zob. esej 3 i esej 13), dedykujemy dwie obserwacje. Po pierwsze, istnieje mnóstwo programów do gry w szachy, które grają perfekcyjnie i są w stanie pokonywać arcymistrzów. Wystarczy jednak zmienić jedną regułę gry (np. wprowadzić nowe zasady ruchu konika), a programy takie, w przeciwieństwie do ludzi, stają się bezużyteczne. Po drugie, mamy mnóstwo programów do wykonywania skomplikowanych obliczeń inżynierskich, ale rzecz tak prosta dla nas, jak rozpoznawanie twarzy, jest dla komputerów nieosiągalna.

Dlaczego kłopoty wyłuszczone wyżej nazywać poznawczymi? Dla dwóch ważkich powodów. Z jednej strony, dotyczą one procesów poznawczych ludzi (spostrzegania, uczenia się, komunikowania), z drugiej strony zaś, nasza ogólna wiedza o tych procesach jest jak na razie dalece niewystarczająca.

4.2. Jeśli komputerowa realizacja pewnych, bardzo istotnych, zadań napotyka opisane wyżej trudności, a mimo to nie chcemy poprzestawać na poziomie mało skomplikowanych danych, musimy zadać arcyważne pytanie: *czy coś da się zrobić*? Nad jego pozytywnym rozstrzygnięciem informatycy głowią się od lat. Wspomnijmy o dwóch rozważanych strategiach.

Pierwsza polega na opracowywaniu algorytmów równoległych, tj. takich, w których informacje byłyby przetwarzanie nie sekwencyjnie, jedna po drugiej, lecz równolegle, nawet po kilkaset na raz. Zauważmy, że tak właśnie – równolegle, a nie sekwencyjnie – pracuje ludzki mózg. Zauważmy nadto, że informatycy – niezależnie od prac na procesorami równoległymi – wynaleźli pewien sprytny i odwołujący się wprost do analogii komputer-mózg sposób przetwarzania danych. Sposób ten realizują w praktyce różnego rodzaju sztuczne sieci neuronowe (▷zob. esej 4, §2).

Druga strategia polega na wyposażaniu komputerów w wiedzę, najlepiej podobną do ludzkiej. Przecież ludzie jakoś rozwiązują trudne problemy: układają plany zajęć, dowodzą twierdzeń matematycznych, potrafią kierować skomplikowanymi urządzeniami (chociażby takimi jak wspomniany wcześniej marsjański łazik). Swoją sprawność zawdzięczają być może nie tylko przyrodzonej inteligencji, ale i gromadzonym przez lata doświadczeniom. Poprzez doświadczenie (zależne w największej mierze od uczenia się) nabierają umiejętności działania „na skróty", pomijania nieistotnych szczegółów, wybierania tylko takich metod, które obiecują osiągnięcie celu. A więc, czy tak właśnie – poprzez wszczepianie w nie ludzkich doświadczeń lub samodzielne uczenie się – należałoby programować komputery? Jeśli czytelnik widzi sens takiej właśnie strategii postępowania, może zajrzeć do eseju 5, p.t. *„Czy komputery powinny się uczyć?"*.

WARTO PRZECZYTAĆ

Harel D., *Rzecz o istocie informatyki. Algorytmika*, Wydawnictwa Naukowo-Techniczne, Warszawa 2000.

Wirth N., *Algorytmy + struktury danych = programy*, Wydawnictwa Naukowo-Techniczne, Warszawa 1989.

Bolc L., Zaremba J., *Wprowadzenie do uczenia się maszyn*, Akademicka Oficyna Wydawnicza RM, Warszawa 1992.