wartość, zaczynając od 1. Potem sprawdzamy, czy wartość na pozycji i macierzy jest równa wartości x. Jeśli tak jest, zwracamy numer pozycji i. W przeciwnym przypadku zwiększamy i i próbujemy kolejną pozycję, aż do osiągnięcia wartości n, czyli długości macierzy, a wtedy zwracamy 0.

Dla tego rodzaju algorytmu obliczamy złożoność jako liczbę iteracji pętli: 1 w najlepszym przypadku (jeśli x jest równy macierzy [1]), n zaś w najgorszym przypadku (jeśli z jest równy macierzy [n] lub, gdy wartości x nie ma w macierzy), a średnio $n/2$, jeśli x jest losowo umieszczony w jednej z n komórek macierzy. Przy macierzy 10 razy większej algorytm będzie 10 razy wolniejszy. Złożoność jest więc proporcjonalna do n, czyli „liniowa" względem n. Złożoność liniowa względem n jest uważana za szybką, w przeciwieństwie do złożoności wykładniczej względem n. Wprawdzie przetwarzanie więcej wartości na wejściu będzie wolniejsze, ale w większości zastosowań praktycznych będą to różnice sekundowe.

Jednak wiele użytecznych algorytmów jest wolniejsza i ma złożoność większą niż liniowa. Książkowym przykładem są algorytmy sortowania: mając listę n wartości w porządku losowym potrzebujemy średnio $n \times \log n$ podstawowych działań, aby uporządkować listę, co czasami określa się jako *złożoność liniowo-logarytmiczną* (*linearithmic complexity*). Ponieważ $n \times \log n$ rośnie szybciej niż n, szybkość sortowania będzie zwalniać szybciej niż proporcjonalnie do n, jednak takie algorytmy sortowania pozostają w przestrzeni obliczeń *praktycznych*, czyli obliczeń wykonywanych w rozsądnym czasie.

W jakimś punkcie dochodzimy do kresu możliwości stwierdzenia, czy coś daje się rozwiązać nawet dla relatywnie małych wielkości danych. Weźmy najprostszy przykład z kryptoanalizy: poszukiwanie siłowe tajnego klucza. Jak pamiętamy z rozdziału 1, mając tekst jawny P i szyfrogram $C = \mathbf{E}(K, P)$, znalezienie n-bitowego klucza symetrycznego wymaga 2^n prób, gdyż mamy 2^n możliwych kluczy – to przykład, gdy złożoność rośnie wykładniczo. Według teoretyków *złożoność wykładnicza* (*exponential complexity*) oznacza problem, który jest praktycznie niemożliwy do rozwiązania, gdyż w miarę jak n rośnie, nakład pracy potrzebny do znalezienia rozwiązania bardzo szybko staje się nieosiągalny.

Możemy mieć obiekcje, że porównujemy tu jabłka i pomarańcze: w funkcji `search()` z listingu 9.1 liczyliśmy liczbę operacji `if (array[i] == x)`, podczas gdy znajdowanie klucza liczy liczbę szyfrowań, a każda jest tysiące razy wolniejsza od zwykłego porównania. Ta niespójność może mieć znaczenie, jeśli porównujemy dwa algorytmy o podobnej złożoności, ale w większości przypadków nie ma to znaczenia, gdyż liczba operacji będzie miała większy wpływ niż koszt pojedynczej operacji. Ponadto oszacowania złożoności ignorują *składniki stałe*: gdy mówimy, że algorytm zajmuje czas rzędu n^3 (co jest złożonością sześcienną), może to zająć w rzeczywistości $41 \times n^3$ operacji lub $12345 \times n^3$ operacji – ale i tak w miarę jak n rośnie, składniki stałe tracą na znaczeniu, aż do momentu, gdy można je zignorować. Analiza złożoności dotyczy teoretycznej trudności jako funkcji wielkości wejścia; nie ma znaczenia dokładna liczba cykli procesora potrzebnych na naszym komputerze.