Rozdział 3.

Analiza złożoności algorytmów

Podstawowe kryteria pozwalające na wybór właściwego algorytmu zależą głównie od kontekstu, w jakim zamierzamy go używać. Jeśli chodzi nam o sporadyczne korzystanie z programu do celów „domowych" czy też po prostu prezentacji wykładowej, współczynnikiem najczęściej decydującym bywa *prostota algorytmu*.

Nieco inna sytuacja powstaje w momencie zamierzonej komercjalizacji programu, ewentualnie udostępnienia go szerszej grupie osób.

Klient, który otrzymuje płytę z programem w postaci binarnej (gotowym do instalacji i użytkowania), jest w nikłym stopniu (jeśli w ogóle) zainteresowany estetyką wewnętrzną programu, klarownością i pięknem użytych algorytmów, etc. Użytkownik ten — zwany czasem *końcowym* — będzie się koncentrował na tym, co jest dla niego bezpośrednio dostępne: rozbudowanych systemach menu, pomocy kontekstowej, jakości prezentacji wyników w postaci graficznej itp. Taki punkt widzenia jest często spotykany i programista, który zapomni go uwzględnić, ryzykuje wyeliminowaniem się z rynku programów komercyjnych.

Konflikt interesów, z którym mamy tu do czynienia, jest zresztą typowy dla wszelkich relacji typu producent – klient: pierwszy jest głęboko zainteresowany, aby stworzyć produkt jak najtaniej i sprzedać jak najdrożej, aby stworzyć program jak *najtaniej* i sprzedać jak *najdrożej*, natomiast drugi chciałby za niewielką sumę nabyć produkt najwyższej jakości. Upraszczając dla potrzeb naszej dyskusji wyżej zaanonsowaną problematykę, możemy wyróżnić podstawowe kryteria oceny programu. Są to:

- sposób komunikacji z użytkownikiem,
- szybkość wykonywania podstawowych funkcji programu,
- awaryjność.

W rozdziale tym zajmiemy się wyłącznie aspektem sprawnościowym wykonywania programów, problem komunikacji — jako zbyt obszerny — zostawiając może na inną okazję, a o zagadnieniach awaryjności oprogramowania już w ogóle nie będę wspominał!

Tematyką tego rozdziału jest tzw. złożoność obliczeniowa algorytmów, czyli próba odpowiedzi na pytanie: *który z dwóch programów wykonujących to samo zadanie (ale odmiennymi metodami) jest efektywniejszy?* Wbrew pozorom w wielu przypadkach odpowiedź wcale nie jest taka prosta i wymaga użycia dość złożonego aparatu matematycznego. Nie będzie jednak wymagane od Czytelnika posiadanie jakichś szczególnych kwalifikacji matematycznych — prezentowane metody będą w dużym stopniu uproszczone i nastawione raczej na zastosowania praktyczne niż teoretyczne studia.

Istotna uwaga należy się osobom, które byłyby głębiej zainteresowane stroną matematyczną prezentowanych zagadnień, dowodami użytych metod, etc. Ponieważ głównym kryterium doboru zaprezentowanych narzędzi matematycznych była ich prostota, a nie kompletność i zgodność z wszelkimi formalizmami, w celu pogłębienia wiedzy odsyłam do podręczników analizy matematycznej. W końcu nie każdy programista musi być matematykiem.

Tych Czytelników, którym brakuje nieco formalizmu matematycznego, można odesłać do dokładniejszej lektury, np. **[BB87]**, **[Gri84]**, **[Kro89]** czy też klasycznych tytułów: **[Knu73]**, **[Knu69]**, **[Knu75]**.

Pomocne będą także zwykłe podręczniki matematyczne, ale należy zdawać sobie sprawę z tego, iż częstokroć zawierają one nadmiar informacji i wyłuskanie tego, co jest nam niezbędne, jest znacznie trudniejsze niż w przypadku tytułów z założenia przeznaczonych dla programistów.

Definicje i przykłady

Zanurzając się w problematykę analizy sprawnościowej programów, możemy wyróżnić minimum dwa ważne czynniki wpływające na dobre samopoczucie użytkownika programu:

- czas wykonania (*znowu się zawiesił, czy też coś liczy?!*);
- zajętość pamięci (mam już dość komunikatów typu: *Insufficient memory — save your work*[1]).

Z uwagi na znaczne potanienie pamięci RAM w ostatnich latach to drugie kryterium straciło już praktycznie na znaczeniu[2]. Co innego jest z pierwszym! Wcale nie jest aż tak dobrze z szybkością współczesnych komputerów[3], aby przestać się tym zupełnie przejmować. Bo cóż z tego, że komputer X jest 12 razy szybszy od Y, jeśli dla algorytmu A i problemu P oznacza to przyspieszenie czasu zakończenia obliczeń z... 12 lat do „zaledwie" jednego roku?! Abstrahuję tu od tego, że nikt by algorytmu A do tego zadania nie użył. Dla tego samego problemu znaleziono inny algorytm, który zrobił to samo w przeciągu kilku godzin.

Jednym ze szczególnie istotnych problemów w dziedzinie analizy algorytmów jest dobór właściwej *miary złożoności obliczeniowej*. Musi być to miara na tyle reprezentatywna, aby użytkownicy np. małego komputera osobistego i potężnej stacji roboczej — obaj używający tego samego algorytmu — mogli się ze sobą porozumieć co do jego sprawności obliczeniowej. Jeśli ktoś stwierdzi, że *jego program jest szybki, bo wykonał się w 1 minutę*, to nie dostaniemy w ten sposób żadnej reprezentatywnej informacji. Musi on jeszcze odpowiedzieć na dodatkowe pytania, na przykład:

- Jakiego komputera użył?
- Jaka była liczba przetwarzanych informacji?
- Jaka jest częstotliwość pracy zegara taktującego procesor?
- Czy program był jedynym wykonującym się wówczas w pamięci? Jeśli nie, to jaki miał priorytet?

---

[1] *Brak pamięci — zachowaj swoje dane*: niegdyś częsty w wielu profesjonalnych programach pisanych dla systemu Windows. Obecnie z uwagi na duże dyski twarde symulujące pamięć dynamiczną występuje on rzadziej, ale czy nie wynika to bardziej z postępu technologii niż ze wzrostu jakości oprogramowania?

[2] Stwierdzenie to jest fałszywe w odniesieniu do niektórych dziedzin techniki; niektóre algorytmy używane w syntezie obrazu pochłaniają tyle pamięci, że w praktyce są ciągle nieużywane w komputerach osobistych. Ponadto należy sobie zdać sprawę, że obsługa skomplikowanych struktur danych jest na ogół dość czasochłonna — jedno kryterium oddziałuje zatem na drugie!

[3] Oczywiście mam na myśli komputery osobiste.

- Jakiego kompilatora użyto podczas pisania tego programu?
- Jeśli to był kompilator *XYZ*, to czy zostały włączone opcje optymalizacji kodu?

Od razu jednak widać, że daleko w ten sposób nie zajdziemy. Potrzebna jest nam *miara uniwersalna*, niemająca nic wspólnego ze szczegółami natury, nazwijmy to, sprzętowej.


Uwaga

W świecie „fizycznym", gdzie mamy do czynienia z pamięciami dyskowymi, sieciami i opóźnieniami transmisji, często operuje się pojęciem *wydajności oprogramowania*. Odbiorca oprogramowania definiuje wobec dostawcy swoje wymagania wydajnościowe (np. czasy przetwarzania raportów miesięcznych), a dostawca próbuje je zrealizować, biorąc pod uwagę wszelkie uwarunkowania technologiczne. Wydajność jest jednak dalece mylącym pojęciem, bowiem tak naprawdę niewiele mówi o zaaplikowanych algorytmach. Znacznie bardziej interesujące byłoby porównanie (nawiązując do podanego wyżej przykładu) czasu wykonania raportu miesięcznego w przypadku podwojenia liczby danych wejściowych (np. rekordów z bazy danych).

Parametrem najczęściej decydującym o czasie wykonania określonego algorytmu jest rozmiar danych *n*, z którymi ma on do czynienia[4]. Pojęcie *rozmiaru danych* jest wieloznaczne: dla funkcji sortującej tablicę będzie to po prostu rozmiar tablicy, natomiast dla programu obliczającego wartość funkcji `silnia` — wielkość danej wejściowej.

Podobnie funkcja wyszukująca dane w liście (patrz rozdział 5.) będzie bardzo „uczulona" na jej długość. Wszystkie te przypadki określa się właśnie jako rozmiar danych wejściowych. Ponieważ odczytanie właściwego znaczenia tego terminu jest intuicyjnie bardzo proste, dalej będziemy używać właśnie tego nieprecyzyjnego określenia w miejsce rozwlekłych wyjaśnień cytowanych powyżej.


Definicja

Uogólniając, można powiedzieć, że *n* jest to najbardziej znaczący parametr algorytmu, wpływający na czas jego wykonania.

Powróćmy jeszcze do przykładu przytoczonego na samym początku tego rozdziału. Nieprzygotowany Czytelnik, widząc stwierdzenia: „czas wykonania programu to 12 lat", ma prawo się nieco obruszyć — czy to jest w ogóle możliwe?! W istocie w miarę rozwoju techniki mamy do czynienia z coraz szybszymi komputerami i być może kiedyś 12 lat mogło być nawet prawdą, ale obecnie?

Niestety trzeba podkreślić, że podany czas wcale nie jest tak przerażająco długi. Proszę spojrzeć na tabelę 3.1.

Tabela 3.1. *Czasy wykonania programów dla algorytmów różnej klasy*

	10	*20*	*30*	*40*	*50*	*60*
n	0,000 01 s	0,000 02s	0,000 03 s	0,000 04 s	0,000 05 s	0,000 06
n^2	0,000 1 s	0,000 4s	0,000 09 s	0,001 6 s	0,002 5 s	0,003 6s
n^3	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,216 s
2^n	0,001 s	1,0 s	17,9 min	12,7 dni	35,7 lat	366 w
3^n	0,59 s	58 min	6,5 lat	3855 w	$200 \cdot 10^6$ w.	$1{,}3 \cdot 10^{13}$ w.
$n!$	3,6 s	768 w.	$8{,}4 \cdot 10^{16}$ w.	$2{,}6 \cdot 10^{32}$ w.	$9{,}6 \cdot 10^{48}$ w.	$2{,}6\ 10^{66}$ w.

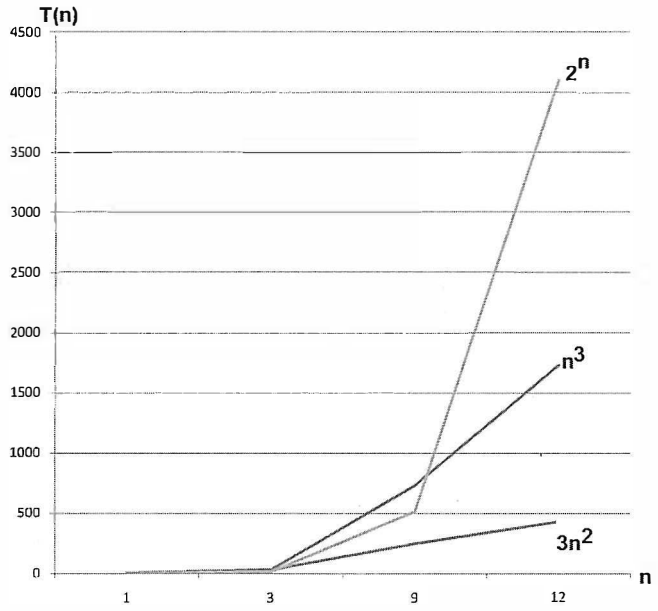
[4] W toku dalszego wykładu okaże się, że nie jest to bynajmniej jedyny współczynnik decydujący o czasie wykonania programu.

Tabela zawiera krótkie zestawienie czasów wykonania algorytmów[5] przy następujących założeniach:

- Niech czas wykonania algorytmu *A* jest proporcjonalny do pewnej wybranej funkcji matematycznej, np. dla danej wejściowej o rozmiarze `x` i funkcji `n!` czas wykonania programu jest proporcjonalny do `x!`.
- Niech czas wykonania operacji elementarnej wynosi jedną mikrosekundę (np. dla n = 10 i funkcji silnia wynik `10!=3628800/1000000=3,6s`).

Przy powyższych założeniach można otrzymać zacytowane w tabelce wyniki — dość szokujące, zwłaszcza jeśli spojrzymy na ostatnie jej pozycje.

Popatrzmy jeszcze, jak szybko wzrasta czas wykonania programów dla pozornie bliskich sobie klas funkcji (rysunek 3.1), przy nawet małym wzroście *n*.



Rysunek 3.1. *Graficzna ilustracja czasów wykonania algorytmów różnej klasy*

Wartości *n* zostały celowo dobrane jako bardzo małe, gdyż... tylko dla nich wyniki mieściły się na wykresie!

Wykres ten powinien nas doprowadzić do istotnej konkluzji: tylko dla małych klas złożoności (bliskich liniowej) przyspieszenie mocy obliczeniowej komputera faktycznie wpływa na możliwość rozwiązywania problemów. Proste symulacje pokazują, że dla „kosztownych" czasowo funkcji krótszy czas realizacji uzyskany z przyspieszenia jest i tak niewystarczający, aby pozwolić obsłużyć więcej problemów wejściowych.

Aby to dokładniej zrozumieć, spójrzmy na rysunek 3.2.

Widać na nim dwie tabelki czasów wykonania programów klasy *100n* i 2^n, w zakresie *n* od *10* do *10 000 000* w wersji podstawowej i po *1000*-krotnym zwiększeniu mocy obliczeniowej. Interesuje nas graniczny czas realizacji około milion (np. sekund) i sprawdzenie, jaki zakres wartości *n* może być obsłużony przez oba programy w tym zadanym czasie.

[5] Oznaczenia: *s* — sekunda, *w.* — wiek.

Rysunek 3.2. *Złudna wiara w moc komputera*

	Przed przyspieszeniem		Po przyspieszeniu 1000x	
n	100n	2^n	100n	2^n
10	1 000	1 024	1	1
20	2 000	1 048 576	2	1 049
30	3 000	1 073 741 824	3	1 073 742
40	4 000	1 099 511 627 776	4	10 995 116 278
50	5 000	...	5	...
100	10 000	...	10	...
1 000	100 000	...	100	...
10 000	1 000 000	...	1 000	...
100 000	10 000 000	...	10 000	...
1 000 000	100 000 000	...	100 000	...
10 000 000	1 000 000 000	...	1 000 000	...

Okazuje się, że dla programu klasy *100n* liczba możliwych do rozwiązania w zadanym czasie problemów wzrasta drastycznie (o trzy rzędy wielkości), gdy tymczasem dla drugiego algorytmu zyskujemy raptem możliwość obsłużenia mniej więcej dodatkowych 50% (praktyczny zakres *10 – 20* rozszerza się zaledwie do *10 – 30*)!

Teraz każdy sceptyk powinien przyznać należyte miejsce dziedzinie wiedzy pozwalającej uniknąć nużącego, kilkusetwiecznego oczekiwania na efekt zadziałania programu...

Aby lepiej zrozumieć mechanizmy obliczeniowe używane przy analizie złożoności algorytmów, wspólnie zgłębimy kilka charakterystycznych przykładów obliczeniowych. Nowe pojęcia związane z obliczaniem złożoności obliczeniowej algorytmów zostaną wprowadzone na reprezentatywnych przykładach, co wydaje się lepszym rozwiązaniem niż zacytowanie suchych definicji.

Jeszcze raz funkcja silnia

Do zdumiewających zalet funkcji `silnia` należy niewątpliwie mnogość zagadnień, które można za jej pomocą zilustrować. Z rozdziału poprzedniego pamiętamy jeszcze zapewne rekurencyjną definicję:

$$0! = 1,$$
$$n! = n*(n-1)!,\ n \geq 1,\ n \in N$$

Odpowiadająca tej formule funkcja w C++ ma następującą postać:

```
int silnia(int n)
{
if (n==0)
   return 1;
else
   return n*silnia(n-1);
}
```

Przyjmijmy dla uproszczenia założenie, bardzo zresztą charakterystyczne w tego typu zadaniach, że najbardziej czasochłonną operacją jest tutaj instrukcja porównania `if`. Przy takim założeniu czas, w jakim wykona się program, możemy zapisać również w postaci rekurencyjnej:

$$T(0) = t_c,$$
$$T(n) = t_c + T(n-1)\ dla\ n \geq 1.$$

Powyższe wzory należy odczytać w sposób następujący: dla danej wejściowej równej zero czas wykonania funkcji, oznaczany jako *T(0)*, równa się czasowi wykonania jednej instrukcji porównania, oznaczonej symbolicznie przez t_c. Analogiczny czas dla danych wejściowych *≥ 1* jest równy, zgodnie z formułą rekurencyjną, $T(n) = t_c + T(n-1)$.

Niestety tego typu zapis jest nam do niczego nieprzydatny — trudno np. powiedzieć od razu, ile czasu zajmie obliczenie `silnia(100)`. Widać już, że do problemu należy podejść nieco inaczej. Zastanówmy się, jak z tego układu wyliczyć *T(n)*, tak aby otrzymać jakąś funkcję nierekurencyjną

pokazującą, jak czas wykonania programu zależy od danej wejściowej n. W tym celu spróbujmy rozpisać równania:

$$\begin{aligned} T(n) &= t_c + T(n-1), \\ T(n-1) &= t_c + T(n-2), \\ T(n-2) &= t_c + T(n-3), \\ &\vdots \\ T(1) &= t_c + T(0), \\ T(0) &= t_c. \end{aligned}$$

Jeśli dodamy je teraz stronami, to powinniśmy otrzymać:

$$T(n)+T(n-1)+\cdots+T(0) = (n+1)t_c + T(n-1)+\cdots+T(0),$$

co powinno dać, po zredukowaniu składników identycznych po obu stronach równości, następującą zależność:

$$T(n) = (n+1)t_c.$$

Jest to funkcja, która w satysfakcjonującej, nieskomplikowanej formie pokazuje, w jaki sposób rozmiar danej wejściowej wpływa na liczbę instrukcji porównań wykonanych przez program — czyli de facto na czas wykonania algorytmu. Znając bowiem parametr t_c i wartość n, możemy powiedzieć dokładnie, w ciągu ilu sekund (minut, godzin, lat...) wykona się algorytm na określonym komputerze.

Tego typu rezultat dokładnych obliczeń zwykło się nazywać *złożonością praktyczną* algorytmu. Funkcja ta jest zazwyczaj oznaczana tak jak wyżej — przez T.

W praktyce rzadko interesuje nas aż tak dokładny wynik. Dla odpowiednio dużych n niewiele bowiem się zmieni, jeśli zamiast $T(n) = (n+1)t_c$ otrzymamy $T(n) = (n+3)t_c$!

Do czego zmierzam? Otóż w dalszych rozważaniach będziemy głównie szukać odpowiedzi na pytanie:

> Jaki *typ funkcji matematycznej*, występującej w zależności określającej złożoność praktyczną programu, odgrywa w niej najważniejszą rolę, wpływając najsilniej na czas wykonywania programu?


Definicja

Tę poszukiwaną funkcję będziemy zwać *złożonością teoretyczną* lub klasą algorytmu i z nią najczęściej można się spotkać przy opisach katalogowych określonych algorytmów. Funkcja ta jest najczęściej oznaczana przez O. Zastanówmy się, w jaki sposób możemy ją otrzymać.

Istnieją dwa klasyczne podejścia, prowadzące z reguły do tego samego rezultatu: albo będziemy opierać się na pewnych twierdzeniach matematycznych i je aplikować w określonych sytuacjach, albo też dojdziemy do prawidłowego wyniku metodą intuicyjną.

Wydaje mi się, że to drugie podejście jest zarówno szybkie, jak i znacznie przystępniejsze, dlatego skoncentrujemy się najpierw na nim. Popatrzmy w tym celu na tabelę 3.2 zawierającą kilka przykładów wyłuskiwania złożoności teoretycznej z równań określających złożoność praktyczną.

Wyniki zawarte w tej tabelce możemy wyjaśnić w następujący sposób: w równaniu $3n+1$ pozwolimy sobie pominąć stałą *1* i wynik nie ulegnie znaczącej zmianie. W równaniu n^2-n+1 o wiele ważniejsza jest funkcja kwadratowa niż liniowa zależność od n; podobnie w ostatnim równaniu dominuje funkcja 2^n.

Tabela 3.2. *Złożoność teoretyczna algorytmów — przykłady*

T(n)	*O*
6	O(1)
3n+1	O(n)
n^2–n+1	O(n^2)
2^n+n^2+4	O(2^n)

W algorytmice spotyka się często kilka charakterystycznych funkcji *O*, oto kilka z nich:

- Klasa *O(1)* umownie oznacza, że liczba operacji wykonywanych przez algorytm jest niezależna od rozmiarów problemu[6]. Czy taka sytuacja w przypadku algorytmów nietrywialnych w ogóle może mieć miejsce? Okazuje się, że w pewnym przybliżeniu tak, weźmy na przykład funkcje wyszukiwania oparte na metodzie transformacji kluczowej (hashing) omówionej w rozdziale 7. Teoretycznie wielkość zbioru do przeszukania nie ma znaczenia, jeśli pewna dobra funkcja *H* pozwoli nam dotrzeć do poszukiwanego rekordu w mniej więcej tym samym, skończonym czasie. Mam świadomość, że teraz brzmi to enigmatycznie, ale po lekturze wspomnianego rozdziału na pewno zgodzisz się z tym stwierdzeniem.
- Klasa *O(n)* oznacza, że algorytm wykonuje się w czasie proporcjonalnym do rozmiaru problemu. Przykładem takiego algorytmu może być przetwarzanie sekwencyjne ciągu znaków, obsługa kolejki itp. Jest to prosta zależność liniowa, gdzie każdej z *n* danych wejściowych algorytm musi poświęcić pewien czas na wykonanie swoich obliczeń.
- Złożoność typu *O(log n)* jest lepsza od liniowej[7], co arytmetycznie oznacza, że jeśli klasa problemu rośnie geometrycznie (np. o rząd wielkości, ze 100 na 1 000), to wzrost złożoności będzie arytmetyczny (tutaj dwukrotny). Jeśli *n* rośnie niezbyt szybko, to algorytm zwalnia, ale nie drastycznie. Ze złożonością logarytmiczną spotkamy się np. w algorytmie przeszukiwania posortowanej tablicy, gdzie w każdym kroku algorytmu będziemy pomijali część danych (patrz rozdział 7.).
- Klasa $O(n^2)$ jest często spotykana w rozważaniach arytmetycznych lub kombinatorycznych, gdzie mamy do czynienia z regułą „każdy z każdym". Przykładem może być dodawanie matryc o rozmiarach $n \times n$. Przez analogię dla klasy $O(n^3)$ można jako przykład podać mnożenie matryc o rozmiarach $n \times n$. Nie powinno być dla nikogo niespodzianką, że algorytmy „kwadratowe" i wyższe nadają się do wykorzystania dla raczej małych wartości *n*.
- Klasa wykładnicza $O(2^n)$ jest często przytaczana jako swego rodzaju straszak, choć w praktyce algorytmy tej klasy mogą być też używane, oczywiście jeśli zwracają wyniki w sensownym dla użytkownika czasie.

Poglądowe przykłady na funkcję *O* można by jeszcze mnożyć, ale pojęcie to jest na tyle kluczowe, że przytoczę formalną definicję matematyczną.

W tym celu odświeżmy następujące oznaczenia znane z podręczników analizy matematycznej:

- $\mathbb{N}, \mathbb{R}$ są zbiorami liczb odpowiednio naturalnych i rzeczywistych (wraz z zerem).
- $\mathbb{N}^+$ jest zbiorem liczb naturalnych dodatnich.
- Za pomocą $\mathbb{R}^+$ będziemy oznaczać zbiór liczb rzeczywistych dodatnich łącznie z zerem.
- Znak graficzny $\mapsto$ oznacza przyporządkowanie.

---

[6] Co wcale nie oznacza, że będzie mała; ta „1" jest często mylnie utożsamiana z pojedynczą instrukcją!

[7] Przypomnę, że logarytm liczby $x > 0$ o podstawie $b \neq 1$, oznaczony jako $u = \log_b x$, jest to taka liczba u spełniająca zależność $b^u = x$, np. $3 = \log_2 8$.

- Znak graficzny $\forall$ należy czytać jako „dla każdego".
- Znak graficzny $\exists$ należy czytać jako „istnieje".
- Znak graficzny $\in$ należy czytać jako „należy do" lub „należący do".
- Małe litery pisane *kursywą* na ogół oznaczają nazwy funkcji (np. *g*).
- Dwukropek zapisany po pewnym symbolu *S* należy odczytywać: *S taki, że...*

Bazując na powyższych oznaczeniach, klasę *O* dowolnej funkcji $T:\mathbb{N}\mapsto\Re^*$ możemy zdefiniować jako:

$$O(T(n))=\left\{g:T:\mathbb{N}\mapsto R^* \mid (\exists M\in\Re^+)(\exists n_0\in\mathbb{N})(\forall n\geq n_0)\left[|g(n)|\leq|M\cdot T(n)|\right]\right\}$$

Jak wynika z powyższej definicji, klasa *O* (wedle definicji jest to zbiór funkcji) ma charakter wielkości asymptotycznej, pozwalającej wyrazić w postaci arytmetycznej wielkości z góry nieznane w postaci analitycznej. Samo istnienie tej notacji pozwala na znaczne uproszczenie wielu dociekań matematycznych, w których dokładna znajomość rozważanych wielkości nie jest konieczna.

Dysponując tak formalną definicją, można łatwo udowodnić pewne „oczywiste" wyniki, np.: $|5n^3+3n^2+2|\in O(n^3)$ (można dobrać doświadczalnie takie M i n_0, dla których zawsze będzie spełnione $\{\ |5n^3+3n^2+2|\leq M\cdot|n^3|$, dla każdego $n\geq n_0\}$. W sposób zbliżony można przeprowadzić dowody wielu podobnych zadań.

Funkcja *O* jest wielkością, której można używać w równaniach matematycznych. Notacja dużego O jest czasami zwana notacją Landaua od nazwiska niemieckiego matematyka (1877 – 1938) żydowskiego pochodzenia, autora wielu prac z teorii liczb, który miał wątpliwe szczęście urodzić się w Berlinie i żyć w Niemczech w okresie kulminacji polityki nazizmu i prześladowań etnicznych.

Oto kilka własności, które mogą posłużyć do znacznego uproszczenia wyrażeń je zawierających:

$$\begin{aligned} c\cdot O(f(n)) &= O(f(n)) \\ O(f(n))+O(f(n)) &= O(f(n)) \\ O(O(f(n))) &= O(f(n)) \\ O(f(n))\cdot O(g(n)) &= O(f(n)\cdot g(n)) \\ O(f(n)\cdot g(n)) &= f(n)\cdot O(g(n)) \end{aligned}$$

Do ciekawszych należy pierwsza z powyższych własności, która znosi wpływ wszelkich współczynników o wartościach stałych. Własność ta pomoże nam dalej zrozumieć, dlaczego w literaturze mówi się, że „algorytm A jest klasy *O(log N)*", a nie „*O(log₂N)*".

Ktoś o bardzo radykalnym podejściu do wszelkich sztucznych założeń, mających ułatwić wyliczenie pewnych normalnie skomplikowanych zagadnień, mógłby zakwestionować przyjmowanie podstawy *2* za punkt odniesienia, zapytując się przykładowo: „a dlaczego nie *2,5* lub *3*"? Pozornie takie postawienie sprawy wydaje się słuszne, ale na szczęście tylko pozornie!

Najpierw zauważmy, że używanie dwójki jako podstawy obliczeń jest pewną manierą związaną z rozpowszechnieniem systemu dwójkowego. Z tego powodu zakładamy, że np. rozmiar tablicy jest wielokrotnością liczby *2* i następnie na podstawie takich założeń częstokroć wyliczana jest złożoność praktyczna i z niej dedukowana jego klasa, czyli funkcja *O*.

Przypomnijmy sobie jednak elementarny wzór podający zależność pomiędzy logarytmami o różnych podstawach:

$$\log_a N=\frac{\log_b N}{\log_b a}=\log_b N\cdot\frac{1}{\log_b a}.$$

Widać wyraźnie, że logarytmy o odmiennych podstawach (a i b) różnią się pomiędzy sobą tylko pewnym współczynnikiem stałym, który zostanie „pochłonięty" przez O na podstawie własności:

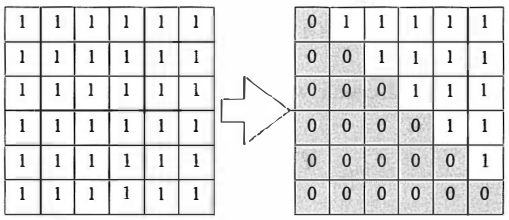
$$c \cdot O\big(f(n)\big) = O\big(f(n)\big).$$

Popatrzmy jeszcze na inny aspekt stosowania O-notacji. Załóżmy, że pewien algorytm A został wykonany w dwóch wersjach: $W1$ i $W2$, charakteryzujących się złożonością praktyczną odpowiednio: $100\ log_2 N$ i $10N$. Na podstawie uprzednio poznanych własności możemy szybko określić, że $W1 \in O(logN)$, $W2 \in O(N)$, czyli $W1$ jest lepszy od $W2$. Niestety ktoś szczególnie złośliwy mógłby się uprzeć, że jednak algorytm $W2$ jest lepszy, bowiem dla np. $N = 2$ mamy $100\ log_2 2 > 10 \cdot 2 ...$ Wobec takiego stwierdzenia nie należy wpadać w panikę, tylko wziąć do ręki *odpowiednio duże* N, dla którego algorytm $W1$ okaże się jednak lepszy od $W2$! Nie należy bowiem zapominać, że O-notacja ma charakter asymptotyczny i jest prawdziwa dla „odpowiednio dużych wartości N".

Zerowanie fragmentu tablicy

Rozwiążemy teraz następujący problem: jak wyzerować fragment tablicy (tzn. macierzy) poniżej przekątnej (wraz z nią)? Tę ideę przedstawia rysunek 3.3.

Rysunek 3.3.
Koszt zerowania tablicy



Funkcja wykonująca to zadanie jest bardzo prosta:

```
int tab[n][n];
void zerowanie()
{

int i,j;
i=0;                    // ta
while (i<n)             // tc
  {
   j=0;                 // ta
   while (j<=i)         // tc
     {
      tab[i][j]=0;      // ta
      j=j+1;            // ta
     }
   i=i+1;               // ta
   }
}
```

Oznaczenia:

- t_a — czas wykonania instrukcji *przypisania*;
- t_c — czas wykonania instrukcji *porównania*.

Do dalszych rozważań niezbędne będzie zrozumienie funkcjonowania pętli typu `while`:

```
i=1;
while (i<=n)
{
```

```
    instrukcje;
    i=i+1;
}
```

Jej działanie polega na wykonaniu n razy instrukcji zawartych pomiędzy nawiasami klamrowymi, warunek natomiast jest sprawdzany n+1 razy[8].

Korzystając z powyższej uwagi oraz informacji zawartych w liniach komentarza, możemy napisać:

$$T(n) = t_c + t_a + \sum_{i=1}^{N}\left(2t_a + 2t_c + \sum_{j=1}^{i}\left(t_c + 2t_a\right)\right).$$

Po usunięciu sumy z wewnętrznego nawiasu otrzymamy:

$$T(n) = t_c + t_a + \sum_{i=1}^{N}\left(2t_a + 2t_c + i\left(t_c + 2t_a\right)\right). \quad (*)$$

Przypomnijmy jeszcze użyteczny wzór na sumę szeregu liczb naturalnych od *1* do *N*:

$$1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}.$$

Po jego zastosowaniu w równaniu (*) otrzymamy:

$$T(n) = t_c + t_a + 2N\left(t_a + t_c\right) + \frac{N(N+1)}{2}\left(t_c + 2t_a\right).$$

Ostateczne uproszczenie wyrażenia powinno nam dać:

$$T(n) = t_a\left(1 + 3N + N^2\right) + t_c\left(1 + 2{,}5t_c + \frac{N^2}{2}\right);$$

co sugeruje od razu, że analizowany program jest klasy $O(n^2)$.

Nie było to przyjemne, prawda? A problem wcale nie należał do specjalnie złożonych. Nie zrażajmy się jednak trudnym początkiem, wkrótce okaże się, że można było zrobić to znacznie prościej! Do tego potrzebna nam będzie odrobina wiedzy teoretycznej, dotyczącej rozwiązywania równań rekurencyjnych. Poznamy ją szczegółowo po przerobieniu kolejnego przykładu zawierającego pewną pułapkę, której istnienie trzeba niestety co najmniej raz sobie uświadomić.

Wpadamy w pułapkę

Zadania z dwóch poprzednich przykładów charakteryzowała istotna cecha: czas wykonania programu nie zależał od wartości, jakie przybierała dana, lecz tylko od jej rozmiaru. Niestety nie zawsze tak jest! Takiemu właśnie przypadkowi poświęcone jest kolejne zadanie obliczeniowe. Jest to fragment większego programu, którego rola nie jest dla nas istotna w tym miejscu. Załóżmy, że otrzymujemy ten wyrwany z kontekstu fragment kodu i musimy się zająć jego analizą:

```
const int N=10;
int t[N];
funkcja_ad_hoc()
   {
   int k,i;
   int suma=0;                    // t_a
```

[8] Warto zauważyć, że istniejące w C++ pętle łatwo dają się sprowadzić do odmiany pętli zacytowanej powyżej.

```
while (i<N)                    // t_c
   {
   while (j<=t[i])             // t_c
      {
      suma=suma+2;             // t_a
      j=j+1;                   // t_a
      }
   i=i+1;                      // t_a
   }
}
```

Uprośćmy nieco problem, zakładając, że:

- Najbardziej czasochłonne są instrukcje porównania, wszelkie inne zaś ignorujemy jako niemające większego wpływu na czas wykonania programu.
- Zamiast pisać explicite t_c, wprowadzimy pojęcie czasu jednostkowego wykonania instrukcji, oznaczając go przez 1.

Niestety jedno zasadnicze utrudnienie pozostanie aktualne: nie znamy zawartości tablicy, a zatem nie wiemy, ile razy wykona się wewnętrzna pętla `while`! Popatrzmy, jak możemy sobie poradzić w tym przypadku:

$$T(n) = t_c + \sum_{i=1}^{N}\left(t_c + \sum_{j=1}^{t[i]} t_c\right), \qquad (*)$$

$$T(n) = t_c + Nt_c + \sum_{i=1}^{N} t[i]t_c, \qquad (**)$$

$$T(n) = t_c + Nt_c + Nt[i]t_c,$$

$$T(n) = t_c\left(1 + N + Nt[i]\right),$$

$$T(n) \approx \max\left(N, Nt[i]\right).$$

Początek jest klasyczny: zewnętrzna suma od *1* do *N* z równania (*) zostaje zamieniona na *N*-krotny iloczyn swojego argumentu. Podobny trik zostaje wykonany w równaniu (**), po czym możemy już spokojnie zająć się grupowaniem i upraszczaniem. Czas wykonania programu jest proporcjonalny do większej z liczb: *N* i *Nt[i]* i tylko tyle możemy na razie stwierdzić. Niestety, kończąc w tym miejscu rozważania, wpadlibyśmy w pułapkę. Naszym problemem jest bowiem nieznajomość zawartości tablicy, a ta jest potrzebna do otrzymania ostatecznego wyniku! Nie możemy przecież zastosować funkcji matematycznej do wartości nieokreślonej.

Nasze obliczenia doprowadziły zatem do momentu, w którym zauważamy brak pełnej informacji o rozważanym problemie. Gdybyśmy przykładowo wiedzieli, że o tym fragmencie programu, w którym pracuje nasza funkcja, można z dużym prawdopodobieństwem powiedzieć, iż tablica wypełniona jest głównie zerami, to nie byłoby w ogóle problemu! Nie mamy jednak żadnej pewności, czy rzeczywiście zajdzie taka sytuacja. Jedynym rozwiązaniem wydaje się zwrócenie do jakiegoś matematyka, aby ten — po przyjęciu dużej liczby założeń — przeprowadził analizę statystyczną zadania i doprowadził do ostatecznego wyniku w satysfakcjonującej nas postaci.

Różne typy złożoności obliczeniowej

Rozważmy ponownie problem: należy sprawdzić, czy pewna liczba `x` znajduje się w tablicy o rozmiarze `n`. Został już on rozwiązany w rozdziale 2., spróbujmy teraz napisać iteracyjną wersję tej samej procedury. Nie jest to czynność szczególnie skomplikowana i sprowadza się do napisania następującego programu:

szukaj.cpp

```
const int n=10;
int tab[n]={1,2,3,2,-7,44,5,1,0,-3};
int szukaj(int tab[n], int x)
   {
   int pos=0;
   while ((pos<n) && (tab[pos]!=x)) pos++;
   if (pos<n)
      return pos; // element znaleziony
   else
      return -1;  // porażka poszukiwań
   }
int main()
{
cout << szukaj(tab,7) <<endl; // wynik = -1
cout << szukaj(tab,5) <<endl; // wynik = 6
}
```

Idea tego algorytmu polega na sprawdzeniu, czy w badanym fragmencie tablicy lewy skrajny element jest poszukiwaną wartością `x`. Wywołując procedurę w następujący sposób: `szukaj(tab,x)`, powodujemy przebadanie całej tablicy o rozmiarze `n`. Co można powiedzieć o złożoności obliczeniowej tego algorytmu, przyjmując jako kryterium liczbę porównań wykonanych w pętli `while`? Na tak sformułowane pytanie można się niestety tylko obruszyć i mruknąć: „To zależy, gdzie znajduje się `x`"! Istotnie mamy do czynienia z co najmniej dwoma skrajnymi przypadkami:

- Znajdujemy się w komórce `tab[0]`, czyli $T(n) = 1$ i trafiamy na tzw. *najlepszy przypadek*.
- W poszukiwaniu `x` przeglądamy całą tablicę, czyli $T(n) = n$ i trafiamy na tzw. *najgorszy przypadek*.

Jeśli na jedno precyzyjne pytanie: „Jaka jest złożoność obliczeniowa algorytmu liniowego przeszukiwania tablicy n-elementowej?" otrzymujemy dwie odpowiedzi, obarczone klauzulami *„jeśli"*, *„w przypadku gdy..."*, to jedno jest pewne: odpowiedzi na pytanie ciągle nie mamy!

Błąd tkwił oczywiście w pytaniu, które powinno uwzględniać konfigurację danych, a ma ona w przypadku przeszukiwania tablicy kluczowe znaczenie. Proponowane odpowiedzi mogą być zatem następujące: rozważany algorytm ma w najlepszym przypadku złożoność praktyczną równą $T(n) = 1$, a w najgorszym przypadku — $T(n) = n$. Ponieważ jednak życie toczy się raczej równomiernie i nie balansuje pomiędzy skrajnościami (co jest dość prowokacyjnym stwierdzeniem, ale przyjmijmy chwilowo, że jest to prawda), warto byłoby poznać również odpowiedź na pytanie: jaka jest *średnia* wartość $T(n)$ tego algorytmu? Należy ono do gatunku nieprecyzyjnych, jest zatem stworzone dla statystyka. Nie pozostaje nam nic innego, jak przeprowadzić analizę statystyczną omawianego algorytmu.

Oznaczmy przez p prawdopodobieństwo, że x znajduje się w tablicy *tab*, i przypuśćmy, że jeśli istotnie x znajduje się w tablicy, to wszystkie miejsca są jednakowo prawdopodobne.

Oznaczmy również przez $D_{n,i}$ (gdzie $0 \le i < n$) zbiór danych, dla których x znajduje się na i-tym miejscu tablicy i przez $D_{n,n}$ zbiór danych, gdzie x jest nieobecne. Wedle przyjętych wyżej oznaczeń możemy napisać, że:

$$P(D_{n,i}) = \frac{p}{n} \text{ i } P(D_{n,n}) = 1 - p.$$

Koszt algorytmu oznaczmy klasycznie przez T, tak więc:

$$T(D_{n,i}) = i \text{ oraz } T(D_{n,n}) = n.$$

Otrzymujemy zatem wyrażenie:

$$T_{średnie} = \sum_{i=0}^{N} P(D_{n,i})T(D_{n,i}) = (1-p)n + \sum_{i=0}^{n-1} i\frac{p}{n} = (1-p)n + (n+1)\frac{p}{2}.$$

Przykładowo: wiedząc, że x na pewno znajduje się w tablicy ($p = 1$), możemy od razu napisać:

$$T_{średnie} = (1-1)n + \frac{(n+1)}{2} = \frac{(n+1)}{2}.$$

Zdefiniowaliśmy zatem trzy podstawowe typy złożoności obliczeniowej (dla przypadków: *najgorszego*, *najkorzystniejszego* i *średniego*), warto teraz zastanowić się nad użytecznością praktyczną tych pojęć. Z matematycznego punktu widzenia te trzy określenia definiują w pełni zachowanie się algorytmu, ale czy aby na pewno robią to dobrze?

W katalogowych opisach algorytmów najczęściej mamy do czynienia z rozważaniami na temat przypadku *najgorszego* — tak aby wyznaczyć sobie pewną górną granicę, której algorytm na pewno nie przekroczy (jest to informacja najbardziej użyteczna dla programisty).

Przypadek *najkorzystniejszy* ma podobny charakter, dotyczy jednak progu dolnego czasu wykonywania programu.

Widzimy, że pojęcia złożoności obliczeniowej programu w przypadkach *najlepszym* i *najgorszym* mają sens nie tylko matematyczny, lecz dają programiście pewne granice, w których może on go umieścić. Czy podobnie możemy rozpatrywać przypadek *średni*?

Jak łatwo zauważyć, wyliczenie przypadku średniego (inaczej to określając: *typowego*) nie jest łatwe i wymaga założenia szeregu hipotez dotyczących możliwych konfiguracji danych. Między innymi musimy umówić się co do definicji zbioru danych, z którym program ma do czynienia — niestety zazwyczaj nie jest to ani możliwe, ani nie ma żadnego sensu! Programista dostający informację o średniej złożoności obliczeniowej programu powinien być zatem świadomy tych ograniczeń i nie brać tego parametru za informację wzorcową.

Nowe zadanie: uprościć obliczenia!

Nie sposób pominąć faktu, że wszystkie nasze dotychczasowe zadania były dość skomplikowane rachunkowo, a tego leniwi ludzie (czytaj: programiści) nie lubią. Jak zatem postępować, aby wykonać tylko te obliczenia, które są naprawdę niezbędne do otrzymania wyniku? Otóż warto zapamiętać następujące sztuczki, które znacznie ułatwią nam to zadanie, pozwalając niejednokrotnie natychmiastowo określić poszukiwany wynik:

- W analizie programu zwracamy uwagę tylko na najbardziej czasochłonne operacje (np. poprzednio były to instrukcje porównań).
- Wybieramy jeden wiersz programu znajdujący się w najgłębiej położonej instrukcji iteracyjnej (pętle w pętlach, a te jeszcze w innych pętlach...), a następnie obliczamy, ile razy się on wykona. Z tego wyniku dedukujemy złożoność teoretyczną.

Pierwszy sposób był już wcześniej stosowany. Aby wyjaśnić nieco szerzej drugą metodę, proponuję przestudiować poniższy fragmentu programu:

```
while (i<N)
   {
   while (j<=N)
      {
      suma=suma+2;
      j=j+1;
      }
   }
```

Wybieramy instrukcję `suma=suma+2` i obliczamy w prosty sposób, iż wykona się ona $\frac{N(N+1)}{2}$ razy. Wnioskujemy, że ten fragment programu ma złożoność teoretyczną równą $O(n^2)$.

Analiza programów rekurencyjnych

Większość programów rekurencyjnych nie da się, niestety, rozważyć przy użyciu poznanej wcześniej metody. Istotnie zastosowana tam metoda rozwiązywania równania rekurencyjnego, polegająca na rozpisaniu jego składników i dodaniu stronami układu równań, nie zawsze się sprawdza. U nas doprowadziła ona do sukcesu, tzn. do uproszczenia obliczeń — niestety, zazwyczaj równania potraktowane w ten sposób jeszcze bardziej się komplikują.

W tym paragrafie przedstawiona zostanie metoda mająca charakter o wiele ogólniejszy. Ma ona swoje uzasadnienie matematyczne, którego z powodu jego skomplikowania nie będę przedstawiał. Osoby szczególnie zainteresowane stroną matematyczną powinny dotrzeć bez kłopotu do odpowiedniej literatury (patrz uwagi zamieszczone we wstępie rozdziału).

Terminologia i definicje

Lektura kilku następnych paragrafów wymaga od nas poznania terminologii, którą będziemy się dość często posługiwać. Pomimo ich „groźnego" wyglądu zrozumienie poniższych definicji nie powinno Czytelnikowi sprawić szczególnych kłopotów.

Szereg rekurencyjny liniowy **SRL** jest to szereg o następującej postaci:

$$X_{n+r,n\geq 0} = \sum_{i=1}^{r} a_i X_{n+r-i} + u(n,m).$$

Oznaczenia:

u(n,m) — nierekurencyjna reszta równania, będąca wielomianem stopnia *m* i zmiennej *n*.

Przykładowo:

- Jeśli $u(n,m) = 3n+1$, to mamy wielomian stopnia *pierwszego*.
- Jeśli $u(n,m) = 2$, to jest to wielomian stopnia *zerowego*.

Uwagi:

- Współczynniki a_i są dowolnymi liczbami rzeczywistymi.
- r jest liczbą całkowitą.

Skomplikowaną postacią tego wzoru nie należy się przejmować, jest to po prostu sformalizowany zapis ogólnego równania rekurencyjnego, podany raczej gwoli formalności niż w jakimś praktycznym celu.


Definicja

Równanie charakterystyczne **RC** jest to wielomian sztucznie stworzony na podstawie równania rekurencyjnego, powstały wg wzoru:

$$R(x) = X^r - \sum_{i=1}^{r} a_i x^{r-i}.$$

Równanie to można rozwiązać, otrzymując rozkład postaci:

$$R(x) = \prod_{i=1}^{p} (x - \lambda_i)^{m_i}.$$

Przykład: $SRL = x_n - 3x_{n-1} + 2x_{n-2} = 0$ daje $R(x) = x^2 - 3x + 2 = (x-1)(x-2)$.


Definicja

Otrzymane powyżej współczynniki λ_i posłużą do skonstruowania tzw. ***rozwiązania ogólnego*** **RO** liniowego równania rekurencyjnego:

$$RO = \sum_{i=1}^{p} P_i \lambda_i.$$

Dodatkowo będziemy potrzebować tzw. ***rozwiązania szczególnego*** **RS** liniowego równania rekurencyjnego.

Postać **RS** zależy od formy, jaką przybiera reszta $u(n,m)$. Poszczególne przypadki rozpisane są poniżej.

- Jeśli $u(n,m) = 0$, to $RS = 0$.
- Jeśli $u(n,m)$ jest wielomianem stopnia m i zmiennej n oraz *1* (jeden) *nie jest* rozwiązaniem RC, wówczas $RS = Q(n,m)$, gdzie $Q(n,m)$ jest pewnym wielomianem stopnia m i zmiennej n o współczynnikach nieznanych (do odnalezienia).
- Jeśli $u(n,m)$ jest wielomianem stopnia m i zmiennej n oraz *1* jest rozwiązaniem RC, wtedy $RS = n^p Q(n,m)$, gdzie p jest stopniem pierwiastka.

  Przykładowo: jeśli jedynka jest pierwiastkiem pojedynczym RC, to $p = 1$, jeśli pierwiastkiem podwójnym, to $p = 2$ itd.
- Jeśli $u(n,m) = \alpha^n$ i α *nie jest* rozwiązaniem RC, wtedy:

  $$RS = c\alpha^n.$$
- Jeśli $u(n,m) = \alpha^n$ i α *jest* pierwiastkiem stopnia p RC, wtedy:

  $$RS = c\alpha^n n^p.$$
- Jeśli $u(n,m) = \alpha^n W(n,m)$ i α *nie jest* rozwiązaniem RC (tradycyjnie już $W(n,m)$ jest pewnym wielomianem stopnia m i zmiennej n), będziemy wówczas mieli:

  $$RS = \alpha^n S(n,m),$$

  gdzie $S(n,m)$ jest pewnym wielomianem stopnia m i zmiennej n.


Uwaga

Występujące po prawej stronie wzorów wielomiany i stałe mają charakter zmiennych, które należy odnaleźć!

Rozwiązaniem równania rekurencyjnego jest suma obu równań: *ogólnego* i *szczególnego*.

Cały ten bagaż wzorów był naprawdę niezbędny! Dla zilustrowania metody rozwiążemy proste zadanie.

Ilustracja metody na przykładzie

Spójrzmy jeszcze raz na przykład ze strony 53 dotyczący funkcji silnia. Otrzymaliśmy wtedy następujące równanie:

$$T(0) = 1,$$
$$T(n) = 1 + T(n-1).$$

Spróbujmy je rozwiązać nowo poznaną metodą.

- **ETAP 1.** Poszukiwanie równania charakterystycznego:

  Z postaci ogólnej $SRL = T(n)–T(n–1)$ wynika, że $RC = x–1$.

- **ETAP 2.** Pierwiastek równania charakterystycznego:

  Jest to oczywiście $r = 1$.

- **ETAP 3.** Równanie ogólne:

  $RO = Ar^n$, gdzie A jest jakąś stałą do odnalezienia. Ponieważ $r = 1$, to $RO = A$ (1 podniesione do dowolnej potęgi da nam oczywiście 1). Stałą A wyliczymy dalej.

- **ETAP 4.** Poszukiwanie równania szczególnego:

  Wiemy, że $u(n,m) = 1$ (jeden jest wielomianem stopnia zero!). Ponadto 1 jest pierwiastkiem pierwszego stopnia równania charakterystycznego. Tak więc:
  $S = n^p c = n \cdot c$.

  Pozostaje nam jeszcze do odnalezienia stała c. Wiemy, że RS musi spełniać pierwotne równanie rekurencyjne, zatem po podstawieniu go jako $T(n)$ otrzymamy:

$$\begin{aligned} n \cdot c &= 1 + (n-1)c, \\ n \cdot c &= 1 + n \cdot c - c, \\ c &= 1. \end{aligned}$$

- **ETAP 5.** Poszukiwanie ostatecznego rozwiązania:

  Wiemy, że ostatecznym rozwiązaniem równania jest suma RO i RS:
  $T(n) = RO+RS = A+n \cdot c = A+n$. Stałą A możemy z łatwością wyliczyć poprzez podstawienie przypadku elementarnego:

$$\begin{aligned} T(0) &= 1, \\ 1 &= A + 0, \\ A &= 1. \end{aligned}$$

  Po tych karkołomnych wyliczeniach otrzymujemy: $T(n) = n+1$.

  Jest ono identyczne z poprzednim rozwiązaniem[9].

Metoda równań charakterystycznych jest, jak widać, bardzo elastyczna. Pozwala ona na szybkie określenie złożoności algorytmicznej nawet dość rozbudowanych programów. Są oczywiście zadania wymagające interwencji matematyka, ale zdarzają się one rzadko i dotyczą zazwyczaj programów rekurencyjnych o nikłym znaczeniu praktycznym.

Rozkład logarytmiczny

Z poprzedniego rozdziału pamiętamy zapewne zadanie poświęcone przeszukiwaniu binarnemu. Jedną z możliwych wersji funkcji[10] wykonującej to zadanie jest:

```
int binary_search(int *tab, int x, int left, int right)
{
   if (left==right)
   if (t[left]==x)
      return left;
      else                    // element znaleziony
         return -1;           // element nieodnaleziony
   else
      {
```

[9] Jeśli dwie metody prowadzą do takiego samego, prawidłowego wyniku, to istnieje duże prawdopodobieństwo, iż obie są dobre.

[10] Nieco innej niż poprzednio zaproponowana.

```
        int mid=(left+right)/2;
        if (tab[mid]==x)
           return mid;        // element znaleziony!
        else
           if (x<tab[mid])
           return binary_search(tab,x,left,mid);
        else
           return binary_search(tab,x,mid,right);
        }
}
```

Jaka jest złożoność obliczeniowa tej funkcji? Analiza liczby instrukcji porównań prowadzi nas do następujących równości:

$$T(1) = 1 + 1 = 2,$$
$$T(n) = 1 + 1 + T\left(\frac{n}{2}\right) = 2 + T\left(\frac{n}{2}\right).$$

Widać już, że powyższy układ ma się nijak do podanej poprzednio metody. W określeniu równania charakterystycznego przeszkadza nam owo dzielenie n przez *2*. Otóż można z tej pułapki wybrnąć, np. przez podstawienie $n = 2p$, ale ciąg dalszy obliczeń będzie dość złożony. Na szczęście matematycy zrobili w tym miejscu programistom miły prezent: bez żadnych skomplikowanych obliczeń można określić złożoność tego typu zadań, korzystając z kilku gotowych reguł. Prezent ten jest tym bardziej cenny, że zadania o rozkładzie podobnym do powyższego występują bardzo często w praktyce programowania. Przed ostatecznym jego rozwiązaniem musimy zatem poznać jeszcze kilka wzorów matematycznych, ale obiecuję, że na tym już będzie koniec, jeśli chodzi o matematykę „wyższą".

Załóżmy, że ogólna postać otrzymanego układu równań rekurencyjnych przedstawia się następująco:

$$T(1) = 1,$$
$$T(n) = aT\left(\frac{n}{b}\right) + d(n).$$

(Przy założeniu, że $n \geq 2$ oraz a i b są pewnymi stałymi).

W zależności od wartości a, b i $d(n)$ otrzymamy różne rozwiązania zgrupowane w tabeli 3.3.

Tabela 3.3. *Analiza przeszukiwania binarnego*

	Klasa algorytmu	
$a > d(b)$	$T(n) \in O\left(n^{\log_b a}\right)$	
$a < d(b)$	$T(n) \in O\left(n^{\log_b d(b)}\right)$	gdyd(n) = n^{α} toT(n) $\in O\left(n^{\alpha}\right) = O(d(n))$
$a = d(b)$	$T(n) \in O\left(n^{\log_b d(b)} \log_b n\right)$	gdyd(n) = n^{α} toT(n) $\in O\left(n^{\alpha} \log_b n\right)$

Wzory te są wynikiem dość skomplikowanych wyliczeń bazujących na następujących założeniach:

- n jest potęgą b, co pozwala wykonać podstawienie $n = b^k$ sprowadzające równanie nieliniowe do równania $T(b^k) = aT(b^{k-1}) + d(b^k)$. Podstawiając ponadto $t_k = T(b^k)$, otrzymujemy równanie liniowe $t_k = at_{k-1} + d(b^k)$ z warunkiem początkowym $t_0 = 1$. Dyskusja wyników tego równania prowadzi do wniosków końcowych, przedstawionych w tabeli 3.3.
- Funkcja $d(n)$ musi spełniać następującą własność: $d(xy) = d(x)d(y)$ (np. $d(n) = n^2$ spełnia tę własność, a $d(n) = n-1$ już nie).

Pomimo tych ograniczeń okazuje się, iż bardzo duża klasa równań może być dzięki powyższym wzorom z łatwością rozwiązana. Spróbujmy dla przykładu skończyć zadanie dotyczące przeszukiwania binarnego. Jak pamiętamy, otrzymaliśmy wówczas następujące równania:

$$T(1) = 2,$$
$$T(n) = 2 + T\left(\frac{n}{2}\right).$$

Patrząc na zestaw podanych powyżej wzorów, widzimy, że nie jest on zgodny z wzorcem podanym wcześniej. Nic nie stoi jednak na przeszkodzie, aby za pomocą prostego podstawienia doprowadzić do postaci, która będzie nas satysfakcjonowała:

$$U(n) = T(n) - 1 \quad \Leftrightarrow \quad U(1) = T(1) - 1 = 1,$$
$$T(n) - 1 = 1 + T\left(\frac{n}{2}\right) \quad \Leftrightarrow \quad U(n) = U\left(\frac{n}{2}\right) + 1.$$

Identyfikujemy wartości stałych: $a = 1$, $b = 2$ i $d(n) = 1$, co pozwala nam zauważyć, iż zachodzi przypadek trzeci: $a = d(b)$. Poszukiwany wynik ma zatem postać:

$$U(n) \in O\left(n^{\log_2 1} \log_2 n\right) = O\left(n^0 \log_2 n\right) = O\left(\log_2 n\right).$$

Zamiana dziedziny równania rekurencyjnego

Pewna grupa równań charakteryzuje się zdecydowanie nieprzyjemnym wyglądem i nijak nie odpowiada podanym uprzednio wzorom i metodom. Czasem jednak zwykła zmiana dziedziny powoduje, iż rozwiązanie pojawia się niemal natychmiastowo. Przeanalizujmy następujący przykład:

$$a_n = 3^2_{n-1} \text{ dla } n \geq 1,$$
$$a_0 = 1.$$

Równanie nie jest zgodne z żadnym poznanym wcześniej schematem. Podstawmy jednak $b_n = log_2 a_n$ i zlogarytmujmy obie strony równania:

$$\log_2 a_n = \log_2\left(3^2_{n-1}\right),$$

otrzymując w efekcie:

$$\left.\begin{aligned} b_n &= 2b_{n-1} + 3\log_2 3 \\ b_0 &= 0 \end{aligned}\right\} \textit{równanie liniowe.}$$

Zadanie w tej postaci nadaje się już do rozwiązania! Po dokonaniu niewielkich obliczeń możemy otrzymać: $b_n = (2_n - 1)log_2 3$, co ostatecznie daje $a_n = 2^{\left(2^n-1\right)\log_2 3} = 3^{2^n-1}$.

Funkcja Ackermanna, czyli coś dla smakoszy

Gdyby małe dzieci znały się odrobinę na informatyce, to rodzice na pewno straszyliby je nie kominiarzem, ale funkcją Ackermanna. Jest to wspaniały przykład ukazujący, jak pozornie niegroźna z wyglądu funkcja rekurencyjna może być kosztowna w użyciu. Spójrzmy na listing:

A.cpp

```
Int A(int n, int p)
{
if (n==0)
   return 1;
```

```
if ((p==0)&&(n>=1))
   if (n==1)
      return 2;
   else
      return n+2;
if ((p>=1)&&(n>=1))
   return A(A(n-1,p),p-1);
}
int main()
{
cout << "A(3,4)="<<A(3,4) <<endl;
}
```

Pytanie dotyczące tego programu brzmi: jaki jest powód komunikatu *Stack overflow!* (przepełnienie stosu) podczas próby jego wykonania? Komunikat ten jednoznacznie sugeruje, iż podczas wykonywania programu nastąpiła znaczna liczba wywołań funkcji Ackermanna. Jak znaczna, okaże się już za chwilę.

Pobieżna analiza funkcji A prowadzi do następującego spostrzeżenia:

$$\forall n \geq 1,\ A(n,1) = A(A(n-1,1),0) = A(n-1,1) + 2,$$

co daje natychmiast

$$\forall n \geq 1,\ A(n,1) = 2n.$$

Analogicznie dla *2* otrzymamy:

$$\forall n \geq 1,\ A(n,2) = A(A(n-1,2),1) = 2A(n-1,2),$$

co z kolei pozwala nam napisać, że:

$$\forall n \geq 1,\ A(n,2) = 2^n.$$

Z samej definicji funkcji Ackermanna możemy wywnioskować, że:

$$\forall n \geq 1\, A(n,3) = A(A(n-1,3),2) = 2^{A(n-1,3)} \text{ oraz } A(0,3) = 1.$$

Na bazie tych równań możliwe jest rekurencyjne udowodnienie, że:

$$\forall n \geq 1,\ A(n,3) = 2^{\overset{n}{\overbrace{2^{\cdot^{\cdot^{2}}}}}}.$$

Nieco gorsza sytuacja występuje w przypadku *A(n,4)*, gdzie trudno jest podać wzór ogólny. Proponuję spojrzeć na kilka przykładów liczbowych:

$$\begin{aligned} A(1,4) &= 2. \\ A(2,4) &= 2^2 = 4. \\ A(3,4) &= 2^{2^{2^2}} = 65536. \\ A(4,4) &= 2^{\overset{65536}{\overbrace{2^{\cdot^{\cdot^{2}}}}}}. \end{aligned}$$

Wyrażenie w formie liczbowej *A(4,4)* jest — co może będzie zbyt dyplomatycznym stwierdzeniem — niezbyt oczywiste, nieprawdaż? W przypadku funkcji Ackermanna trudno jest nawet nazwać jej klasę — stwierdzenie, że zachowuje się ona wykładniczo, może zabrzmieć jak kpina!

Złożoność obliczeniowa to nie religia!

Wbrew pozorom złożoność obliczeniowa nie zawsze stanowi istotne kryterium decyzyjne, gdy piszemy programy lub dobieramy algorytmy „z biblioteki".

Oto kilka sytuacji, gdy możemy nieco odprężyć się w tej kwestii:

- Zdarza się, że pisany jest algorytm, który ma za zadanie coś obliczyć kilka razy i jego optymalizacja mija się z celem. Taniej będzie spróbować go uruchomić lub ekstrapolować czas wykonania i ocenić, czy jest on do zaakceptowania. Oczywiście taka sytuacja ma miejsce dla funkcji (procedur), których czas wykonania i tak mieści się w wymaganiach wydajnościowych aplikacji.
- Prostota: czasem optymalne algorytmy są zupełnie niezrozumiałe na pierwszy rzut oka i stopień ich skomplikowania łatwo prowadzi do błędów logicznych, nawet spowodowanych trywialną pomyłką w stylu określenia warunków brzegowych lub logicznych w pętlach.
- Precyzja, tak potrzebna np. w algorytmach numerycznych, może być ważniejsza niż klasa algorytmu lub jego fragmentu.
- Notacja dużego O tak naprawdę odgrywa rolę dla takich danych wejściowych, dla których czas wykonania może zbliżać się do bardzo dużych liczb (nie wspominając już o nieskończoności, która jest BARDZO dużą liczbą...). W praktyce może to oznaczać, że algorytm „kwadratowy" będzie w pewnych konfiguracjach lepszy od „logarytmicznego"!

Jako zasadę generalną przyjmijmy zatem zdrowy rozsądek, który uchroni nas przed popadaniem w zbyt akademickie decyzje. Programy wykonują się bowiem nie w teoretycznych środowiskach albo modelach, lecz w prawdziwych komputerach. Co z tego, że przyspieszymy czas realizacji wyliczeń finansowych 100-krotnie, np. ze 100 ms do 1 ms, gdy na koniec czas zapisu wyniku do bazy danych i tak wyniesie 2 – 3 s?

Techniki optymalizacji programów

Załóżmy, że stworzyliśmy program i wydaje nam się, że działa on zbyt wolno[11]. Jak możemy się zorientować, gdzie tkwi problem wydajnościowy? Podstawowym problemem przy optymalizacji jest zlokalizowanie tego miejsca, które warto optymalizować. Jeśli nasz program jest w pełni funkcjonalny i dysponujemy danymi, które są reprezentatywne (np. baza danych wypełniona tysiącami rekordów, a nie kilkoma fałszywymi wpisani), to możemy przystąpić do pomiaru czasów realizacji modułów lub procedur. To, gdzie mierzymy, zależy już od architektury i trzeba postępować ze świadomością, jakie są przepływy danych i wąskie gardła.

Jednym z prostszych sposobów jest wykorzystanie metod śledzenia czasów realizacji procedur poprzez wbudowanie do nich dodatkowych liczników. Ważne jest jednak, aby te liczniki nie obciążały samych algorytmów, zatem powinny one być zrealizowane z wykorzystaniem mechanizmów systemowych (czas odczytywany z systemu, a nie liczony przez program).

Gdy już posiadamy wyniki naszych pomiarów, to postępujemy wg zasady Pareto, czyli koncentrujemy naszą uwagę na najbardziej obciążonych lub najczęściej wywoływanych fragmentach kodu (np. pętlach, procedurach).

[11] W tym punkcie nie interesuje mnie kwestia *rozmiaru* kodu, gdyż w obecnych czasach to kryterium trochę straciło na znaczeniu.

Możliwe są teraz dwa podejścia:

- Analizujemy algorytmy według znanych nam reguł (ten rozdział!) i tutaj szukamy przysłowiowej „dziury w całym".
- Optymalizujemy kod, wyszukując w nim fragmenty, w których można dokonać uproszczeń, np. przez zastosowanie jednej ze sztuczek:
  - Wykorzystanie pamięci podręcznej (ang. *cache*) : liczymy coś jeden raz i zapisujemy do pamięci wykorzystywanej przez cały czas życia programu, oferując, np. poprzez zmienne statyczne, widzialność wyników w całym programie lub modułach.
  - Eliminujemy operacje czasochłonne, np. bazodanowe, wykonując je wówczas, gdy jest to naprawdę potrzebne.
  - Zastępujemy kosztowne czasowo instrukcje mnożenia lub potęgowania poprzez szybkie operacje przesuwania bitów w lewo lub w prawo (np. n <<3 oznacza to samo co $n \cdot 2^3 = n \cdot 8$). Oczywiście dobre kompilatory tak naprawdę robią to samodzielnie, jeśli się „domyślą", gdzie to zrobić i... będą miały włączone odpowiednie opcje optymalizacji!
  - Przerzucamy część logiki aplikacji, która przetwarza dane na serwer bazodanowy (procedury wbudowane).
  - Stosujemy wielowątkowość.
  - Wykorzystujemy znane nam cechy architektury komputera, np. zmuszamy kompilator do użycia konkretnych, zoptymalizowanych instrukcji konkretnego procesora.
  - Wykorzystujemy cechy języka programowania (np. skuteczne, choć utrudniające analizę, instrukcje `break` lub `goto` w C++).

Technik optymalizacji kodu jest naprawdę wiele i piszę o nich w tym miejscu tylko po to, aby zasygnalizować obszar, którym można się zainteresować podczas poważnego programowania!

Zadania

Zadanie 1.

Proszę rozważyć problem prawdziwości lub fałszu poniższych równań:

- $T(n^3) \in O(n^3)$;
- $T(n^2) \in O(n^3)$;
- $T(2^{n+1}) \in O(2^n)$;
- $T((n+1)!) \in O(n!)$;
- $T(n) \in O(n) \Rightarrow \{T(n)\}^2 \in O(n^2)$;
- Twój własny przykład?

Zadanie 2.

Jednym z analizowanych już wcześniej przykładów był tzw. ciąg Fibonacciego. Funkcja obliczająca elementy tego ciągu jest nieskomplikowana:

```
unsigned long int fib(int x)
{
if (x<2)
   return x;
```

```
else
   return fib(x-1)+fib(x-2);
}
```

Proszę określić, jakiej klasy jest to funkcja.

Zadanie 3.

Proszę przeanalizować taki ze swoich programów, w którym jest dużo wszelkiego rodzaju zagnieżdżonych pętli i tego rodzaju skomplikowanych konstrukcji. Czy nie dałoby się go zoptymalizować w jakiś sposób?

Przykładowo często się zdarza, że w pętlach są inicjowane pewne zmienne i to za każdym przebiegiem pętli, choć w praktyce wystarczyłoby je zainicjować tylko raz. W takim przypadku instrukcję przypisania przenosi się przed pętlę. Skraca to czas wykonywania się pętli. Podobnie, odpowiednio układając kolejność pewnych obliczeń, można wykorzystywać częściowe wyniki, będące rezultatem pewnego bloku instrukcji, w dalszych blokach — oczywiście pod warunkiem że nie zostały zamazane przez pozostałe fragmenty programu. Zadanie polega na obliczeniu złożoności praktycznej naszego programu *przed* i *po* optymalizacji i przekonaniu się na własne oczy o osiągniętym (ewentualnie) przyspieszeniu.

Zadanie 4.

Proszę rozwiązać następujące równanie rekurencyjne:

$$u_n = u_{n-1} - u_n \cdot u_{n-1} \, (dla \; n \geq 1),$$
$$u_0 = 1.$$

Rozwiązania i wskazówki do zadań

Zadanie 2.

Równanie rekurencyjne ma postać:

$$T(0) = 0,$$
$$T(1) = 1,$$
$$T(n) = T(n-1) + T(n-2).$$

Mimo dość skomplikowanej postaci w zadaniu tym nie kryje się żadna pułapka i rozwiązuje „się" ono całkiem przyjemnie. Spójrzmy na szkic rozwiązania:

ETAP 1. Poszukiwanie równania charakterystycznego:

Z postaci ogólnej SRL: $T(n) = T(n-1)-T(n-2)$ wynika, że $RC = x^2-x-1$.

ETAP 2. Pierwiastki równania charakterystycznego:

Po prostych wyliczeniach otrzymujemy dwa pierwiastki tego równania kwadratowego:

$RC = x^2-x-1 = (x-r_1)(x-r_2)$, gdzie: $r_1 = \frac{1+\sqrt{5}}{2}$ i $r_2 = \frac{1-\sqrt{5}}{2}$

ETAP 3. Równanie ogólne:

Z teorii wyłożonej wcześniej wynika, że równanie ogólne ma postać $RO = Ar_1^n + Br_2^n$ — zostawmy je chwilowo w tej formie.

ETAP 4. Poszukiwanie równania szczególnego:

Wiemy, że $u(n,m) = 0$, a zatem RS = 0.

ETAP 5. Poszukiwanie ostatecznego rozwiązania:

Poszukiwanym rozwiązaniem jest suma *RO* i *RS*:

$$T(n) = RO + RS = Ar_1^n + Br_2^n.$$

Pozostają nam do odnalezienia tajemnicze stałe *A* i *B*. Do tego celu posłużymy się warunkami początkowymi (tzn. przypadkami elementarnymi, aby pozostać w zgodzie z terminologią z rozdziału 2.) układu równań rekurencyjnych ($T(0) = 0$ i $T(1) = 1$). Po wykonaniu podstawienia otrzymamy:

$$0 = A + B,$$
$$1 = Ar_1 + Br_2.$$

Jest to prosty układ dwóch równań z dwoma niewiadomymi (*A* i *B*). Jego wyliczenie powinno nam dać poszukiwany wynik. Skończenie tego zadania pozostawiam Czytelnikowi.

Zadanie 4.

Oto szkic rozwiązania:

Załóżmy, że $u_n \neq 0$, co pozwoli nam podzielić równania przez $u_n u_{n-1}$:

$$\frac{1}{u_{n-1}} = \frac{1}{u_n} - 1.$$

Podstawmy wówczas $v_n = \frac{1}{u_n} - 1$, co da nam bardzo proste równanie, z którym już mieliśmy prawo wcześniej się spotkać:

$$v_n = v_{n-1} + 1,$$
$$v_0 = 1.$$

Jego rozwiązaniem jest oczywiście $v_n = n+1$. Po powrocie do pierwotnej dziedziny otrzymamy dość zaskakujący wynik: $u_n = \frac{1}{n+1}$.